UNIVERSIDADE TECNOLÓGICA FEDERAL DO PARANÁ

LUCAS LUIZ KAUTZMANN PELEPENKO

UMA INVESTIGAÇÃO SOBRE ADEQUAÇÃO DE MODELOS DE PREDIÇÃO DE DEFEITOS EM AMBIENTES INDUSTRIAIS DE DESENVOLVIMENTO E TESTE DE SOFTWARE

DOIS VIZINHOS

2021

LUCAS LUIZ KAUTZMANN PELEPENKO

UMA INVESTIGAÇÃO SOBRE ADEQUAÇÃO DE MODELOS DE PREDIÇÃO DE DEFEITOS EM AMBIENTES INDUSTRIAIS DE DESENVOLVIMENTO E TESTE DE SOFTWARE

Trabalho de Conclusão de Curso apresentado como requisito parcial para obtenção do título de Bacharel em Engenharia de Software da Universidade Tecnológica Federal do Paraná (UTFPR).

Orientador: Prof. Dr. Rafael Alves Paes de Oliveira

DOIS VIZINHOS

2021

Ministério da Educação
UNIVERSIDADE TECNOLÓGICA FEDERAL DO PARANÁ
COORD. DO CURSO DE ENG. DE SOFTWARE-DV

TERMO DE APROVAÇÃO

TRABALHO DE CONCLUSÃO DE CURSO - TCC

UMA INVESTIGAÇÃO SOBRE ADEQUAÇÃO DE MODELOS DE PREDIÇÃO DE FALHAS EM AMBIENTES INDUSTRIAIS DE DESENVOLVIMENTO E TESTE DE SOFTWARE

Por

Lucas Luiz Kautzmann Pelepenko

Monografia apresentada às 15 horas 30 min. do dia 13 de dezembro de 2021 como requisito parcial, para conclusão do Curso de Bacharel em Engenharia de Software da Universidade Tecnológica Federal do Paraná, Câmpus Dois Vizinhos. O candidato foi arguido pela Banca Examinadora composta pelos professores abaixo assinados. Após deliberação e conferidas, bem como achadas conforme, as alterações indicadas pela Banca Examinadora, o trabalho de conclusão de curso foi considerado APROVADO.

Banca examinadora:

Prof. Dr. Rafael Oliveira	Membro
Prof. Dra. Marisangela Brittes	Membro
Prof. Dr. Gustavo Jansen	Orientador

Documento assinado eletronicamente por (Document electronically signed by) **RAFAEL ALVES PAES DE OLIVEIRA**, **PROFESSOR DO MAGISTERIO SUPERIOR**, em (at) 22/12/2021, às 09:12, conforme horário oficial de Brasília (according to official Brasilia-Brazil time), com fundamento no (with legal based on) art. 4º, § 3º, do Decreto nº 10.543, de 13 de novembro de 2020.

Documento assinado eletronicamente por (Document electronically signed by) **MARISANGELA PACHECO BRITTES**, **PROFESSOR DO MAGISTERIO SUPERIOR**, em (at) 22/12/2021, às 10:07, conforme horário oficial de Brasília (according to official Brasilia-Brazil time), com fundamento no (with legal based on) art. 4º, § 3º, do Decreto nº 10.543, de 13 de novembro de 2020.

Documento assinado eletronicamente por (Document electronically signed by) **GUSTAVO JANSEN DE SOUZA SANTOS**, **PROFESSOR DO MAGISTERIO SUPERIOR**, em (at) 22/12/2021, às 11:02, conforme horário oficial de Brasília (according to official Brasilia-Brazil time), com fundamento no (with legal based on) art. 4º, § 3º, do Decreto nº 10.543, de 13 de novembro de 2020.

A autenticidade deste documento pode ser conferida no site (The authenticity of this document can be checked on the website) https://sei.utfpr.edu.br/sei/controlador_externo.php?acao=documento_conferir&id_orgao_acesso_externo=0, informando o código verificador (informing the verification code) **2482366** e o código CRC (and the CRC code) **588FB3CF**.

Referência: Processo nº 23064.056939/2021-12 SEI nº 2482366

Termo de Aprovação - o arquivo TermoAprovacao.pdf deve ser substituído pelo fornecido pelo Professor Responsável pelo TCC

RESUMO

PELEPENKO, Lucas. UMA INVESTIGAÇÃO SOBRE ADEQUAÇÃO DE MODELOS DE PREDIÇÃO DE DEFEITOS EM AMBIENTES INDUSTRIAIS DE DESENVOLVIMENTO E TESTE DE SOFTWARE. 70 f. Trabalho de Conclusão de Curso – Coordenadoria do Curso de Engenharia de Software, Universidade Tecnológica Federal do Paraná. Dois Vizinhos, 2022.

Um dos grandes esforços das empresas de desenvolvimento de software é a redução dos custos de suas equipes de trabalho, como também a melhora de processos para ter-se um software de maior confiabilidade. Com isto, uma das maneiras utilizadas para alcançar-se este objetivo é a utilização de ferramentas preditivas. Tais ferramentas buscam analisar e encontrar pontos do software que possuem uma maior probabilidade de predizer as áreas mais problemáticas, além de facilitar a gestão de infraestruturas e recursos humanos para analisar e resolver os defeitos de forma focada, evitando que seja inserido novos problemas nas alterações futuras e aumentando a confiabilidade do produto final. Contudo, poucas empresas utilizam tais ferramentas de predição em seu processo, isso se deve tanto pela falta de evidências de melhorias ao utilizá-las, quanto à mudança de processos internos e culturais, demandando de um alto nível de maturidade para que possa ser aplicado corretamente e trazer resultados explícitos. Como forma de contribuir com essa temática, o presente trabalho aplica uma ferramenta preditiva de defeitos e inconformidades já disponível em situações reais, para que assim sejam identificado os cenários para a aplicabilidade e diretrizes necessárias para que esta ferramente seja utilizada em ambientes reais desenvolvimento na indústria. Entre as contribuições do trabalho, destacam-se os relatos de experiência ao adicionar cerimônias de qualidade dentro de times ágeis de desenvolvimento.

Palavras-chave: predição de defeitos, Integração Contínua, Qualidade de Software, Engenharia de Software

ABSTRACT

PELEPENKO, Lucas. AN INVESTIGATION INTO THE SUITABILITY OF DEFECT PREDICTION MODELS IN INDUSTRIAL SOFTWARE DEVELOPMENT AND TESTING ENVIRONMENTS. 70 f. Trabalho de Conclusão de Curso – Coordenadoria do Curso de Engenharia de Software, Universidade Tecnológica Federal do Paraná. Dois Vizinhos, 2022.

One of the great efforts of software development companies is to reduce the costs of their work teams, as well as improving processes to have more reliable software. Thus, one of the ways used to achieve this goal is the use of predictive tools. Such tools seek to analyze and find points in the software that are more likely to predict the most problematic areas, in addition to facilitating the management of infrastructure and human resources to analyze and resolve defects in a focused way, preventing new problems from being inserted in future changes and increasing the reliability of the final product. However, few companies use such prediction tools in their process, this is due both to the lack of evidence of improvements when using them, and to the change in internal and cultural processes, demanding a high level of maturity so that it can be correctly applied and bring explicit results. As a way to contribute to this theme, this work applies a predictive tool for defects and non-conformities already available in real situations, so that the scenarios for applicability and necessary guidelines are identified for this tool to be used in real development environments in the industry. Among the contributions of the work, the reports of experience by adding quality ceremonies within agile development teams stand out.

Keywords: prediction of defects, Continuous Integration, Software quality, Software Engineering

LISTA DE FIGURAS

LISTA DE SIGLAS

CI	Integração Contínua (do inglês “Continuous Integration)
CD	Entrega Contínua (do inglês “Continuous Delivery)
CVDS	Ciclo de Vida de Desenvolvimento de Software
ES	Engenharia de Software
GQM	Métrica de pergunta de meta (do inglês “Goal-Question-Metric”)
IA	Inteligência Artificial
IDEs	(do inglês “Integrated Development Environment”)
IEEE	Instituto de Engenheiros Eletricistas e Eletrônicos
SBP	Previsão de bug de software (do inglês “Software Bug Prediction”)
SEWBOK	(do inglês, Software Engineering Body of Knowledge)
SVM	Máquinas de Vetor de Suporte (do inglês “Support Vector Machines”)
UML	Linguagem de Modelagem Unificada (do inglês “Unified Modeling Languag
UTF	Universidade Tecnológica Federal
UTFPR	Universidade Tecnológica Federal do Paraná
XP	Programação Extrema (do inglês eXtreme Programming)

SUMÁRIO

1 INTRODUÇÃO

Todo ano a indústria de software melhora suas metodologias e ferramentas para aumentar a qualidade dos sistemas de software desenvolvidos, buscando poupar esforços e diminuir os recursos gastos. Isso se deve à busca pela entrega de sistemas de qualidade com menos gastos em infraestrutura e recursos humanos. Um dos pilares para que se tenha sucesso nessa atividade consiste nas metodologias, atividades e ferramentas de testes (DELAMARO J. C. MALDONADO, 2016). Tais estratégias buscam diversos cenários possíveis com situações que podem levar o sistema a apresentar alguma falha ou comportamento inconsistente.

Existe uma diversidade de modelos e ferramentas para auxiliar o entendimento de onde e como os problemas e comportamentos inesperados ocorrem, para que se possa abordá-los de forma mais eficiente e precisa. Contudo, a maioria dos métodos são executados após o desenvolvimento de alguma funcionalidade ou manutenção, ocasionando retrabalho para as equipes de desenvolvimento e manutenção, fazendo com que uma tarefa que deveria ser de curto período, demore mais do que o previsto.

Buscando reduzir seus custos de desenvolvimento, as empresas estão investindo em melhorias nos seus processos e técnicas. Dessa forma, a partir do ano 1971, surgiram os modelos de predição de defeitos, os quais visam encontrar futuros defeitos que determinadas alterações podem ocasionar ao sistema (NAM, 2014). Devido aos múltiplos propósitos, os sistemas de software tornam-se muito distintos um dos outros; dessa forma, uma das abordagens utilizadas para a predição de defeitos é a de modelos genéricos (FENTON, 1999). Tal abordagem consiste do uso do emprego de uma estratégia de aprendizagem por meio do histórico de alterações do sistema para se adaptar aos diversos cenários, como também por meio de análise da complexidade do código e das relações entre a alteração e o problema causado (HASSAN, 2009).

A técnica de predição de defeitos trabalha para avaliar e prever se as alterações podem ou não gerar problemas, tornando assim a estimativa de tempo gasto em determinadas

tarefas mais acurada e precisa (PUNITHA, 2013). Dessa forma, o emprego de estratégias de predição de defeitos colabora com o time de desenvolvimento e testadores para validar situações nas quais existe uma maior possibilidades de ocorrer inconsistências, uma vez que quanto mais sujeito a problemas o sistema é, deve-se realizar mais testes e atividades de refatoração de forma mais focada, aumentando a garantia de que o sistema faz o que para ele é especificado (KUMAR, 2018).

Assim sendo, o presente trabalho visa estudar e analisar as diferentes formas para se utilizar e aplicar os dados gerados pela análise e predição de defeitos em ambientes ágeis, e com base nisso pretende-se prover estrategias e formas para sua aplicação em cenários reais de desenvolvimento e entrega contínua contemporâneos.

1.1 CONTEXTUALIZAÇÃO

Dentro da *Engenharia de Software* (ES), uma das diretrizes mais importantes para a garantia de qualidade do desenvolvimento, está relacionada à eficiência de processos e às atividades de testes onde consomem em media aproximadamente 20 por cento do tempo gasto (DELAMARO J. C. MALDONADO, 2016). Ambas demandam uma quantidade de tempo e recursos humanos consideráveis para serem aplicadas de forma correta. Com isto, acaba-se diminuindo a velocidade de produção de software, bem como a frequência de entregas. Um dos motivos do alto esforço nas equipes de teste é a impossibilidade de validar o sistema completo, abrangendo todos os cenários de teste possíveis (MALDONADO, 2004). Dessa maneira, sempre que é encontrada uma nova situação não mapeada, é melhorado o processo, buscando abranger o maior número de cenários, para que testes como, por exemplo, o teste de regressão, torne-se mais prático e automatizado (BERNARDO, 2008).

Diante disso, as empresas de desenvolvimento estão mudando seus processos para as integrações contínuas e o uso de DevOps (CRUZ, 2018). Por integração contínua, entende-se como sendo a prática de integrar, rotineiramente, alterações de código no *branch* principal de um repositório e testar as alterações, o mais cedo e com a maior frequência possível (VASSALLO, 2019). Nos processos de integração contínua tem-se o objetivo de acelerar as entregas de versões estáveis e agilizar o desenvolvimento de forma dinâmica, sem comprometer a qualidade. Para tanto, utilizam-se métodos ágeis de desenvolvimento, de forma que as necessidades atuais dos usuários finais sejam atendidas o mais rápido possível. Já no DevOps, o intuito principal é unir o time de desenvolvimento ao time de operações, para que assim se tenha uma visão geral de como os setores responsáveis por cada uma das partes impactam no produto final, e dessa forma buscar intermediar e

solucionar os problemas do dia a dia de forma eficiente, buscando melhorar e automatizar os processos (CHEN, 2019).

Nesses ambientes ágeis, um dos tópicos em alta é a predição de defeitos (VERHAEG, 2016). Genericamente, as estratégias de predição de defeitos utilizam ferramentas com base em modelos estatísticos e lógicos, para analisar as partes do sistema mais sujeitas a falhas e com maior necessidade de refatoração. Utiliza-se para isso, o histórico de alterações do sistema de versionamento dos códigos fonte, relacionando todo o histórico de incidentes que já ocorreram com sua causa raiz (CAVEZZA ROBERTO PIETRANTUONO, 2015). Dessa forma, torna-se possível não somente identificar cenários não encontrados, como também realizar uma melhor gestão de recursos humano. Com isso, é possível a identificação dos blocos do código com maior probabilidade de acarretar em erros, o que reduz a quantidade de novos problemas que possam ser introduzidos ao software, além de melhorar o grau de confiabilidade e qualidade de processos (PUNITHA, 2013).

1.2 MOTIVAÇÃO

Um dos principais objetivos da Engenharia de Software é melhorar continuamente a qualidade dos produtos de software. Em projetos reais, uma das maneiras de se conseguir isso é encontrando e corrigindo os problemas dos sistemas o mais rápido possível, uma vez que há um aumento considerável no custo do desenvolvimento em relação à demora que se tem para encontrar os defeitos durante as fases do ciclo de vida do software. Erros e defeitos são frequentes nos projetos reais, sendo ocasionados por erros humanos, por lógica incorreta ou até mesmo pela falta de comunicação entre o cliente e o analista, ocasionando em requisitos incorretos (EBERT, 2008).

Dessa forma, um problema identificado na fase de levantamento de requisitos, em comparação ao mesmo sendo identificado durante a implantação em cliente, tem o seu custo aumentado diversas vezes (DELAMARO J. C. MALDONADO, 2016). Isso é devido ao fato que o problema irá impactar mais pessoas, setores e fases de desenvolvimento até que seja analisada a situação novamente e realizada a devida manutenção, sem que ocasione outros erros. Prejuízos econômicos e desgastes devido a situações de falhas de software são constantes. Isto se deve ao fato de que, uma vez que é pego nas fases inicias, poucos setores irão ter o re-trabalho, além de que o cliente final não será impactado diretamente (EBERT, 2008).

Boa parte do tempo das equipes contemporâneas de desenvolvimento de software

são gastas com atividades associadas à correção de defeitos (VERHAEG, 2016). Os mesmos defeitos são identificados em homologação ou versões de produção. O ambiente de homologação trata-se de quando se está na fase de desenvolvimento e usando um ambiente controlado, o qual simula o ambiente real do cliente. Já as versões que estão em produção são aquelas executadas no ambiente do cliente com usuários finais, nas quais ocasiona os seguintes problemas:

- perda de valor do produto de software;
- custos de tempo não estimados nos projetos; e
- esforços extra da equipe de desenvolvimento com atividades de manutenção, limitando a evolução rápida do produto de software.

Uma forma de lidar com esses problemas é encontrar os defeitos em ambiente de homologação, evitando possíveis problemas não planejados (MALDONADO, 2004). Como tais problemas estão sendo identificados antes de serem liberados e impactar diretamente o usuário final, torna-se menos custosa a sua correção, considerando que irá impactar menos processos e pessoas para sua análise e correção. Adicionalmente, evita-se também a decepção dos usuários quanto a qualidade do software, que necessitará voltar versões ou operar o sistema em uma versão sujeita a falhas até a disponibilização da correção.

Dessa forma, o uso dos sistemas de software para predição de defeitos buscam por meio da análise do código fonte, quais as áreas, objetos e alterações que possuem maior probabilidade de resultar em problemas (CAVEZZA ROBERTO PIETRANTUONO, 2015). Para assim, possa ser realizado um estudo mais detalhado de determinados pontos do sistema, sendo possível realizar uma gestão de recursos humanos mais eficiente, automatizar novos cenários identificados e focar nas áreas com maior quantidade de problemas.

1.3 OBJETIVOS

Este projeto tem como objetivo avaliar a aplicação de tecnologias de predição de defeitos (ferramentas, *frameworks* e modelos preditivos) em projetos reais de desenvolvimento e manutenção de software na indústria. Além de levantar prós e contras da aplicação de estratégias de predição de defeitos em projetos reais, pretende-se evidenciar se determinados processos trazem melhorias evidentes para os membros das equipes de desenvolvimento de software. Diante do cenário apontado, os objetivos desse projeto são:

- Estudar características das tecnologias de predição de defeitos, identificando cenários de recomendações de uso dessas tecnologias na prática;
- Contribuir com o setor produtivo, por meio da incorporação de uma recomendação da ES em projetos reais de desenvolvimento de software;
- Propor diretrizes para uma maior aplicabilidade da predição de defeitos em ambientes reais de desenvolvimento; e
- Estudar uma ferramenta de predição de defeitos e aplicá-la em projetos reais de desenvolvimento de software.

Dessa forma, o objetivo principal do presente trabalho é estudar a viabilidade técnica de estratégias de predição de defeitos, utilizando uma ferramenta de predição e analise de código, as quais utilizam um modelo de previsão baseado em métricas do projeto para analisar todas as alterações efetuadas no sistema, e por meio dessas obter dados e analise para predizer o impacto de determinada alteração antes do mesmo seguir para o processo de validação e testes automatizados.

Por conta da escassez de recomendações e de estudos teórico-práticos sobre a aplicabilidade de estratégias de predição de defeitos em projetos contemporâneos com times ágeis e tecnologias modernas, as atividades de predição de defeitos têm sido negligenciadas na indústria (FENTON, 1999). O presente estudo implementa estratégias de predição de defeitos em projetos reais de desenvolvimento de uma empresa parceira do setor produtivo, identificando vantagens e destacando limitações das tecnologias preditivas adotadas. Por meio de estudos empíricos apresentados, pretende-se cooperar com a disseminação da cultura de aplicação de modelos preditivos de defeitos em ambientes de homologação, aumentando a qualidade dos produtos finais. Além disso, os resultados obtidos neste projeto serão disponibilizados nas usuais formas de divulgação de trabalhos científicos, como anais de congressos/conferências e revistas científicas, que serão utilizadas como forma de avaliação e análise dos resultados obtidos.

1.4 ORGANIZAÇÃO

Além deste capítulo introdutório, o restante deste trabalho de conclusão de curso está organizado na sequência:

- Capítulo 2: Apresenta todos os aspectos conceituais necessários para o amplo entendimento do estudo realizado, sendo divididos em cinco sub-tópicos, sendo eles:

  - Engenharia de software;
  - Testes de Software;
  - Qualidade de Software;
  - Integração contínua e DevOps; e
  - Predição de defeitos em projetos de softwares.

- Capítulo 3: Revisão de literatura narrativa, que apresenta um estudo sobre a área de foco deste estudo, apresentando os principais estudos, modelos e ferramentas de predição de defeitos;
- Capítulo 4: Metodologia e proposta, apresenta de que modo será executado o estudo para que seja possível alcançar os objetivos planejados; e
- Capítulo 5: Resultados, dados coletados durante pesquisa, e respostas das questões de pesquisa levantados com base nos mesmos.
- Capítulo 6: Considerações finais, explica as implicações do trabalho realizado a médio e longo prazo.

2 ASPECTOS CONCEITUAIS

Neste capítulo apresentam-se os principais aspectos conceituais que embasam o entendimento completo da pesquisa a ser realizada.

2.1 ENGENHARIA DE SOFTWARE

A *Engenharia de Software* (ES) é uma área especializada na especificação, manutenção e desenvolvimento de software, que engloba diversas disciplinas e constituem a base de tudo que norteia o desenvolvimento de soluções de software (PRESSMAN; MAXIM, 2016). Em suas definições, segundo Sommerville (2011), "software é caracterizado como um programa de computador e toda a documentação associada a ele". Já Pressman (2016) define software como sendo "um elemento de sistema lógico, e não físico que não se desgasta". Ou seja, pode-se afirmar que software e hardware possuem diferentes definições e um software é composto de um programa de computador, juntamente da documentação de como utilizar/manter, arquivos de configuração e até mesmo outros sistemas de software.

O termo *Engenharia de Software* surgiu na década dos anos de 1960. Naquele período ocorria a *crise de software*, que se desmembrou devido à falta de maturidade dos processos que envolvem a criação do software, em conjunto com a alta demanda de novos sistemas na época (SOMMERVILLE, 2011). Foi um período no qual não se esperava que os sistemas iriam crescer a tal ponto de se tornar altamente complexos, sendo peça fundamental em diversos setores da sociedade. Em consequência disso, ocasionou-se um grande volume de software com baixa qualidade, por não haver as definições formais e bem estruturadas de qualquer processo durante seu desenvolvimento (YU, 2009). Geralmente, os processos de desenvolvimento iniciavam-se diretamente a partir da implementação, gerando assim um mau entendimento dos requisitos. Por isso, inconsistências que acarretavam em um software que não solucionava o problema a qual foi proposto eram comuns. Esse período também era conhecido por conta de recorrentes estouros de orçamento e prazos estipulados, e consequentemente, uma má reputação para todo o setor de criação de

software. Dessa forma, empresas e clientes tinham uma visão que investir em software era um desperdício de dinheiro, uma vez que raramente o software solucionava o problema para o qual ele foi proposto (BROY, 2018).

Atualmente, existe uma infinidade de modelos, técnicas e processos que compõem a ES, os quais são compilados no SWEBOK (do inglês, *Software Engineering Body of Knowledge* ou Corpo do Conhecimento de Engenharia de Software) (BOURQUE, 2014). O SWEBOK tem como principal objetivo estabelecer um conjunto de normas e critérios comuns entre projetistas e acadêmicos de todo o mundo para a prática da engenharia de software e, por isso, é conhecido como o manual da ES. O mesmo contém os conhecimentos de varias décadas, sendo atualizado constantemente por profissionais com autoridade na área de ES. Cada publicação é revisada por diversos profissionais capacitados de cada uma das 15 (quinze) áreas de conhecimentos, aumentando o nível de especialização como também facilitando o entendimento de cada tópico.

2.2 TESTES DE SOFTWARE

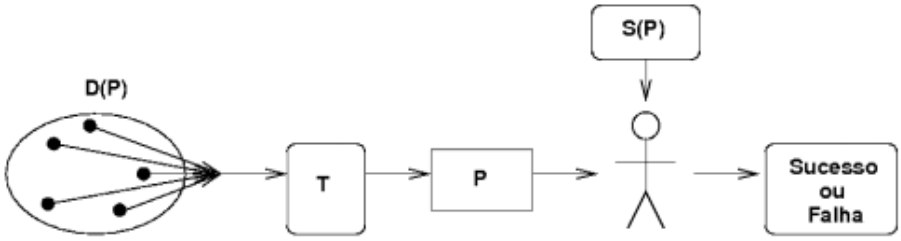
Uma das atividades mais importantes da ES consiste do uso constante de estratégias de teste de software. Segundo a norma padrão NBR ISO/IEC 12200 (12200, 1999) o teste de software faz parte do CVDS (Ciclo de Vida de Desenvolvimento de Software), no qual sua principal função é a verificação e validação de todas as partes do sistema. Fazendo uso de diversas ferramentas, técnicas e métodos, o teste de software é utilizado para validar se o que está sendo desenvolvido está em conformidade com os requisitos acordados, e se o software não apresenta qualquer comportamento indesejado. As atividades de teste de software concentram-se na revelação de falhas, para a identificação de erros que possam permitir que defeitos sejam sanados. Sendo assim, devido à similaridade dos "jargões do teste de software", a norma distingue os termos do teste de software como são apresentados a seguir:

- *Defeito*: passo, processo ou definição de dados incorretos a qual não atende a seus requisitos ou especificações e precisa ser consertado ou substituído (DELAMARO J. C. MALDONADO, 2016);
- *Engano*: Ação humana que produz um defeito (DELAMARO J. C. MALDONADO, 2016);
- *Erro*: E a aparição do defeito durante a execução do software, que se caracteriza por um estado inconsistente ou inesperado (DELAMARO J. C. MALDONADO, 2016); e

- *Falha*: Após a ocorrência de um Erro, pode-se fazer com que o resultado produzido pela execução seja diferente do resultado esperado resultando assim numa falha. incapacitando de executar dentro dos limites especificados anteriormente (DELAMARO J. C. MALDONADO, 2016).

Cada situação acima descrita pode impactar de maneiras diferentes no comportamento do sistema, mesmo que de forma indireta, sendo um potenciais problemas. Cada mudança no sistema pode ocasionar um erro em áreas sequer modificadas, por isso, independente da situação, cada bug em potencial deve ser tratado com importância, para que não afete a qualidade futuramente. Para evitar tais problemas, os modelos de análise e predição trabalham fortemente, para identificar o quanto antes e evitar que se tornem falhas.

Existem muitos fatores que podem levar o sistema a apresentar algum comportamento errôneo, sendo um deles a negligência com a utilização de boas práticas de desenvolvimento, permitindo assim, que seja introduzida uma falha no sistema (MALDONADO, 2004). Por exemplo, na implementação de uma função matemática que realiza determinado cálculo, um simples engano humano ou lógico, pode ocasionar em um erro, divergindo assim o que foi desenvolvido do que foi acordado durante o levantamento de requisitos.



Fonte: (DELAMARO J. C. MALDONADO, 2016)

Figura 1: Cenário comum de Teste

O cenário comum de um teste, como mostrado na Figura 1, é composto por um conjunto de diversas possibilidades de entradas **(D(P))**, que após **T** ser processado, geram uma saída **P**, onde se, o resultado obtido em **P** condiz com o resultado esperado, então não foi encontrado nenhum defeito, ficando por conta do testador **(S(P))** validar se as saídas de **P** estão de acordo com as especificação do programa, e se, a mesma representa uma falha ou sucesso.

Existem diversas maneiras de se testar um software. Técnicas de teste são processos técnicos ou gerenciais que ajudam a avaliação do atendimento dos requisitos com base uma determinada fonte de informação (DELAMARO J. C. MALDONADO, 2016). Tais fontes de informação podem ser, por exemplo, requisitos do sistema ou, então, código fonte.

As principais e mais amplamente utilizadas técnicas de teste são duas:

- *Técnica de teste funcional*: essa técnica de teste visa a utilizar requisitos e especificações para a criação de cenários de teste. Pelo fato de não considerar aspectos de código do sistema, essa técnica é conhecida como sendo *teste caixa-preta* (BEIZER, 1995);
- *Técnica de teste estrutural*: técnica que utiliza como referência o código da aplicação para a criação e execução de cenários de teste, levando em consideração aspectos de cobertura de código e de fluxo de controle. Pelo fato de considerar o código, é conhecido popularmente como *teste caixa-branca* (CARNIELLO; JINO; CHAIM, 2005).

Cada técnica de teste conta com diferentes critérios. Considerando as diferentes técnicas, é possível afirmar que critérios de teste consistem de diretrizes ou métodos para guiar condições e tomadas de decisões durante as atividades de teste (BERTOLINO, 2007). Considerando, por exemplo, a técnica de teste funcional destacam-se dois critérios de teste: um deles e conhecido como "teste aleatório", no qual é criado um conjunto de entradas **T** de forma aleatória para que assim sejam validados diversos subdomínios do conjunto. Já a outra técnica mais utilizada é a de "teste de subdomínios", no qual diferentemente do teste aleatório é realizado um estudo em cenários pontuais a serem validados, e encontrado qual são todos os subconjuntos de entradas **(D(P))** que irão buscar abranger todas as possibilidades ou pelo menos a maioria dos possíveis subdomínios (DELAMARO J. C. MALDONADO, 2016). Esse critério garante um conjunto de entradas logicamente mais consistente e, diferentemente do primeiro critério apresentado, irá entregar um resultado mais conciso, porém sua desvantagem em relação ao teste aleatório é o tempo gasto para se estudar e encontrar os domínios de entrada.

Por mais rigoroso e minucioso que sejam os testes, um sistema não está seguro de não possuir algum tipo de problema ao ser utilizado pelo usuário final, considerando que não há como saber todas as variâncias possíveis que influenciam no funcionamento do sistema (DELAMARO J. C. MALDONADO, 2016). Dessa forma, os testes são realizados para evitar os principais problemas e validar o máximo de possibilidades e situações possíveis, focando nas principais funcionalidades que devem ser utilizadas constantemente e, principalmente nas que possuem uma maior complexidade, onde estatisticamente possuem a maior probabilidade de apresentarem problemas e funcionamento incorreto devido a dificuldade de compreender e mapear todos os impactos que qualquer mudança pode resultar (HASSAN, 2009). Dessa forma a proposta do trabalho busca a aplicação de

uma ferramenta de analise conforme explicado nos tópicos abaixo, para que seja levantada uma estimativa mais realista.

Os tipos de testes podem variar muito dependendo de qual é o intuito do software que está sendo desenvolvido, como também em qual fase o projeto se encontra (DELAMARO J. C. MALDONADO, 2016). Por exemplo, um software que lida com vidas humanas ou situações críticas deve ser testado rigorosamente e focado não somente em qualidade ou funcionalidade, mas também em segurança, consistência e estabilidade. Isso é devido ao fato de que uma pequena falha momentânea, não identificada pelos projetistas, pode custar não somente enormes prejuízos financeiros como também vidas humanas (MALDONADO, 2004). Dessa forma, com o decorrer do tempo, surgiram diversos métodos de testes em função dos requisitos do sistema, onde cada um foca em validar determinados aspectos do produto.

Apesar da variedade de requisitos ser importante para a adoção de técnicas e critérios mais rigorosos de teste, existe um fator comum entre os diversos tipos de sistema: o teste de software é aplicado em fases (BINDER, 1996). As fases de teste são distribuídas durante o processo de desenvolvimento de software, e entre eles existem dezenas de técnicas de teste. Porém os que mais influenciam na predição de defeitos são os seguintes:

- *Teste de Unidade:* menor parte possível de realizar uma validação, sendo um componente, classe ou função. Seu intuito é validar se determinado bloco tem o comportamento esperado e se as saídas condizem com o planejado;
- *Teste de Integração:* Garante que diversas unidades do sistema ou diversos sistemas estão funcionando corretamente ao serem unidas e integradas;
- *Teste de Segurança:* tem como intuito validar os requisitos e o software para que se encontre falhas de processos ou funcionalidades que permitam um acesso a informações não autorizadas.

Como mencionado, os testes de unidade podem ser aplicados para qualquer pequena alteração realizada, forçando o sistema a passar pela nova parte para testar seu comportamento ou analisando se as entrada e saída e todo processo intermediário não possui falhas que possam afetar o comportamento. Um exemplo disso, são partes do código que nunca serão executados, falta de tratamentos nas entradas dos dados ou utilização de conversão incorreta que pode ocasionar um erro fatal. Tais validações são melhores executadas por uma ferramenta de análise de código, a qual pode identificar mais

facilmente todos esses cenários, contudo também pode ser feito pela revisão de alguém experiente na hora de aceitar ou não as alterações.

Já os testes de integração normalmente necessitam executar um roteiro automatizado mais completo dependendo da complexidade do sistema, sendo difícil identificar todas as situações que determinada parte do sistema utiliza e todos os cenários possíveis. Algumas ferramentas podem auxiliar nessa análise de forma mais automática e identificar e avaliar de forma perspicaz se existe algum outro cenário que pode ocasionar algum erro, como será demostrado neste trabalho.

Um dos principais testes e um dos menos aplicados nos processos convencionais é os teste de segurança, isso se dá ao alto nível de maturidade que se é necessário na equipe e empresa para definir métodos e processos para teste adequado nos aspectos de segurança. Uma forma de evitar o alto custo necessário para a composição de uma equipe focada nesses aspectos é a utilização de ferramentas auxiliares que ajudem nessa análise inicial, diminuindo os riscos de segurança, nesse caso, algumas ferramentas preditivas também têm algorítimos para identificar bugs explícitos que causem brechas e permitam um ataque.

2.3 QUALIDADE DE SOFTWARE

Conforme definido pelo Instituto de Engenheiros Eletricistas e Eletrônicos (IEEE – do inglês, *Institute of Electrical and Electronic Engineers*) qualidade de software é o "grau de conformidade de um sistema, componente ou processo com os respectivos requisitos" (IEEE, 1990). Já que na ES o termo qualidade está relacionado ao quão eficiente e conciso são os processos (PRESSMAN; MAXIM, 2016), uma vez que a qualidade das funcionalidades ou até mesmo, o grau de aceitação dos usuários é considerado algo abstrato.

A importância de se ter qualidade no código está tanto na experiência do usuário, que quanto menos problemas encontrar ao utilizar melhor será sua opinião do mesmo, uma vez que ninguém espera encontrar uma situação de mau funcionamento, mas também do ponto de vista do programador. Um código com qualidade facilita diversos pontos do desenvolvimento, onde requer menos tempo para entender e desenvolver, o que gera uma economia dos investimentos para novas melhorias e correções. Quanto maior a qualidade e mais bem estruturado é um software, menor é a incidências de novos defeitos, isso se deve principalmente a organização do código em menos partes, pois torna-se mais difícil compreender o que pode gerar situações não esperadas, e por isso a grande importância em se usar ferramentas para melhorar a qualidade de código e manter um padrão aceitável.

Segundo Pressman (2005), para que um software tenha qualidade é necessário implementar medidas não somente no início do processo, que é o momento no qual são levantados os requisitos. Igualmente não se deve executar medidas no final do projeto, momento no qual a maioria das funcionalidades já foram implementadas. Métricas de software devem ser coletadas em um processo gradativo que permeia por todo o ciclo de vida do desenvolvimento, abrangendo todo o escopo da ES, estando assim altamente presente em todo o processo de desenvolvimento (WHEELER; DUGGINS, 1998). Coleta de métricas devem ser frequentes e medições devem ser feitas com o intuito de melhorar o processo de desenvolvimento constantemente, aprimorando os testes e validações das inconsistências (VERNER, 2008).

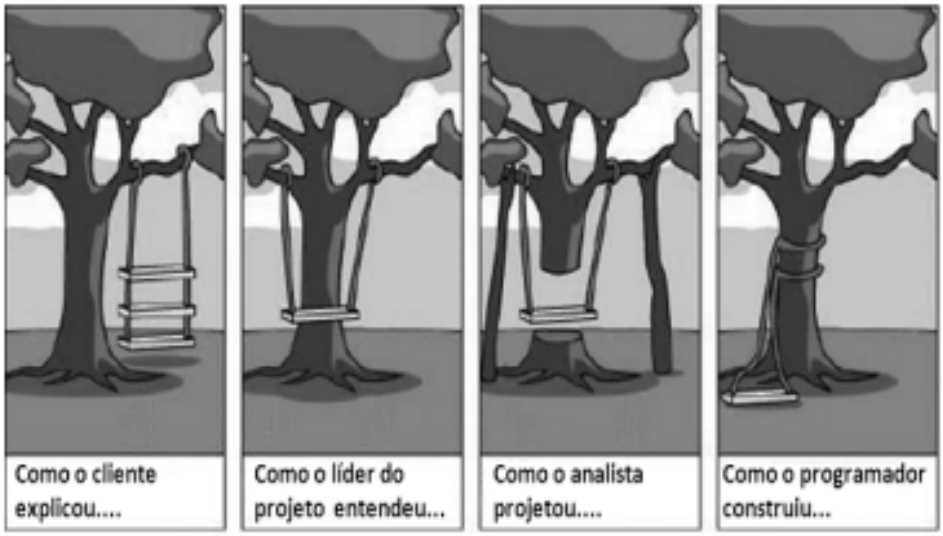
Para tanto, Pressman (2005) elenca que, para se obter qualidade de software deve-se seguir as colocações:

- "Definir explicitamente o termo qualidade de software, quando o mesmo é dito";
- "Criar um conjunto de atividades que irão ajudar a garantir que cada produto de trabalho da engenharia de software exiba alta qualidade";
- "Realizar atividades de garantia da qualidade em cada projeto de software"; e
- "Usar métricas para desenvolver estratégias para a melhoria de processo de software e, como consequência, a qualidade no produto final".

Um dos principais fatores que influenciam para que um software tenha qualidade e atenda a todas as especificações, é a *comunicação* (BERKI; GEORGIADOU; HOLCOMBE, 2004). Juntamente com a *visão geral* que todos os envolvidos (*stakeholders*) compartilham do projeto, no qual, cada uma das partes, como clientes, analistas, desenvolvedores e arquitetos devem ter visão geral comum do produto final. Caso a comunicação e visão do projeto tenham inconsistências, o projeto corre grandes riscos de não atender à solicitação inicial dos clientes (PIKKARAINEN et al., 2008). Isso deve acarretar em mau entendimento e discordância entre as partes. Existem, inclusive, casos nos quais nem o próprio cliente sabe ao certo as funcionalidades que necessita. Nesses casos, cabe ao analista de requisitos entender (ou presumir) o problema e qual a sua real necessidade, e deixar explícita como o produto deve ser e funcionar. Isso deve ser feito tanto para o cliente, quanto para todos os envolvidos no desenvolvimento.

Segundo Pressman (2002) "se você não analisa, é altamente provável que construa uma solução de software muito elegante que resolve o problema errado". Como é

demonstrado na Figura 2, é comum a existência de casos nos quais a falta de entendimento de cada um envolvidos resultará em uma visão errada da solução que está sendo desenvolvida. Essa visão míope do projeto é acumulada durante as fases de desenvolvimento, resultando em baixa qualidade do produto final.



Fonte: Adaptado de (DELAMARO J. C. MALDONADO, 2016)

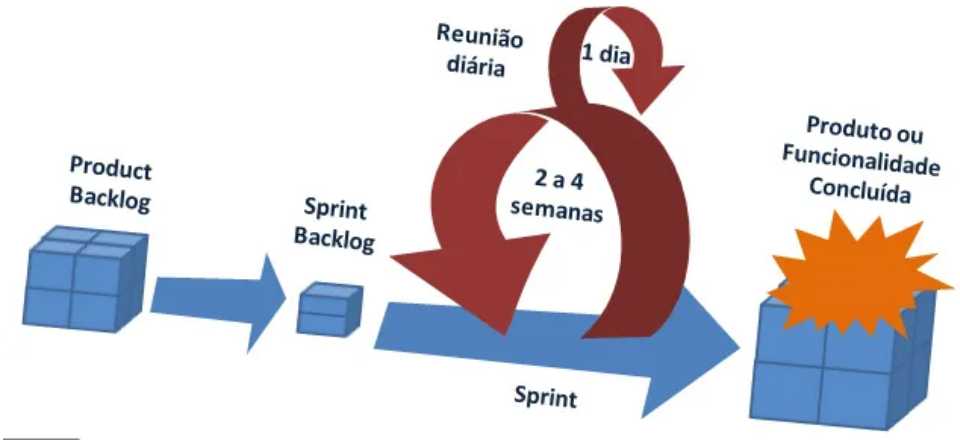
Figura 2: Problema de comunicação na produção do software

Um dos principais métodos para se analisar a qualidade do software é o Modelo de Confiabilidade, onde o termo "confiabilidade" representa a qualidade do software do ponto de vista do usuário (ORTEGA; PÉREZ; ROJAS, 2003). Os principais relatos de preocupação acerca desse tópico tiveram início por Hudson (1967). A grande importância da utilização do modelo de confiabilidade se decorre do fato de de que para o cliente este é um dos principais pontos do produto, e caso o mesmo não esteja em conformidade com o esperado, isso irá gerar grande influência na visão negativa dos usuários em relação ao produto (DELAMARO J. C. MALDONADO, 2016). Uma das formas de se avaliar a confiabilidade da visão de um sistema é levando em considerações uma gama de conceitos e variáveis durante o ciclo de vida, de modo que se obtenha um alto nível de confiabilidade, sendo eles:

- Evitar a introdução de novos defeitos durante o projeto e o desenvolvimento dos programas;
- Fazer uso de estruturas tolerantes a defeitos; e
- Remover defeitos durante a fase de teste e depuração.

A confiabilidade está diretamente ligada à teoria da probabilidade, a qual estabelece meios de avaliar a confiabilidade dos requisitos de um sistema por meio de uma escala que vai de 0 (não confiável) e 1 (confiável). Essa escala permite que se obtenha uma análise quantitativa da situação que o produto se encontra de forma precisa, uma vez que nem todas as variáveis que influenciam são constantes e podem estar sujeito a aleatoriedades (LANGE; CHAUDRON, 2005). Para contornar este problema, alguns dos métodos utilizados para se obter resultados é a utilização da média de tempo entre o surgimento de novos problemas como também a frequência com a qual aparecem. De forma geral, se a correção dos problemas encontrados estiver sendo realizada sem gerar novos problemas durante o processo, isso irá aumentar consideravelmente o grau de confiabilidade do produto, já que o mesmo terá menos complicações aparecendo futuramente.

As metodologias ágeis de desenvolvimento de software foram criadas em 2001 a partir da reunião de 17 profissionais que já utilizavam algumas das praticas ágeis, como Scrum e XP (do inglês *eXtreme Programming* – Programação Extrema). Esses profissionais reuniram-se e escreveram o manifesto ágil, contendo os principais valores e princípios do desenvolvimento de software rápido e eficiente, focando em desenvolvimento com qualidade e prontidão para mudanças de requisitos (BECK et al., 2001). Genericamente, em modelos ágeis de desenvolvimento de software, as demandas e requisitos recebidos são analisados e divididos em diversas partes de modo que não haja dependência com as demais. Assim, esses requisitos são nomeados de "histórias (ou estórias)" do usuário, e caso dependa de outra ou não possa ser concluída em um curto período de tempo, são unidas em um "épico", que contém várias histórias relacionadas. Essas histórias são, então, agrupadas em um banco de demandas conhecido como "Backlog", sendo organizadas por prioridade e esforço necessário para sua conclusão. A equipe de desenvolvimento estima tempo e demanda de esforços para a implementação das histórias e, assim, tais demandas são alocadas para um ciclo de desenvolvimento (no *Scrum*, por exemplo, esses ciclos são conhecidos como *sprints*). Uma prática comum é a atribuição de pontos a cada história, e dessa forma, é possível avaliar de maneira quantitativa cada solicitação.



Fonte: Adaptado de https://mindmaster.com.br/scrum/

Figura 3: Ciclo do modelo Ágil - Scrum

Como é apresentado na Figura 3, que representa o ciclo do modelo ágil mais conhecido – o Scrum, o desenvolvimento é dividido em pequenos ciclos de 2 a 4 semanas chamados de *Sprint*. Nas *Sprints*, as demandas são coletadas do *Backlog* e alocadas para a equipe de desenvolvimento, tendo em vista quantos pontos cada time consegue entregar de acordo com o histórico dos últimos ciclos. Essa quantidade de pontos pode variar, uma vez que a quantidade pode variar entre as equipes em função de diversos fatores, sendo sempre possível alocar mais histórias caso exista tempo na Sprint (SCHWABER, 2006). Como característica do processo, é comum que exista uma curta reunião para organizar e entender o que cada membro da equipe fez e está fazendo (ADKAR, 2018).

A principal vantagem de utilizar modelos ágeis é que a cada ciclo de desenvolvimento, é esperada uma nova versão com melhorias e correções sendo liberada para os usuários. Essa estratégia acaba por acelerar o tempo médio de resposta a problemas, facilitando também a obtenção de *feedbacks* constantes sobre o andamento do projeto e o nível de satisfação dos clientes (MAXIMINI, 2015). Isso permite para a equipe de desenvolvimento saber, de modo antecipado, se o software está indo pelo caminho correto em consonância com o que foi proposto pelos usuários. Assim, com ciclos relativamente curtos, a entrada das novas demandas/requisitos são rapidamente organizadas e absorvidas ao ciclo (RUBART; FREYKAMP, 2009). Isso forma um processo mais fluído, ágil e maleável, considerando que o software encontra-se em constante mudança e focando sempre em melhorias contínuas.

Porém mais recentemente, surgiram diversas ferramentas focadas na análise de código, como os programas de predição de defeitos, análise de segurança e padronizações do código, sendo mais rápidos e eficientes do que seres humanos para analisar o código fonte

e detectar os padrões que afetam a qualidade de código e apontar de forma precisa onde deve-se focar as atividades de manutenção e quais áreas são mais suscetíveis a problemas.

2.4 INTEGRAÇÃO CONTÍNUA E DEVOPS

Nas empresas de desenvolvimento de software normalmente, o ativo mais importante é o código fonte da aplicação (BOBROV et al., 2019). Assim, grande parte do esforço das empresas de desenvolvimento é melhorar seus sistemas e entregar sempre mais qualidade, dessa forma deve-se sempre buscar por um ótimo sistema de versionamento e gerência de configuração de software (SANCTIS; BUCCHIARONE; TRUBIANI, 2019). Atividades de gerência de configuração servem para armazenar todo e qualquer arquivo, como também gerenciar os mesmos, como permissão de leitura e escrita, e principalmente manter um histórico de cada alteração, permitindo facilmente voltar qualquer item de configuração à versão desejada ou até mesmo o projeto como um todo (KEYES, 2007). Dessa maneira é possível realizar auditorias de código fonte, como também entender quem, porquê e quando um membro do time alterou determinado arquivo (KENEFICK, 2008).

Na atualidade o principal sistema de versionamento utilizado é o Sistema de Controle GIT[1]. O GIT foi projetado por Linus Torvalds, no ano de 2005, criado para o desenvolvimento do sistema do Kernel Linux (LOELIGER, 2009). Esse sistema foi implementado em um método chamado de "merge" (mesclagem de arquivos) para unir códigos e artefatos versionados em diferentes repositórios, mantendo um histórico das alterações efetuadas, e buscando atingir um nível de performance aceitável (CONRADI; WESTFECHTEL, 1998).

Sistemas de software contemporâneos são desenvolvidos em equipes. Sendo assim, um sistema de versionamento é indispensável quando se possui mais de uma pessoa alterando qualquer item de configuração (ROSSO; JACKSON, 2013). Assim, qualquer componente que possa vir a ser modificado em um projeto como, por exemplo, arquivos de código fonte, documentos, requisitos e até mesmo outro software pode ser controlado pelo sistema de gerenciamento. O sistema de versionamento é extremamente importante para que não hajam problemas ou perda de trabalho devido à falha na comunicação entre os membros dos projetos. Isso ocorre principalmente quando há equipes distribuídas em diferentes locais.

Na abordagem tradicional de desenvolvimento e manutenção de software, várias

[1] veja: `https://git-scm.com/`

partes interessadas como, departamentos, grupos e fornecedores, são envolvidos durante o ciclo de vida do desenvolvimento de software. Dessa forma, a cultura DevOps (Desenvolvimento e Operações) surgiu como forma de melhorar ainda mais o desenvolvimento de software ágil e de qualidade. (EBERT et al., 2016)

O DevOps busca unir as duas áreas a qual a palavra é composta, referindo-se ao desenvolvimento realizado pelas equipes de desenvolvimento e testes e uni-las com as equipes de operações do sistema, os quais são os administradores responsáveis pela disponibilização dos ambientes, configuração das ferramentas, publicações das versões e monitoramento (CRUZ, 2018). O DevOps visa a segmentar a equipe de desenvolvimento entre operadores (recursos humanos que dão manutenção e estabilidade ao sistema) e desenvolvedores (recursos humanos que implementam novas funcionalidades) (CAPIZZI et al., 2019). A cultura DevOps visa a aproximar equipes de desenvolvimento e operadores, fazendo com que os mesmos cooperem entre si, melhorando a gestão das demandas e diminuindo o caos (MURPHY; KERSTEN, 2019). Um dos pontos mais importantes nesse aspecto consiste do aumento de empatia entre os membros do time de desenvolvimento, melhorando os resultados do projeto.

Fonte: (MUNIZ, 2018)

Figura 4: Ciclo de desenvolvimento e operações

Dessa forma, um analista DevOps foca sempre em resultados que apoiem o movimento ágil, uma vez que esse profissional consegue ter uma visão mais ampla de todo o ciclo de desenvolvimento e entender como determinadas ações podem impactar não somente os demais setores, como também o resultado final e a qualidade do produto entregue (BOBROV et al., 2019). Assim, o DevOps busca, por meio de diversas ferramentas e técnicas, para aprimorar todos os setores de desenvolvimento de software e contribuir de

forma geral na melhoria dos processos, para que assim, obtenha-se melhores resultados (CHEN, 2019). Utilizam-se técnicas para integrar todas as fases do ciclo de vida e garantir que funcionem de forma coesa, realizando melhorias nas ferramentas e automações de processos internos, como por exemplo, a configuração de ambientes e a publicação de novas versões.

Em consequência desses avanços nas metodologias ágeis, surgiu o termo CI (Integração Contínua – do inglês, *Continous Integration*), a qual busca unir, de forma coesa e sem perdas, todas as alterações realizadas isoladamente em ambientes locais e manter o projeto sempre o mais síncrono possível entre os envolvidos (BOBROVSKIS; JURENOKS, 2018). A prática de CI garante que todos os envolvidos no processo de desenvolvimento de software estejam alterando seus códigos com base nas últimas modificações (SMART, 2011). Nesses ambientes é comum ocorrer conflitos entre as alterações durante a junção de alterações realizadas nas mesmas partes do arquivo. Neste caso, o próprio controle de versionamento irá avisar e auxiliar a mesclar os códigos de maneira correta e segura.

Unindo o CI/CD a metodologias de testes automatizados, é possível criar uma pipeline com diversas etapas, como validação do código, geração do executável, testes automatizados e logo em seguida, o deploy em produção, o qual pode ser manual ou automático caso tenha passado com êxito em todos os passos anteriores, dessa forma agiliza-se em muito as entrega das alterações para os clientes, pois possibilita diversas automações que encurtam o tempo gasto, mas para isso é necessário ter conhecimento de como funciona o desenvolvimento e operação para unir de forma eficiente as duas áreas e proporcionar tais benefícios.

2.5 PREDIÇÃO DE DEFEITOS EM PROJETOS DE SOFTWARE

Dentro da cultura DevOps, uma das práticas recomendadas é a adoção de alguma estratégia de predição de defeitos. SBP (do inglês, *Software Bug Prediction*) trata de uma questão importante nos processos de desenvolvimento e manutenção de software, que se preocupa com o sucesso geral do software (Wan et al., 2018). Isso ocorre porque prever as falhas de software, ainda em ambiente de homologação, melhora a qualidade do software, confiabilidade, eficiência e reduz custos (Punitha; Chitra, 2013).

A SBP tem como objetivo encontrar futuros defeitos que determinadas alterações podem ocasionar ao software, utilizando-se para isso técnicas e ferramentas chamadas de "ferramentas de predição" (GIGER et al., 2012). Essas ferramentas fazem uso da análise

de dados, juntamente com o histórico, para dessa forma, aplicar modelos matemáticos que resultem em probabilidades de que alguma alteração ocasione problema, auxiliando desta maneira a encontrar defeitos mais rapidamente, economizando tempo e esforço do time em geral (ZHANG; CHALLIS, 2019). Essas técnicas surgiram em meados do ano 1971 na qual alguns pesquisadores como Ajiyama e McCabe já utilizavam métricas para avaliar a qualidade e complexidade do código.

Como cada software pode ter propósitos totalmente distintos e os códigos fonte não seguem o mesmo padrão arquitetural, os algoritmos de predição necessitam de um conjunto de dados de entrada para que sejam, literalmente, treinados (FENTON, 1999). Esse treinamento pode ser por meio de aprendizado supervisionado ou não supervisionado, no primeiro tipo, os dados já possuem rótulos que indicam qual caminho deve-se seguir e após o algoritmo ser executado, o mesmo é comparado com a resposta previamente conhecida, para assim poder ser analisado se foi ou não rotulado. Cada acerto reforça o que se levou a resposta correta, e a cada erro, é evitado o mesmo caminho, para então depois serem utilizados dados reais onde não há rótulos, obtendo-se um resultado mais preciso (ZHU, 2009). Já no aprendizado não supervisionado, não se possui rótulos nos conjuntos de teste, trabalhando dessa forma no agrupamento de padrões para encontrar relações entre eles (BARLOW, 2008).

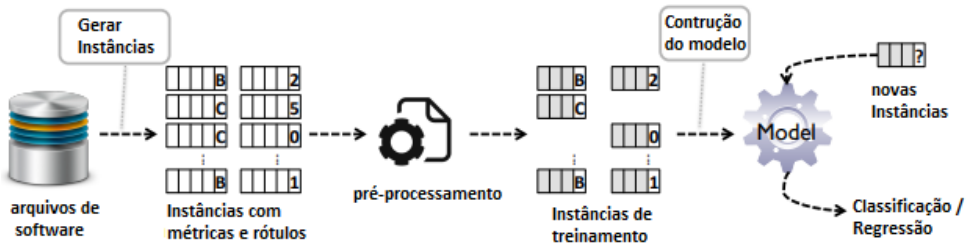
Quando os modelos de predição de defeitos surgiram, os mesmos tinham seu foco exclusivamente na coleta de métricas. Tais métricas eram utilizadas para que assim fosse possível extrair certos dados que indicavam potenciais problemas em relação ao histórico de alterações. Por exemplo, se toda vez que foi alterado um determinado objeto **X** no sistema, tal alteração gerou em um defeito na impressão de um relatório. Então, por meio de tal histórico, caso haja uma nova alteração do mesmo gênero, é altamente provável que ocasione alguma inconsistência naquela funcionalidade e, por isso, a mesmas devem ser validadas com atenção. Dessa forma, algumas relações que não são notadas podem ser encontradas por meio dessas métricas. Quanto mais alterações existirem em determinado objeto ou quanto mais pessoas modificarem o mesmo, maior será a quantidade de possíveis problemas nessa funcionalidade (KAMEI, 2016).

Após o processamento de tais dados, algumas informações podem chamar a atenção, como por exemplo, a densidade de problemas por módulo, arquivo ou função, ajudando dessa maneira a alcançar uma análise mais focada do time. Outro item importante extraído é a análise de defeitos, a qual indica a média de defeitos por linhas de código e quais partes e arquivos estão mais sujeitas a futuras falhas.

Além desses parâmetros, obtêm-se também o grau de complexidade, que utiliza um sistema de pontuação para avaliar cada operação lógica do sistema, em função do seu pior cenário, no qual as entradas representam o pior caso possível para cada cenário. Com isso, o mesmo deve demandar o maior tempo gasto para que seja executando, determinando de forma quantitativa o grau de complexidade do código fonte e auxiliando o time de testadores a validar com mais precisão determinadas situações (HASSAN, 2009). Adicionalmente, as métricas de modelos de predição podem também indicar quais situações podem ser simplificadas ou melhoradas, para que assim, evitem-se problemas, facilitando a manutenção caso necessário, uma vez que quanto mais complexa é determinada situação, torna-se mais sujeita às falhas.

Um método bastante utilizado é a regressão lógica, uma de suas técnicas utilizadas é a de que, ao encontrar um problema não mapeado anteriormente pela ferramenta de predição, quando mesmo for corrigido, é realizada a análise de todo histórico de modificações que impactaram na mesma área. Dessa forma, são mapeadas as possíveis causas até que seja encontrada a alteração que potencialmente causou o problema, assim o relacionando a esta nova situação e melhorando as novas detecções futuras (KUMAR, 2018).

A Figura 5 representa um modelo de previsão com base em aprendizado de máquina, o qual é o mais utilizado por conta da sua simplicidade e eficiência. O mesmo necessita de uma entrada de arquivos de software. Estes arquivos podem vir de um sistema de versionamento, e-mail, ou qualquer meio utilizado, e com eles, é executado um pré-processamento, no qual aplicam-se critérios para conseguir-se dados mais limpos e remover possíveis conjuntos que poderiam atrapalhar o modelo. O conjunto final de instâncias é utilizado para treinamento, no qual os mesmos são processados e comparados, prevendo se uma nova instância tem um defeito de acordo com a classificação atribuída em sua fase de treinamento.



Fonte: Adaptado de (NAM, 2014)

Figura 5: Processo de predição de defeito em software

Por mais que existam diversos *frameworks*, ferramentas e modelos de predição que

podem ser aplicados em projetos de desenvolvimento de software, os modelos preditivos têm suas vantagens e limitações. Muitos times de desenvolvimento acabam por negligenciar as atividades de predição de defeitos devido ao alto nível de maturidade de processos que se é necessário para utilizar-se e ter sucesso. Além disso, há um custo requerido para as ferramentas e um grande esforço humano para se implantar o mesmo. Por fim, é importante salientar que em muitos casos, a diretoria responsável não enxerga vantagem em utilizar e investir nesse método, entregando assim produtos de software com potenciais defeitos que poderiam ser evitados.

Dessa forma, um problema recorrente associado à aplicação de SBP em ambientes reais de desenvolvimento e manutenção de software é a falta de diretrizes para aplicações práticas em projeto contemporâneos. Neste contexto, três principais problemas impedem a transferência tecnológica da academia para a indústria:

- necessidade de modelos de predição flexíveis e adaptáveis para projetos reais;
- necessidade de ferramentas/*frameworks* de apoios; e
- necessidade de processos a serem seguidos para guiar a aplicação de modelos de predição de defeitos.

2.6 TAXONOMIA DE PREDIÇÃO DE DEFEITOS

Existem diversas técnicas que são utilizadas para processar os dados de métricas em esquemas de predição de defeitos. Algumas dessas técnicas têm foco em velocidade de resposta e simplicidade, enquanto algumas focam no detalhamento mais preciso e completo, custando dessa forma mais tempo e processamento para se obter dados e estatísticas mais precisas (CAVEZZA ROBERTO PIETRANTUONO, 2015). Outras são compostas por diversas técnicas para assim se obtenha um melhor resultado, buscando as melhores características de cada (VERHAEG, 2016). Segue algumas das técnicas utilizadas e suas principais características:

- *Regressão logística:* são frequentemente utilizadas com o objetivo de predizer (classificar) utilizando-se com base os valores de suas variáveis auxiliares. Além disso, a técnica visa estimar a probabilidade de determinada variável assumir um valor em função dos de outras variáveis já conhecidas (PAVLYSHENKO, 2016). É amplamente utilizado devido à facilidade de lidar com situações independentes e a facilidade de

realizar agrupamentos, utilizado-se de pouquíssimas suposições e resultado de alto nível de confiabilidade. Esse modelo não faz suposições quanto à forma funcional das variáveis independentes, tornando mais amplo em comparação com outras técnicas como a análise discriminante que leva em consideração esse fator (ZOU YONG HU, 2008). Dessa forma, a Regressão logística é amplamente utilizada em diversos cenários e situações como pesquisa politica, marketing, avaliação de liberação de crédito, *machine learning*, medicina, predição de defeitos em projetos de software e entre outros. Como o próprio nome sugere a mesma utiliza-se basicamente de uma função matemática juntamente com regressão, onde após um conjunto de observações, é possível categorizar os dados e predizer possíveis relações e probabilidades dos conjuntos pertencer ou não a determinado grupo (PAVLYSHENKO, 2016), como por exemplo, avaliar um grupo de pessoas a qual será liberado credito bancário, e com base em alguns dados como idade, gênero, escolaridade, nacionalidade prever quais grupos possuem maior inadimplência, puramente com base nos dados e nas experiências anteriores.

- *Análise discriminante*: Tem seu foco em reconhecimento de padrões, separando em grupos e classes, e determinando dessa forma padrões na estruturação do código. Essa técnica resulta em combinação linear, sendo possível a aplicação de algoritmos matemáticos para encontrar um bom preditor. A mesma leva em consideração que o grupo de dados de testes e de produção são semelhantes, o que torna o modelo mais falho já que na prática podem variar muito quando expostos a cenários reais e a um grande volume de entradas (JI, 2008).

- *Árvore de Decisões*: É composta por *Ramos, Nós, Sub-Nós e Folhas*, como representando na Figura 6 (CHIU YUH-RU YU, 2016). Onde cada resposta à determinada pergunta de sim ou não a divide em duas novas partes, até que todos os critérios sejam encontrados em todas as suas subdivisões. Porém seu grande desafio é lidar com valores e atributos desconhecidos onde não considera todas as suas possibilidades e variações de forma satisfeita (GAVANKAR, 2017).

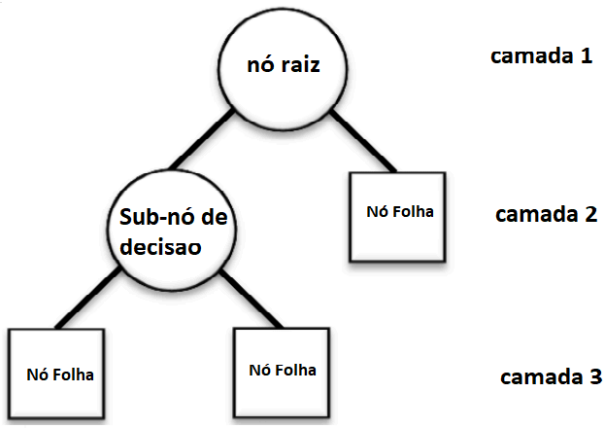


Figura 6: Formato de uma Árvore de Decisão
Fonte: Adaptado de (CHIU YUH-RU YU, 2016)

- *Naive Bayes:* Utiliza do teorema de Bayes para definir classificadores probabilísticos. Este modelo trabalha isolando os pontos e tratando de forma independente cada característica e informação, para no final definir a probabilidade da ocorrência, com base nos critérios estabelecidos para cada parte. Sua vantagem é que necessita de pouco tempo e um pequeno conjunto de dados para seu treinamento, porém deixa a desejar se comparado a outros métodos (HARAHAP; EKADIANSYAH; SARI, 2018).

- *Random forest:* O algoritmo funciona com base em varias árvores de decisão, executando-o diversas vezes com amostras de afirmações vinda do banco de dados e dessa forma, realiza predições com base na média dos valores ou resultados que mais aparecerem (CHAKRABORTY; SUR, 2019).

- *Support Vector Machines (SVM)*: Utiliza o meio de aprendizado supervisionado para dividir cada conjunto finito de entradas em duas diferentes categorias, definindo qual a disparidade entre elas. Para o contexto desse método, quanto mais parecidas, mais próximas serão representadas. Este modelo funciona bem quando não há um grande volume de dados e os mesmos não são consideravelmente difusos ou possuem ruídos, porém funciona bem em domínios complexos, onde exista uma margem de separação bem estabelecida (HEARST S.T. DUMAIS, 1998).

2.7 CONSIDERAÇÕES FINAIS

Este capítulo apresentou conceitos importantes para o entendimento do trabalho de pesquisa a realizado. Aspectos associados às principais atividades da Engenharia de Software foram apresentados. Destacaram-se processos ágeis de produção de software, teste de software, cultura DevOps e estratégias de predição de defeitos.

3 REVISÃO DE LITERATURA NARRATIVA

Para melhor embasar e garantir o ineditismo da pesquisa apresentada, foi realizada uma revisão narrativa de literatura sobre o tema de predição de defeitos em diversos artigos e referências que abordam o assunto de testes, DevOps, e principalmente ferramentas preditivas e seus impacto vide sua utilização.

3.1 INTRODUÇÃO

O presente trabalho utiliza-se de uma revisão narrativa. Revisões narrativas são úteis na busca do entendimento do cenário atual da predição de defeitos em estudos teóricos e práticos. Adicionalmente, é possível identificar os modelos de predição utilizados e quais suas vantagens e desvantagens, além de quais as ferramentas que estão disponíveis e se destacando em meio ao cenário.

Diferentemente de outros estudos realizados, o intuito dessa revisão é avaliar, no cerne da indústria, as dificuldades e limitações da utilização da ferramenta preditivas. Além disso, pretende-se evidenciar se o emprego de modelos de predição trazem vantagens explícitas que compensam suas dificuldades e esforços para implantação e utilização.

3.2 FERRAMENTAS DE PREDIÇÃO DE DEFEITOS

Com a popularização de DevOps e com a atenção dada para questões de qualidade de processos de desenvolvimento de software e maturidade de processos, tanto academia quanto indústria têm voltado suas atenções para o desenvolvimento de ferramentas de predição de defeitos (BOBROV et al., 2019). Nesta seção apresentam-se duas ferramentas que se destacam por permitirem a aplicação de estratégias de predição de defeitos em projetos de sistemas de software contemporâneos.

3.2.1 COMMIT-GURU

O *Commit*-Guru[1], trata-se de uma ferramenta de predição de falha que utiliza de um modelo de regressão para analisar todas as alterações efetuadas no sistema. O mesmo conta com um modelo preditivo para uma análise de cada alteração realizada no código fonte, montando dessa forma um modelo para cada projeto validado. Assim, por meio dessas métricas extraídas do conjunto de dados a ferramenta consegue identificar as alterações com maior probabilidade de conter falhas.

O principal objetivo do *Commit-Guru* é fornecer aos desenvolvedores e gerentes as análises e previsões sobre os risco que o código possui, identificando assim alterações que têm maiores probabilidades de introduzirem *bugs* (ROSEN BEN GRAWI, 2015). A ferramenta possui uma interface gráfica para visualização das métricas como mostrado nas Figuras 7 e 8. Adicionalmente, a ferramenta serve como um servidor responsável pelo processamento e execução dos algoritmos de predição e construção dos modelos.

A ferramenta utiliza como base a suposição que, entre ocorrer problema e o mesmo ser identificado, um tempo precioso de projeto e de recursos humanos pode ser gasto. Dessa forma, são utilizados dados de até 3 (três) meses para que seja executado o modelo preditivo utilizando regressão lógica. O mesmo usa como base 13 (treze) critérios definidos e extraídos dos dados nos quais, a cada critério, se a métrica for estatisticamente significativa e não exista nenhuma métrica anteriormente mais significativa, ela será adicionada ao modelo final (ROSEN BEN GRAWI, 2015). A interface gráfica da ferramenta consiste de uma visão geral do repositório como demostrado na Figura 7. A mesma apresenta o total de *commits* que pode têm mais chances de apresentar defeitos (Número 1, Figura 7). uma visão de diversas métricas e informações referente ao projeto (Número 2, Figura 7) e também por uma legenda na parte inferior (Número 3, Figura 7).

[1] veja `http://commit.guru/`

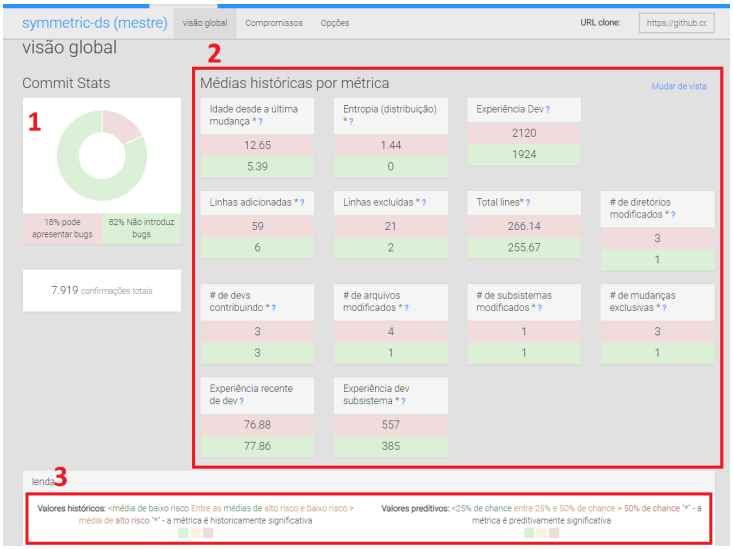


Figura 7: Visão Geral das Métricas

Fonte: Adaptado de `http://commit.guru/`

Já a Figura 8 apresenta uma visão mais detalhadas de todas as alterações enviadas, contendo um filtro com diversas opções para facilitar a análise (Número 1, Figura 8). Adicionalmente, a figura também apresenta duas marcações que representam a probabilidade de falha e também o intuito da alteração que representa se é uma melhoria, correção de falha, adição de recursos entre outros tipos que o modelo pode relacionar (Número 2, Figura 8). Por fim, a ferramenta disponibiliza também um bloco de informações e métricas que se refere ao *commit* específico (Número 3, Figura 8).

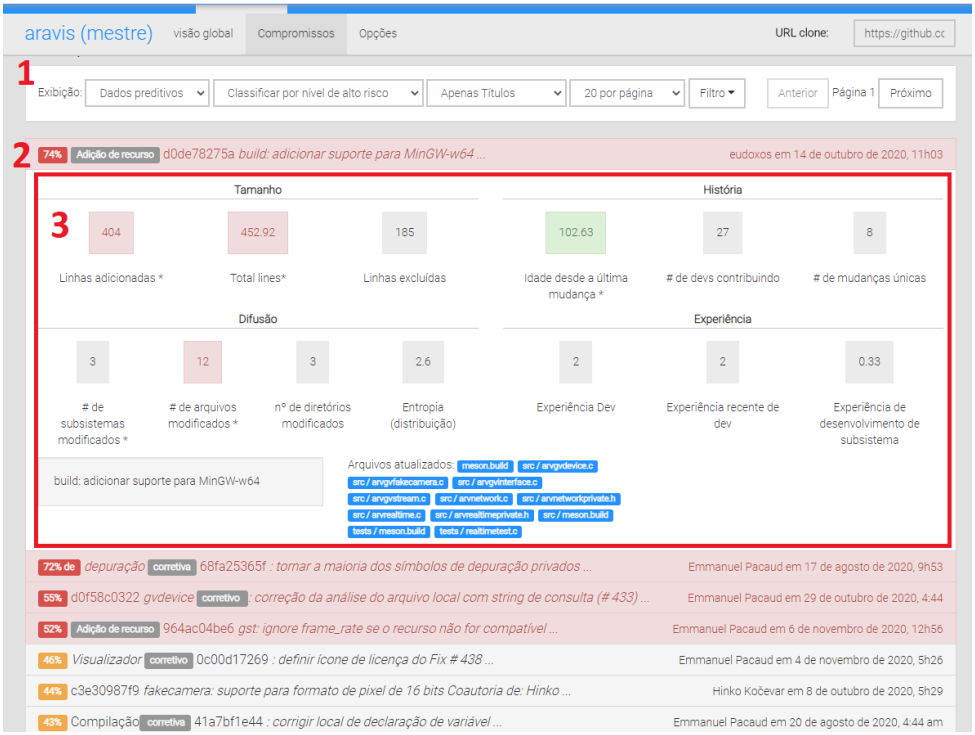


Figura 8: Análise por Commits

Fonte: Adaptado de `http://commit.guru/`

Deste modo, foi visto que a ferramenta possui todas as métricas e funcionalidades necessárias para uma boa ferramenta preditiva, onde o principal problema notado foi a falta de integrações com demais ferramentas de CI/CD.

3.2.2 SONARQUBE

O *SonarQube*[2] é uma plataforma desenvolvida pela *SonarSource* no ano 2007. Trata-se de uma ferramenta de código aberto, a qual tem o intuito de realizar revisões automáticas, utilizado de análise estatísticas para detectar potenciais defeitos, códigos mal estruturados, pontos candidatos de melhoria/refatoração e vulnerabilidades de segurança (CAMPBELL, 2013). Possui suporte a mais de 27 linguagens de programação como também suporte para diversos sistemas de integração contínua como Jenkins[3], Azure, entre outros. Além de possuir a possibilidade de implementação durante revisão de código, permitindo uma análise antes do código fonte ser enviado para o repositório principal, evitando assim quebrar a estabilidade do mesmo (CAMPBELL, 2013).

[2] veja: `https://www.sonarqube.org/`
[3] veja: `https://www.jenkins.io/`

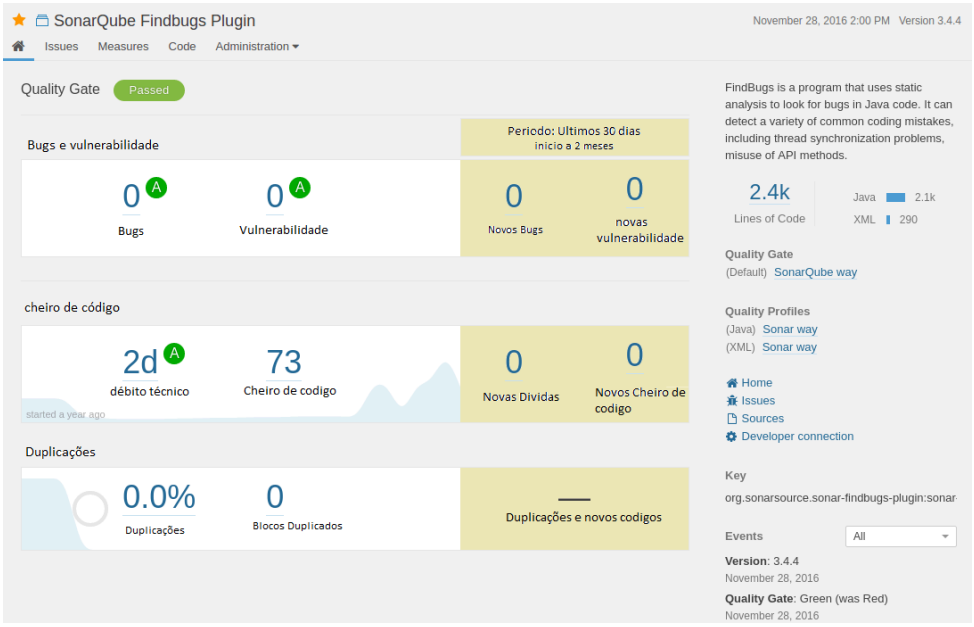


Figura 9: Painel do SonarQube

Fonte: Adaptado de https://en.wikipedia.org/wiki/SonarQube

Como demostrado na Figura 9, a interface principal da ferramenta é bastante simples e apresenta algumas das informações principais, como complexidade de código, duplicação de trechos, padronização, métricas, dívidas técnicas, vulnerabilidades e, principalmente, potenciais *bugs*. Para conseguir essas informações, a ferramenta utiliza um modelo que penetra nas "camadas" do código fonte, e a cada nível, produz valores métricos e estatísticos sobre o código. Assim, o *SonarQube* monta uma análise de código detalhada e aponta possíveis falhas.

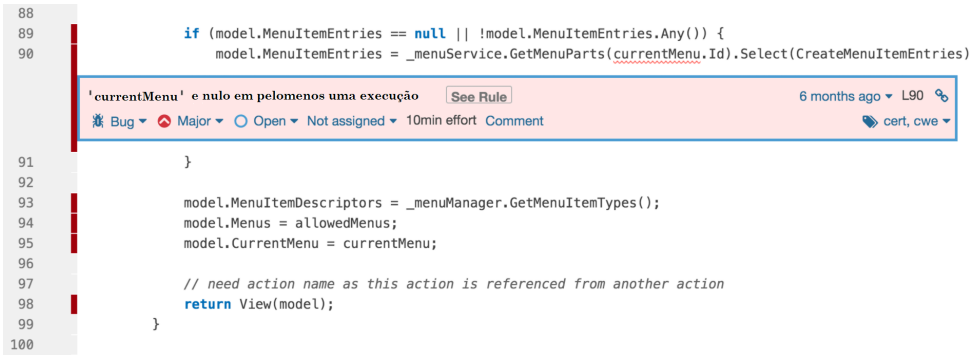


Figura 10: Exemplo de detecção com SonarQube

Fonte: Adaptado de https://www.sonarqube.org/

Ao encontrar determinadas situações comuns com base nas regras de negócio da

própria ferramenta, ela é capaz de apontar o local do problema, sugerindo ainda como deve ser feita a correção (GUSTAFSSON, 2019). O funcionamento da ferramenta é demonstrado na Figura 10. Dessa forma, nota-se que a ferramente é completa e atende a todas as expectativas do projeto.

3.3 ANÁLISES E DISCUSSÕES SOBRE A REVISÃO NARRATIVA

Ambas as ferramentas apontadas funcionam para diversos cenários e linguagens de programação, cobrindo as necessidades do setor produtivo e podendo ser implantadas em muitos projetos reais. Por meio do estudo realizado nesta pesquisa foi possível notar que as ferramentas estudadas possuem integração com diversos sistemas de versionamento. As ferramentas uma vez configurada corretamente, já é possível processar o repositório e gerar todas as informações necessárias para avaliações empíricas e realizações de análises em projetos legados e projetos reais em andamento.

A partir do estudo conduzido e comparando as funcionalidades disponíveis no site oficial de cada ferramenta e também em suas demostrações, notou-se que a diferença mais significativa entre as duas ferramentas trata-se do fato que o *commit-guru* requer a finalização de processamento do código que é realizado pelo servidor externo. Ao contrário disso, o *SonarQube*, que é executado em ambiente local, permite maior controle e confidencialidade de aspectos de código e regras de negócio, uma vez que os repositórios do projeto em análise não necessitam estar disponíveis publicamente para que possam ser executados no projeto, outro fator determinante e que o *commit-guru* não suporta utilização do gitlab de forma nativa, dificultado a implantação no cenário dessa pesquisa.

Tabela 1: Comparação entre as ferramentas SonarQube e Commit-Guru

Funcionalidades	*SonarQube*	*commit-guru*
Detecção de Defeitos	▬	▬
Detecção de Vulnerabilidades de Segurança	▬	
Integração Gitlab	▬	
Integração GitHub	▬	▬
Integração CI/CD	▬	
Código aberto	▬	
Pre-commit	▬	
Pós-commit	▬	▬
Ignorar arquivos específicos	▬	
Servidor interno	▬	

Do ponto de vista comparativo, como visto na Tabela 1 para a execução de avaliações empíricas em projetos reais, o *SonarQube* se mostrou mais vantajoso. Isso se dá, principalmente, pelo fato dele ser uma ferramenta de código aberto, o que permite a

realização de adaptações caso necessário. Além disso, a *SonarQube* possui uma comunidade de usuários ativa, permitindo um suporte a eventuais problemas que podem vir a surgir durante a realização da pesquisa. Adicionalmente, destaca-se que o *SonarQube* também pode ser implantando para ser executado antes de o código ser enviado e aceito no repositório principal, evitando dessa forma que uma alteração mal estruturada introduza defeitos para o sistema em definitivo. Outra vantagem do *SonarQube* são as funcionalidades de detecção de vulnerabilidades que aumentam a qualidade e confiança ao produto, dessa forma a aplicação que foi utilizada ficou sendo o *SonarQube* por ser mais completo e se encaixar melhor no cenário proposto.

3.4 CONSIDERAÇÕES FINAIS

A presente seção apresentou os resultados de uma revisão de literatura narrativa que foi conduzida com dois objetivos:

1. Identificar as principais estratégias de predição de defeitos apontadas na literatura; e
2. Avaliar ferramentas candidatas para estudos empíricos a serem realizados nas próximas etapas da pesquisa.

As técnicas de predição de defeito foram identificadas e categorizadas. Por fim, notou-se que a ferramenta *SonarQube* se mostrou mais adequada para a realização de estudos empíricos decorrentes das próximas fases do projeto.

4 METODOLOGIA E PROPOSTA

Este capítulo apresenta a metodologia, retomada de objetivos, atividades e cronograma de condução para a pesquisa proposta.

4.1 OBJETIVO

Essa pesquisa visa avaliar a aplicação as tecnologias de predição de defeitos (ferramentas, *frameworks* e modelos preditivos) em projetos reais de desenvolvimento de manutenção de software, destacando seus pontos positivos e debatendo suas limitações. O ineditismo do trabalho é focado no apontamento da visão dos desenvolvedores sobre o uso desse recurso em cenários reais com processos de desenvolvimento ágeis.

4.2 METODOLOGIA

Por definição, SBP é o processo de determinação de partes de um sistema de software que pode conter defeitos (SON et al., 2019). A aplicação de modelos de predição no início do ciclo de vida do software permite que os profissionais possam concentrar sua mão de obra de teste de uma maneira que as peças identificadas como "propensas a terem defeitos" sejam testadas com mais rigor em comparação com outras partes do sistema de software (CATAL; DIRI, 2009). Isso leva à redução dos custos de mão de obra durante quando o produto está em ambiente de homologação e também relaxa o esforço de manutenção com produtos de software em produção (SON et al., 2019). Genericamente, modelos SBP são construídos usando duas abordagens:

1. *usando métricas de software*: propriedades mensuráveis do sistema de software em desenvolvimento (RADJENOVIć et al., 2013);
2. *usando dados de falhas/defeitos*: de projeto de software semelhante ou do próprio software em desenvolvimento (MENZIES et al., 2010).

Durante o escopo da pesquisa foram utilizadas as duas abordagens acima. A primeira durante o processamento dos projetos e de todas as novas alterações que ocorrerem durante o fluxo normal do desenvolvimento, para que dessa forma possam ser mensurados os dados de cada projeto. Já na segunda abordagem, foi comparado com os dados disponíveis de antes de sua aplicação, tendo em conta que nem todas as métricas poderão ser comparadas, visto que não foi possível medir alguns dos dados coletados antes de sua aplicação.

Com o intuito de contribuir com os problemas supracitados e promover uma ponte de conhecimento entre teoria e prática na aplicação de modelos de predição de defeitos em projetos reais, o presente projeto de conclusão de curso visou a aplicação de modelos de predição em projetos reais de desenvolvimento de software. Contando com a parceria de uma fábrica de desenvolvimento de software com projetos e times diversos, a ideia central do projeto foi propor e aplicar processos e ferramentas/*frameworks* de predição de defeitos em projetos reais de desenvolvimento de software, revelando potencialidades e identificando limitações. Estudos empíricos diversos foram planejados e executados.

Para alcançar o objetivo foi estruturado um fluxo que se adéqua ao cenário da empresa alvo do estudo, ao qual deve-se seguir para cada nova demanda do sistema, iniciando ao receber a solicitação até ser concluído o desenvolvimento do mesmo, visando assim alcançar maiores resultados na aplicação da ferramenta, conforme Figura 11.

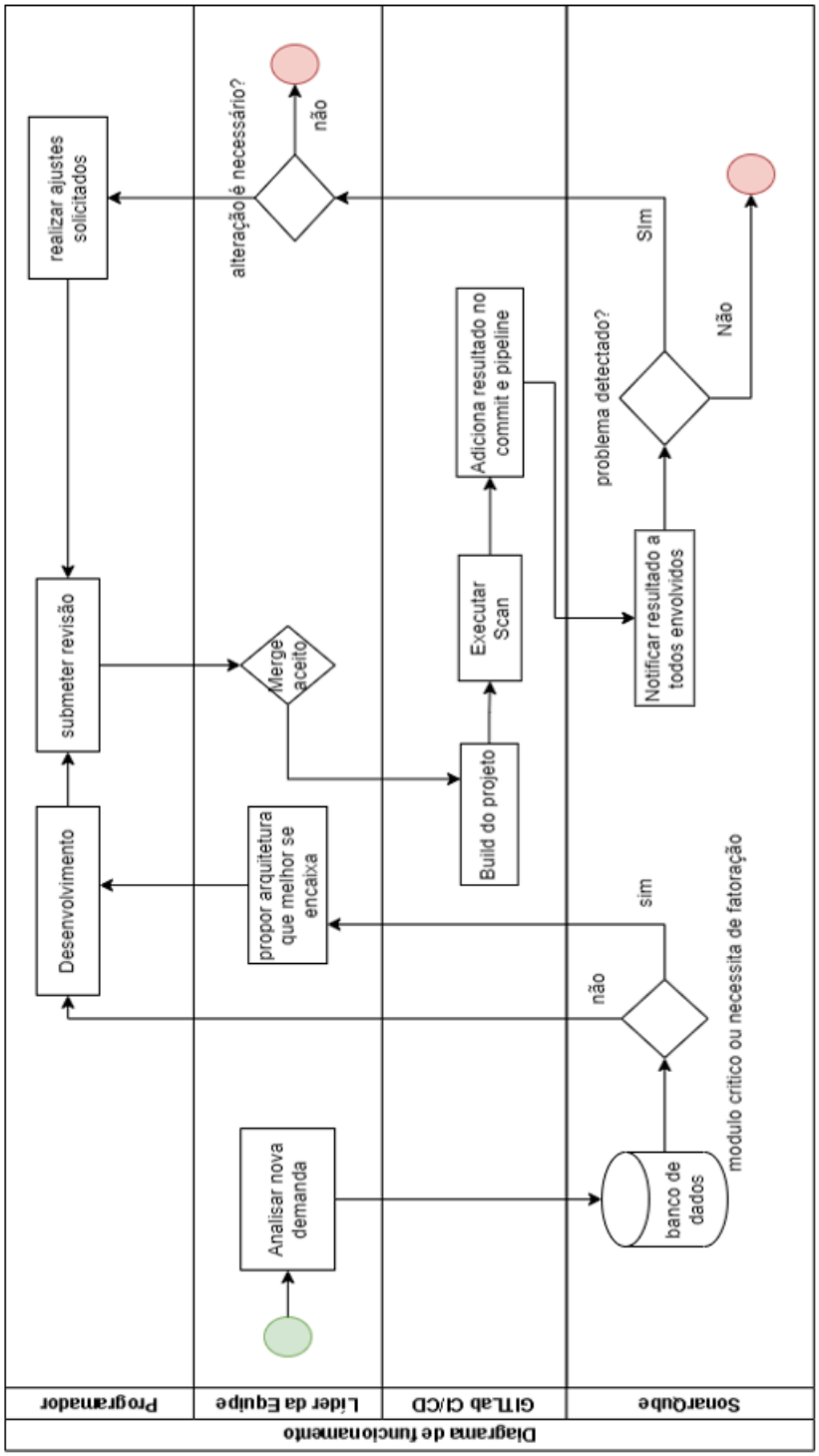


Figura 11: Fluxo de desenvolvimento para novas alterações

Fonte: Pelepenko, Lucas, 2021

4.2.1 QUESTÕES DE PESQUISA

A metodologia estabelecida para a realização da pesquisa foi balizada em Questões de Pesquisa. Sendo assim, os estudos como o proposto nesse trabalho de conclusão de

curso cooperaram para que as seguintes perguntas fossem respondidas:

- A utilização de ferramentas preditivas diminui o tempo gasto em manutenção?
- Qual o esforço da utilização e configuração da ferramenta no dia a dia?
- Qual a melhor estratégia de utilização da predição, Pós-*commit* ou pré-*commit*?
- Qual o efeito da aplicação de estratégia de predição de defeitos nos membros da equipe de desenvolvimento de software?

Visando elaborar uma estratégia para responder as perguntas acima mencionadas, os pesquisadores optaram por utilizar a abordagem conhecida como GQM (do inglês, *Goal-Question-Metric*) (BASILI; CALDIERA; ROMBACH, 1994). O GQM consiste de um método objetivo que planeja medições de forma que estejam baseadas em objetivos específicos da medição.

Sendo assim, algumas etapas do processo de realização do projeto foram respondidas pelo estabelecimento de métricas a serem aferidas com cenários empíricos de desenvolvimento de software.

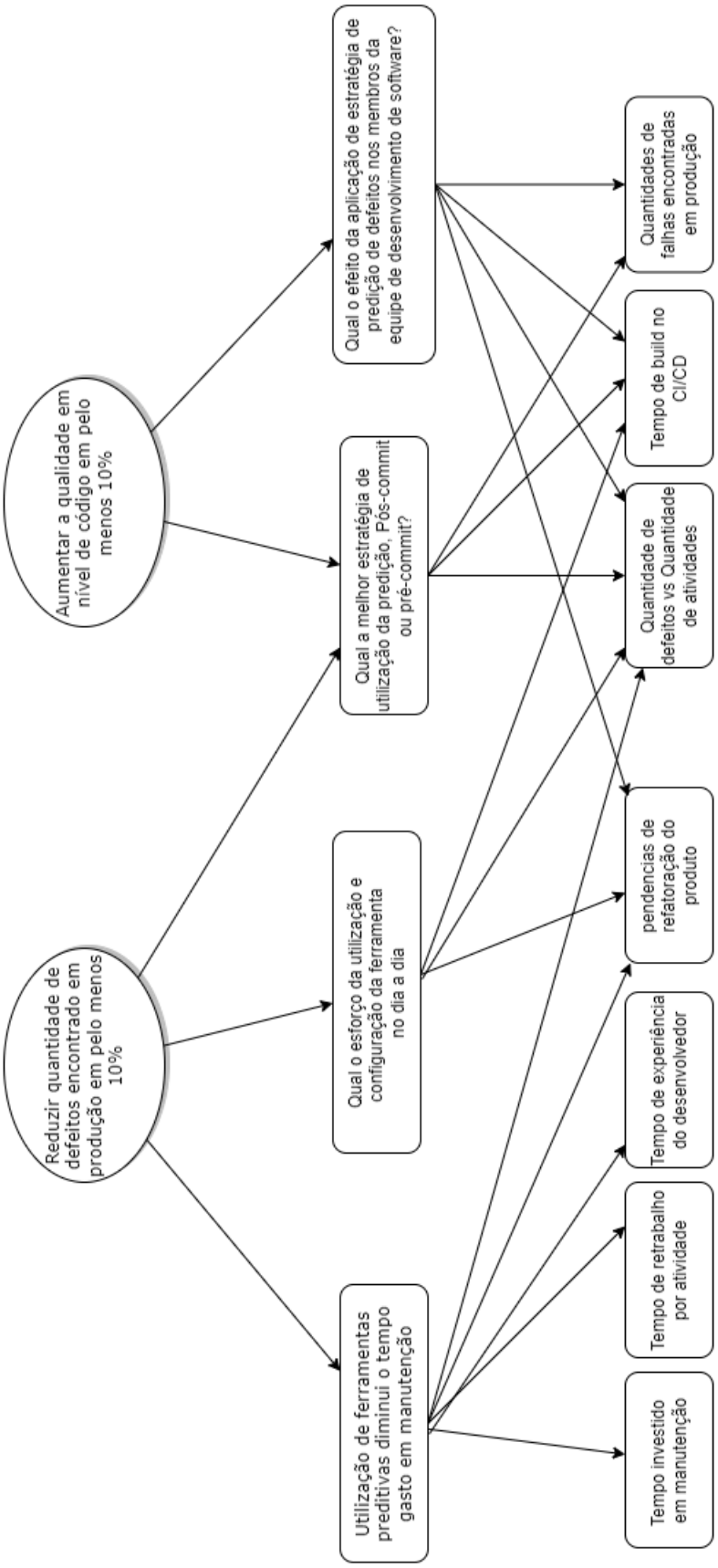


Figura 12: Goal Question Metric para predição de defeitos em ambientes reais

Fonte: Pelepenko, Lucas, 2021

Conforme demostrado na figura acima foram estipulados diversos resultados chave para que seja possível responder as perguntas listadas acima, sendo os seguintes objetos da pesquisa:

- Tempo investido em manutenção: quantas horas são gastas pela equipe em função de atividades já entregue para clientes.
- Tempo de retrabalho por atividade: quantas horas são gastas em ajustes após a entrega da atividade na *Sprint*.
- Tempo de experiência do desenvolvedor: tempo de experiência dos integrantes da equipe.
- Refatoração do produto: quantidade de horas necessárias para refatoração e melhoria do código fonte.
- Quantidade de defeitos *vs* Quantidade de atividades: comparativo entre quantidade de bugs e quantidade de atividades alocada na sprint.
- Tempo de build no CI/CD: Tempo necessário para realizar todas as operações após a entrega de determinada alteração, como a geração de arquivos executáveis e testes automatizados.
- Quantidades de falhas encontradas em produção: quantidade de falhas que foram encontradas em ambiente de produção durante a sprint.

4.2.2 EQUIPE

Além do acadêmico que conduz a pesquisa, o projeto conta com a seguinte equipe:

- Prof. Dr. Rafael Oliveira: orientador principal do projeto, com pesquisa na área de ES e foca principal em teste de software, aspectos de qualidade e processos de desenvolvimento de software contemporâneo;
- Equipe A: Equipe com 10 (dez) integrantes responsáveis por alterações em determinados produtos que utilizam linguagem de programação em sua maioria java.
- Equipe B: Equipe com 4 (quatro) integrantes responsáveis por alterações em produtos de uso interno da empresa como automações, gerenciamento e demandas especificas e produtos de uso do cliente no sentido de automatizar e facilitar certas demandas do suporte. Utiliza principalmente linguagem de programação JavaScript.

4.3 CONSIDERAÇÕES FINAIS

A presente seção apresentou a metodologia e estratégia empírica desenhada para a conclusão do estudo. Utilizando a técnica GQM, questões de pesquisa e métricas foram definidas para a condução de um estudo empírico em um ambiente real de desenvolvimento de software. Com base nisso, foi coletado as métricas das questões de pesquisa levantada, conforme próximo capítulo.

5 RESULTADOS

O presente capítulo visa a apresentar e visualizar os dados coletados a partir da pesquisa conduzida.

5.1 DADOS COLETADOS

Nessa seção, apresentam-se e analisam-se os dados coletados na aplicação da ferramenta preditiva em equipes que utilizam metodologias ágeis. A coleta foi realizada durante doze semanas (seis sprints com duração de quinze dias), sendo dividido em duas equipes, nas quais uma delas utiliza a linguagem Java no desenvolvimento de seus projetos, e a outra equipe utiliza JavaScript. Sendo importante ressaltar que na Sprint 1 (um) e 2 (dois), não foram utilizadas a execução da ferramentas SonarQube, pois foi o período de análise do comportamento dos times antes de sua aplicação, onde os times já possuíam tais dados.

No gráfico da Figura 13 foi feita a análise de quantas horas foram investidas em manutenção somando todas as atividades do gênero em cada equipe durante as seis sprints, nas duas primeiras *sprints* não tendo a aplicação da ferramenta preditiva, e nas demais sim.

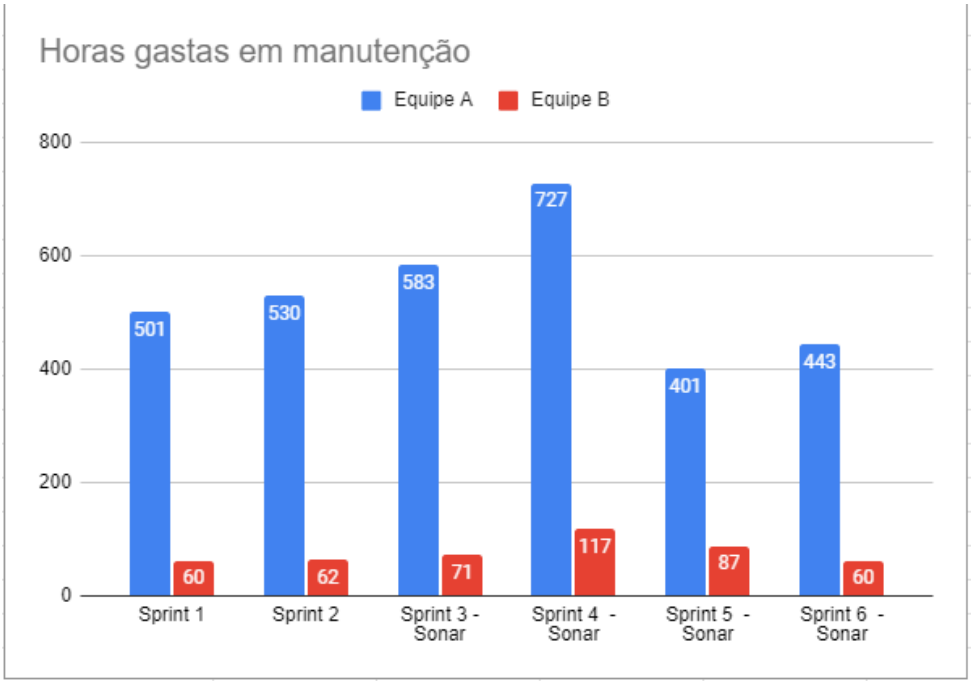


Figura 13: Tempo investido em manutenção por equipe

Na Figura 14 foi realizada a média de quantas horas foram gastas na realização de manutenções, dessa forma e possível visualizar se a utilização da ferramenta trouxe ou não melhoria no tempo gasto de atividades de manutenções.

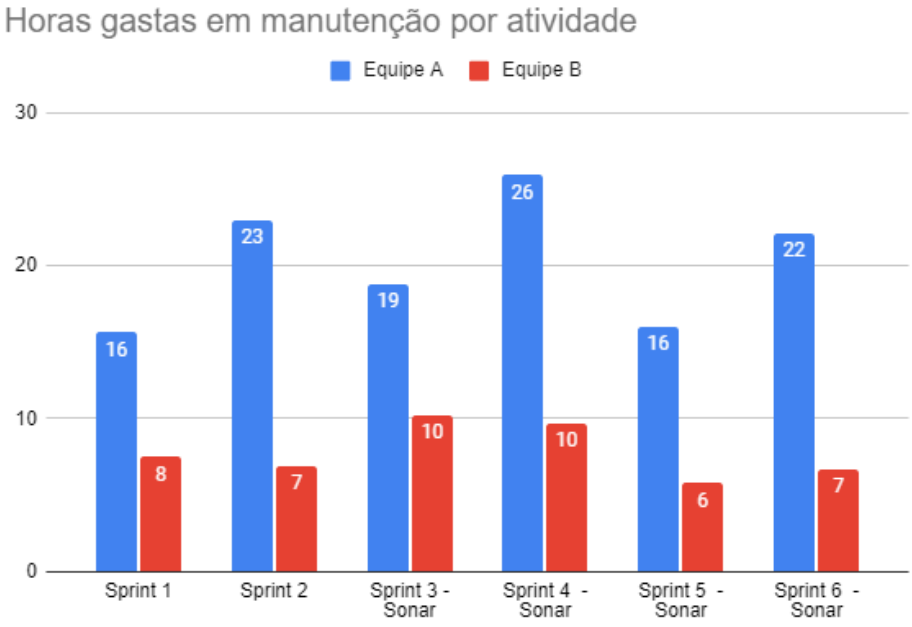


Figura 14: Tempo de manutenção por atividade

No gráfico da Figura 15 foram coletadas informações do tempo de experiência de

cada desenvolvedor dos times, sendo que na equipe A haviam 10 desenvolvedores e na equipe B apenas 4.

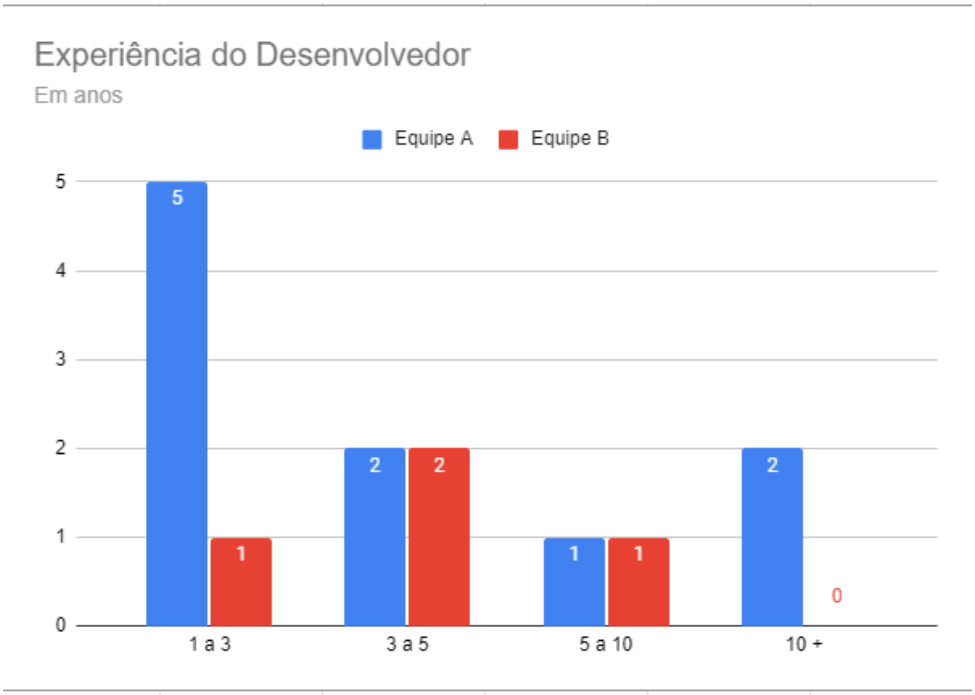


Figura 15: Tempo de experiência do desenvolvedor

Na Figura 16, é feita a comparação em horas de retrabalho identificados em cinco diferente projetos, sendo três deles executados pela equipe A e os outros dois, pela equipe B, onde podemos analisar que os 3 projetos referente a equipe A manteve-se durante as 8 semanas praticamente sem o aumento de retrabalho, já a equipe B e possível analisar uma grande redução durante todo o período.

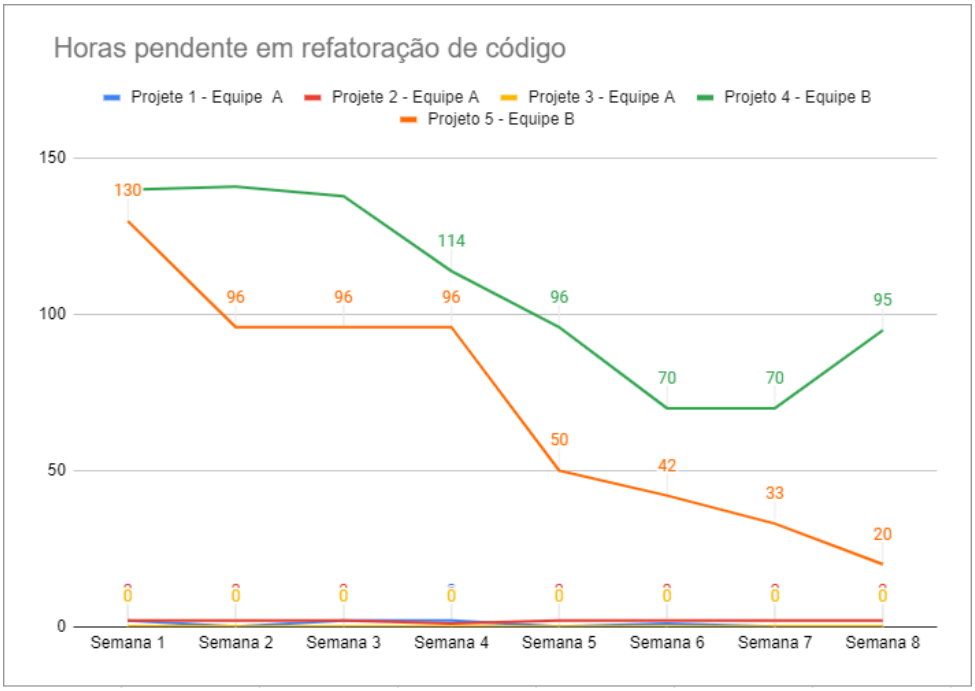


Figura 16: Retrabalho do produto

Na Figura 17 referente à Equipe A e na figura 18 sobre a Equipe B, é possível analisar a quantidade de *bugs*[1] em comparação com a quantidade de atividades, divididas em Itens de valor e *Bugs*, realizadas por *sprint* em cada time.

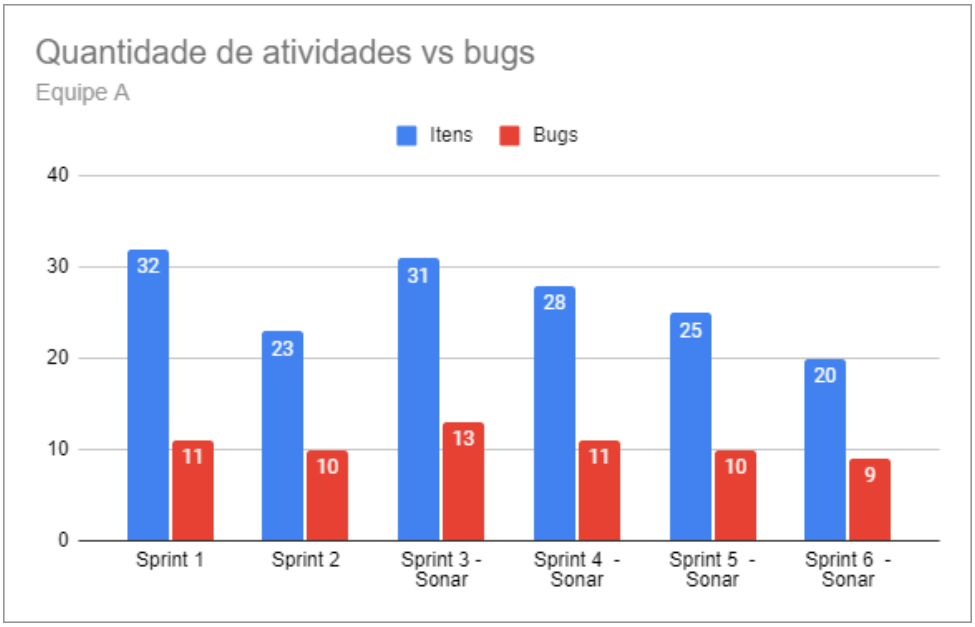


Figura 17: Quantidade de *bugs* vs Quantidade de atividades - Equipe A

[1]Bugs, no contexto deste estudo podem ser considerados inconformidades (ex: defeitos, erros e inconsistências com especificações)

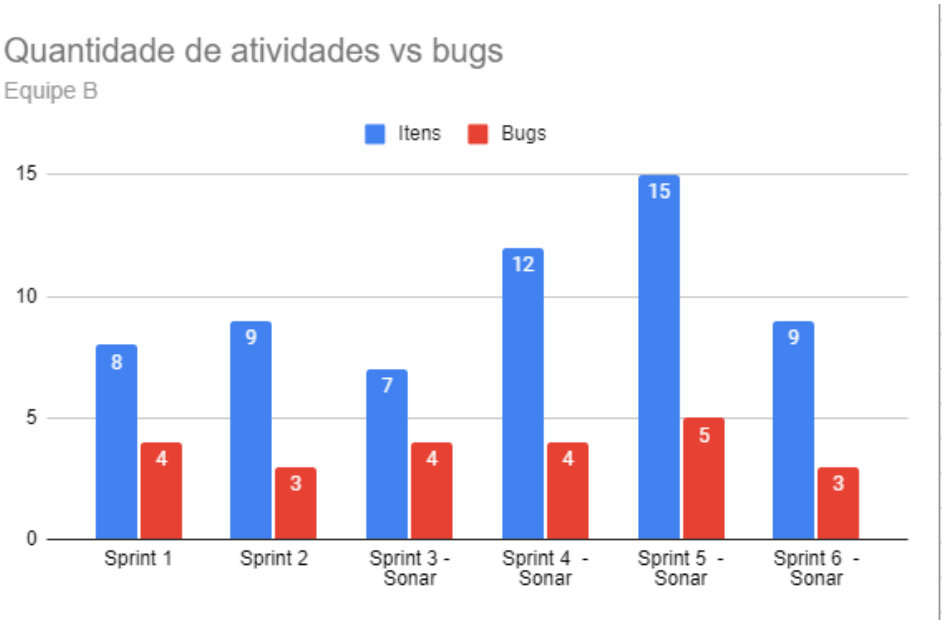


Figura 18: Quantidade de *bugs* vs Quantidade de atividades - Equipe B

Com a Figura 19 foi realizada a análise do tempo necessário para a execução da *build* em cada um dos cinco projetos, comparando o tempo de antes da aplicação do *SonarQube* e depois, onde o aumento considerável no tempo se dá a execução da análise da ferramenta a cada nova alteração, em contrapartida de todos o benefícios que a ferramenta proporciona é aceitável, mas o ideal é encontrar maneiras de reduzir o tempo do CI/CD que refere-se a todo o processo após a entrega da alteração, sendo a compilação e execução da ferramenta preditiva.

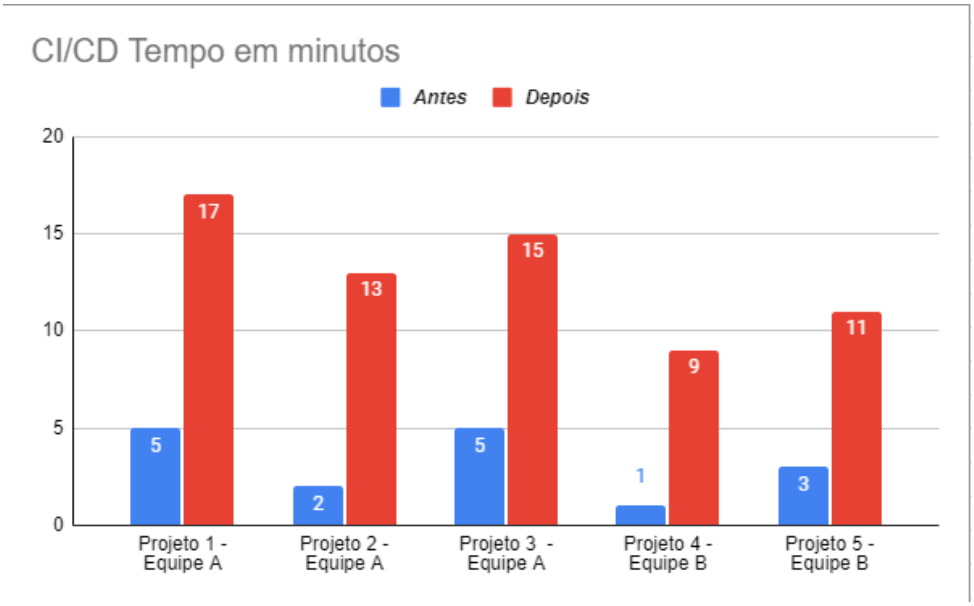


Figura 19: Tempo de *build* no CI/CD

Na Figura 20, é analisada a incidência de defeitos encontrados em produção por *Sprint* e equipe responsável, antes e durante a aplicação do *SonarQube*, importante analisar que devido a retenção das liberações de versão para os cliente e também a redução de atividades entregues houve uma queda significativa nas últimas duas *Sprints*.

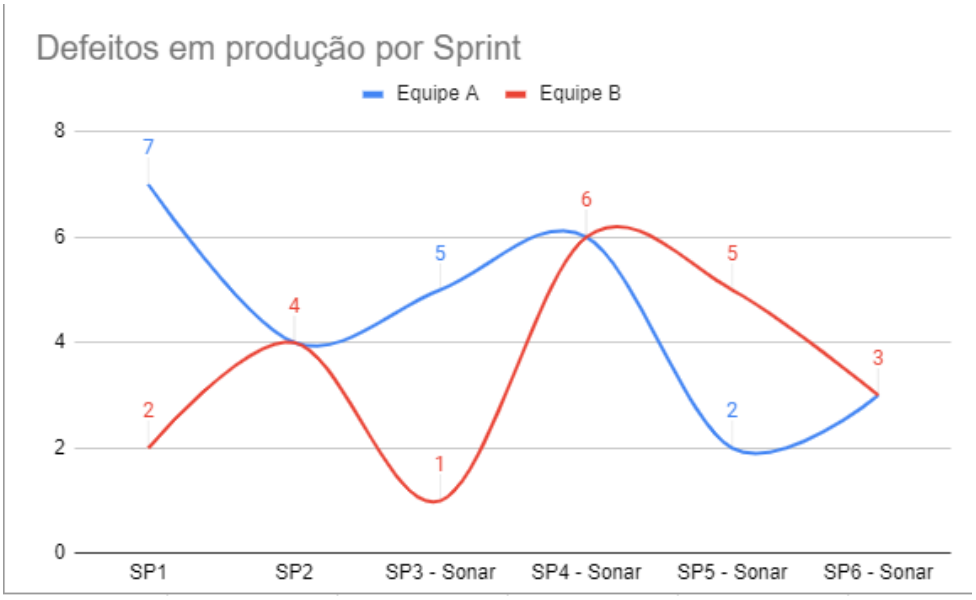


Figura 20: Quantidades de falhas encontradas em produção

5.2 QUESTÕES DE PESQUISA

Conforme definido no GQM na Figura 12, foram definidos quatro principais perguntas sobre o cenário da pesquisa como um todo e quais resultados esperados pelo mesmo, sendo positivos ou negativos, e quais seus impactos em todo o processo do desenvolvimento do software, sendo assim nos sub-tópicos abaixo serão analisados de forma mais detalhadas cada uma das questões levantadas.

5.2.1 UTILIZAÇÃO DE FERRAMENTAS PREDITIVAS DIMINUI O TEMPO GASTO EM MANUTENÇÃO?

Ao fazer uma análise diante do que foi possível responder das questões de pesquisa a partir do estudo realizado, foi analisado que devido ao curto prazo de oito semanas na aplicação da ferramenta, não foi possível observar diferenças significativas no tempo médio das atividades em ambas as equipes e projetos, conforme demostrado nas Figuras 13 e 14. Visto que é algo que necessita de constância e acompanhamento, para que seja sentida e notada a diferença significativa no tempo gasto em comparação ao esperado. Após realizada determinada alteração é ainda mais demorada para que a mesma seja testada

e liberada de fato em uma versão oficial, e mais ainda para que um grande número de clientes atualizem o sistema e de fato seja considerado como em entregue para o usuário final tais alterações.

Além do tempo necessário para que o cliente receba as alterações efetuadas, nota-se que o processo de melhora de qualidade que a ferramenta proporciona é a longo prazo, dessa forma para que seja de fato detectadas melhorias por meio das métricas, é necessário de um longo tempo aplicando boas práticas e monitorando as alterações, que não foi previsto inicialmente, para que o produto tenha melhorias perceptíveis, porém, cada pequeno ajuste que foi detectado com o auxílio da predição de defeitos é uma pequena melhoria na qualidade. Esse efeito é o mesmo que faz com que uma empresa busque um equilíbrio no tempo de experiência de seus times, conforme foi percebido pela Figura 15, pois quase sempre um profissional com mais tempo de serviços conhece melhor as práticas e estratégias para um código com maior qualidade, evitando assim futuros problemas.

Assim sendo, foi notado que em um cenário industrial existem muitos fatores que podem afetar as métricas e dificultar a análise se observado apenas os dados mensuráveis. Entre esses dados, foram levantados os seguintes pontos que ocorreram durante o estudo.

- Curto tempo para uma análise mensurável.
- Mudança de processo interno devido a certificação da LGPD (Lei Geral de Proteção de Dados Pessoais) na empresa alvo do estudo;
- Período com muita volatilidade, devido a ser o último trimestre do ano; e
- Contratações e realocações de colaboradores.

Conforme os tópicos levantados acima, o estudo empírico na empresa foi aplicado no último trimestre do ano, o qual é o mais corrido devido as metas que a empresa busca alcançar. Com isso, a empresa necessitou realocar pessoas do time para outros projetos por conta de ações contratuais e solicitações de novos clientes, além de ser um período mais crítico, onde é evitado alterações críticas para não causar problemas em clientes nessa época mais festiva do final de ano, juntamente com o período com o maior fluxo de colaboradores entrando em férias.

Tudo isso contribui para situações onde fica difícil mensurar metricamente os efeitos a curto prazo da ferramenta de predição trazendo resultado como as Figuras 17 e 18 onde em algumas *sprints* houve até mesmo o aumento da quantidade de horas gastas,

sendo necessário um maior tempo para entender como cada situação afetou os dados e não poluir os mesmos, entretanto, além de todas as situações, foi possível perceber diversas vantagens e recomendações de uso, principalmente nos retrabalho conforme demonstrado pela figura 16 houve grande benefício.

5.2.2 QUAL O ESFORÇO DA UTILIZAÇÃO E CONFIGURAÇÃO DA FERRAMENTA NO DIA A DIA?

O esforço da utilização e configuração da ferramenta no dia a dia pode ser dividido em cinco principais tópicos, sendo eles:

- Configuração inicial da ferramenta;
- Tempo de execução;
- Arquitetura para novos defeitos;
- Arquitetura pra defeitos antigos;
- Análise caso a caso.

A configuração da ferramenta trata-se tanto da instalação e manutenção do SonarQube, quanto dos ajustes específicos de cada projeto. Sua instalação na empresa usada no estudo não houve muitos contratempos, uma vez que foi utilizado o serviço Docker para subir o mesmo, onde basta configurar o mesmo em uma nova máquina que todas suas configurações já vem com template pronto. Basta ajustar o apontamento pro banco de dados e firewall na rede, fazendo com que serviço já esteja rodando em poucas horas.

Porém, após estar online é necessário configurar projeto a projeto, tanto a ferramenta, como regras de exclusão de arquivo, tipos de análise, dados a serem coletados e monitorados, como também na ferramenta de CI/CD, onde no caso usado, foi o nativo do GITLab. Foi necessário configurar um novo passo além da construção do projeto, apontando o mesmo para a ferramenta e disponibilizando o acesso de leitura para a execução, sem contar de que o tempo necessário para cada projeto pode variar, tendo em conta a possibilidade de se esbarrar com alguma situação atípica, a qual necessidade ser tratada caso a caso, melhorando sua configuração.

Um ponto que encontrou-se um alto impacto, foi o tempo de execução do processo de CI/CD, onde conforme o levantamento do tempo gasto para concluir a pipeline aumentou

mais que 100 (cem) porcento, conforme mostra o gráfico da figura 19, o que se dá ao fato de que para uma melhor análise da ferramenta, é necessário rodar o escâner completo no projeto a cada nova implementação para se obter um melhor resultado. Para contornar a situação de tempo de compilação, foram levantadas duas soluções para não afetar o time, sendo elas, rodar a análise assim que é aberta a solicitação de integração ou após a construção e a geração do arquivo de instalação, onde ambas, por mais que demore sua execução, o fluxo da alteração não é impactado.

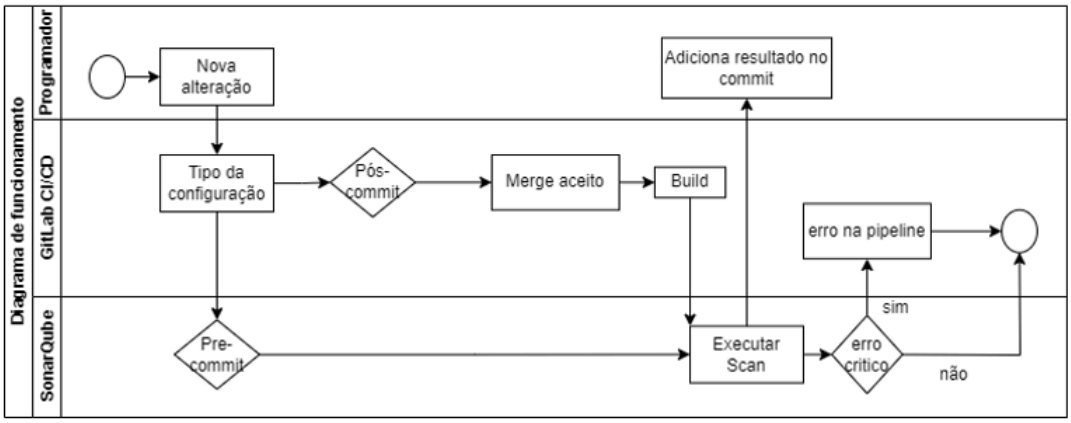
Conforme Figura 11, para cada nova solicitação, é realizada a revisão na ferramenta de quais sugestões de refatorações e situações que envolvem qualidade que realmente necessitam ser corrigidos e quais locais que a nova solicitação irá impactar. Para isso é feita uma análise desses retrabalhos, que por meio dessa ação trouxe grande redução na quantidade de horas pendentes para refatoração, conforme Figura 16, para que seja possível aproveitar a oportunidade encontrada para alterar trechos do código e já resolver os problemas reportados pela ferramenta, para que assim não seja necessário alocar mais tempo para rever situações antigas de refatorações que foram detectadas pelo *SonarQube*.

Além disso, após uma nova solicitação de alteração, caso seja detectado que a mesma possui defeitos que impactam na qualidade do produto, é realizada a notificação para quem abriu a solicitação e para quem será responsável por aprovar a mesma ou não, além de notificar os que já estão configurados para sempre receber tais avisos, aumentando a quantidade de *bugs* conforme Figuras 17 e 18 e reduzindo-os em produção, para que assim seja realizada a avaliação se irá ser mesclada ou se precisa realizar as correções apontadas. Porém, pelo curto prazo não é possível afirmar se houve melhora nesse aspecto. Por mais que o ideal seja sempre ajustar todos os problemas relatados, no dia a dia, muitas situações necessitam que se aceite o risco devido ao cronograma apertado além de que dependendo da configuração do projeto, nem sempre o que foi detectado é de fato um problema que necessita ser ajustado, por isso, fica a critério do responsável por analisar se deve prosseguir dessa forma.

5.2.3 QUAL A MELHOR ESTRATÉGIA DE UTILIZAÇÃO DA PREDIÇÃO, PÓS-COMMIT OU PRÉ-COMMIT?

Conforme planejado foi realizada a implementação de duas formas para entender seus impactos e influência ao time, sendo eles o modelo de aplicar no *pré-commit* e no *pós-commit*, conforme demostrado na Figura 21. A primeira consiste em que, ao ser aberta uma nova solicitação de mesclagem para o projeto principal com base no *fork* do desenvolver,

será rodado o escâner da ferramenta de predição para que, antes mesmo da alteração ser aprovada, já possuir o *feedback* se foi encontrado algum *defeito* na qualidade do código, podendo ser códigos fora do padrão, blocos duplicados, código mal estruturado, entre outros preditos. Já no modelo de *pós-commit* é executado o escâner apenas após o aceite da solicitação, sendo notificado caso encontre alguma das situações citadas anteriormente.



Fonte: Pelepenko, Lucas, 2021

Figura 21: Fluxo de entrega Pré-commit, Pós-commit

Conforme especificado na Figura 21, caso seja *pré-commit*, após ser concluído a execução do escâner é adicionado o resultado no commit, exibindo o mesmo na solicitação do merge. Dessa forma, o responsável por aceitar o merge pode verificar se acusou algum erro, e caso tenha, pode solicitar ao programador para que ajuste a alteração da sugestão antes do mesmo ser aceito e entregue no repositório. Uma vez aceito, todos os desenvolvedores podem receber tais alterações. Porém, no modelo *pós-commit*, mesmo que seja encontrada uma situação problemática, o mesmo já foi aceito e pode causar problemas aos demais membros da equipe.

Entretanto, por mais atrativo que seja utilizar o modelo de *pré-commit*, pode ocasionar alguns problemas no fluxo do time, uma vez que é necessário aguardar a execução da ferramenta antes de poder ou não prosseguir com o aceite do *merge*. Dessa forma, em situações críticas e, principalmente no final do expediente, caso tenha cliente esperando alguma alteração urgente para o dia seguinte, pode gerar inconveniências, considerando a demora relativa para sua execução, conforme demonstrado na Figura 19.

Os dados da figuras 17, 18, 19 e 20 são os que mais se relacionam com a questão levantada. Contudo, ao analisar tais informações, não foi possível extrair dados quantitativos sobre as diferenças causadas em curto prazo no uso das técnicas Pré-commit e Pós-commit, considerando também as mudanças ocorridas no time e no processo, devido

a volatilidade comumente encontrada nas fábricas de software, com exceção da figura 19 onde teve grande aumento no tempo gasto para finalizar a pipeline de entrega de uma nova alteração, podendo assim, incomodar alguns do time que estavam esperando finalizar rapidamente como era antes.

De acordo com o *feedback* retornado pelo Time A, foi optada a utilização do modelo *pós-commit*, evitando-se que em situações críticas o desempenho não fosse afetado. Com o tempo há ainda a possibilidade de que seja alterado o modelo usado, devido a adoção e entendimento da nova cultura, mas é algo que necessita de tempo e acompanhamento.

Já o Time B, possui um fluxo de entrega sobre demandas, foi definido o uso do modelo *pré-commit*, considerando que a equipe não aloca toda sua carga produtiva para desenvolvimento de novas alterações, e que dessa forma torna-se mais flexível a mudanças no fluxo e entregas, pois seu principal cliente é a própria empresa por meio de solicitações internas. Com isso, facilita-se a alteração do processo sem afetar negativamente o desempenho e quase sempre trazendo resultados de valor, pois devido ao escopo reduzido da equipe, muitas das entregas demoram a ser testadas, e nesse cenário, uma análise prévia feita pela ferramenta preditora traz resultados mais rapidamente, facilitando a correção e não consumindo muito tempo do testador.

5.2.4 QUAL O EFEITO DA APLICAÇÃO DE ESTRATÉGIA DE PREDIÇÃO DE DEFEITOS NOS MEMBROS DA EQUIPE DE DESENVOLVIMENTO DE SOFTWARE?

Ao decorrer da pesquisa e conforme a utilização da ferramenta nos times, foram percebidos alguns dos benefícios a curto, médio e longo prazo, onde cada um impacta de uma forma diferente no dia a dia da equipe, do software e do cliente que irão utilizar o mesmo, e podem se resumir em quatro itens, sendo eles:

- Boas práticas;
- Propor melhores soluções;
- Aprendizado futuro; e
- Cuidado extra.

Uma das vantagens percebidas a curto prazo é a utilização das boas práticas de desenvolvimento, onde caso seja enviada uma alteração que não se enquadre nos padrões e recomendações, a ferramenta detecta imediatamente e adiciona avisos de qual a forma

ideal e em partes do código é necessária a modificação, conforme demostrado na figura 22 abaixo. além de afetar diretamente a refatoração confirme figura 16 evitando seu aumento. Além do aumento na quantidade de *bugs* encontrado internamente conforme figura 17 e 18. Dessa forma, a ferramenta auxilia o desenvolvedor a aprender e também detectar quando foi enviado algo que pode prejudicar a qualidade do produto, para que com o tempo evite-se os erros mais comuns.

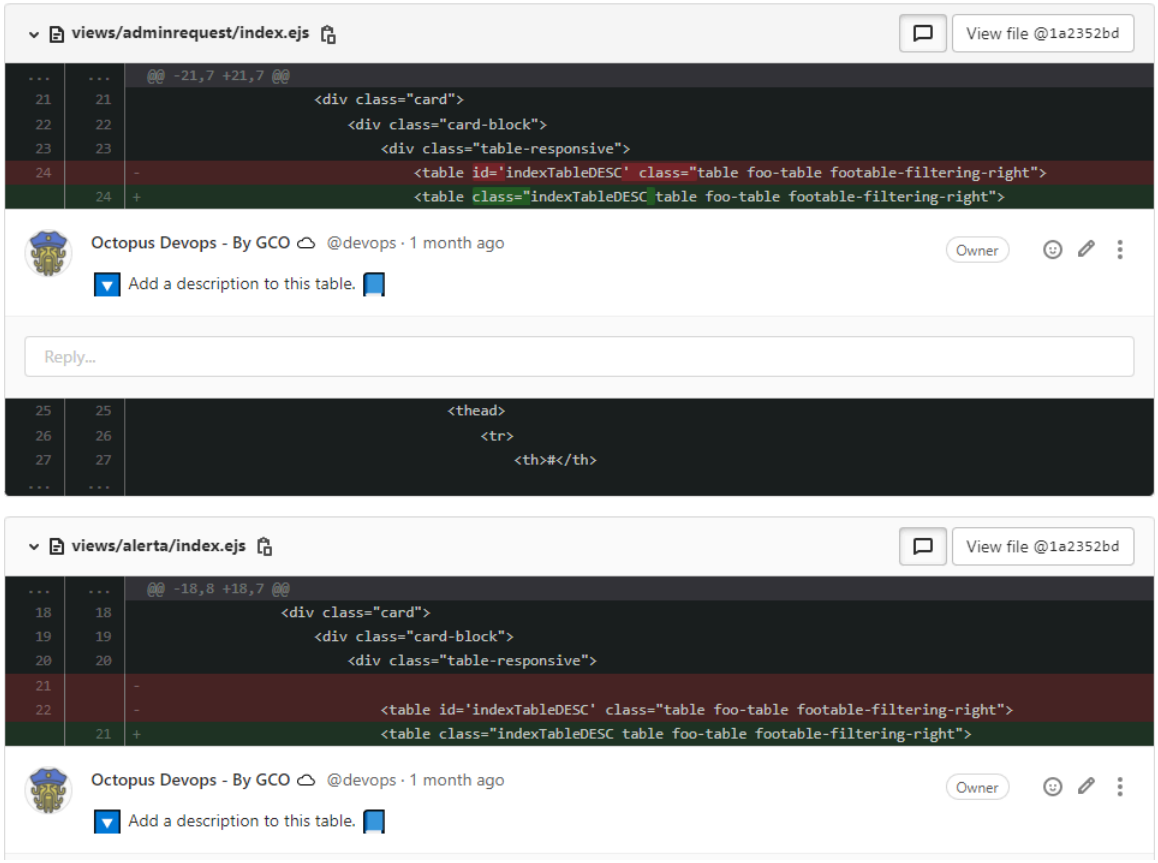


Figura 22: Notificação de boas práticas em html

Fonte: Pelepenko, Lucas, 2021

Outra vantagem notada, é a proposta de melhores soluções, tanto para o desenvolvedor, que com o tempo já entende onde e porque fazer determinada tarefa para evitar que a ferramenta detecte e notifique o ajuste, como também para o arquiteto, que pode analisar os dados coletados no projeto, como complexidade e linhas duplicadas e propor mudanças que possam vir a auxiliar a redução de Retrabalho de código e funções mal estruturadas, e aproveitando assim as novas demandas para que seja feita a refatoração de pequenas partes do sistema que a médio e longo prazo influenciará positivamente na qualidade do sistema como um todo.

Dessa forma, conforme o time utiliza os dados gerados, vai-se tendo um novo

entendimento do porquê cada situação ocasionar um defeito ou reduzir a qualidade do código. Com o tempo todos acabam aprendendo melhores práticas e formas de desenvolver corretamente pelo fato da própria ferramenta detectar algum problema e já fornecer soluções de artigos de qual a maneira correta, indicando o que o código pode gerar caso não seja corrigido, sendo falha de segurança ou defeito no sistema, conforme demonstrado na Figura 23. Onde uma vez que o programador já passou e corrigiu determinado problema, em próximas alterações terá um cuidado redobrado ao realizar determinada alteração porque já sabe a forma correta de fazê-la.

Porém o principal efeito negativo percebido e o aumento drástico do tempo para conclusão do executável conforme visto pela Figura 19, uma vez que sempre são buscados meios para diminuir o tempo, agilizando assim a execução dos roteiros automatizados e organização do time. Contudo, depende da forma que o time realiza o processo de entrega pode impactar de forma diferente, como por exemplo a equipe A não apoiou o uso do pré-*commit* conforme descrito nos tópicos acima, já a equipe B por ser menor e trabalhar com outro processo, o tempo de *build* no dia a dia não atrapalha sua produtividade sendo aceitável o aumento do mesmo e podendo ser executado antes de ser aceita a alteração.

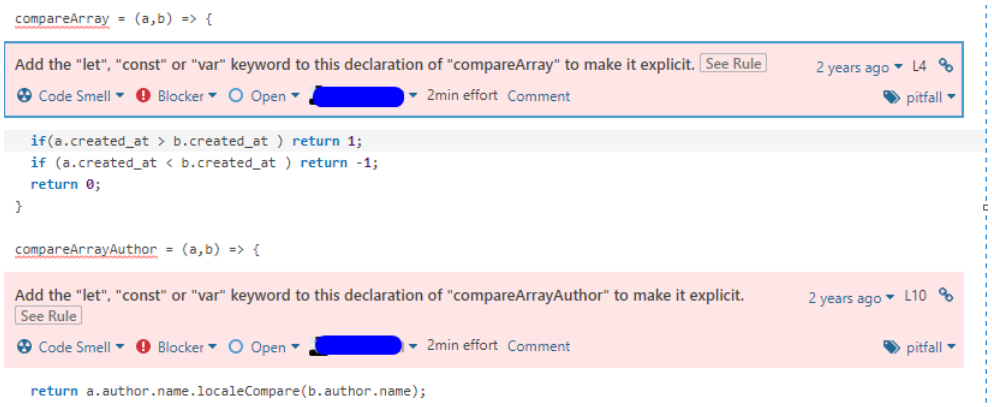


Figura 23: Sugestão de como resolver o problema detectado
Fonte: Pelepenko, Lucas, 2021

5.3 MELHORIA DA QUALIDADE DO CÓDIGO E DO PRODUTO

Conforme descrito anteriormente neste documento, quanto maior a qualidade do código, menor será a incidências de novas falhas e menor o custo de manutenção e melhorias futuras. Em virtude disso, foram criadas atividades focadas na melhoria do código nos pontos mais criticos apontados pelo *SonarQube*, como por exemplo na Figura 24, onde pode ser vista algumas áreas onde o retrabalho está alto, causando dificuldade no

desenvolvimento de novas alterações, gerando um efeito de bola de neve, onde cada vez fica pior.

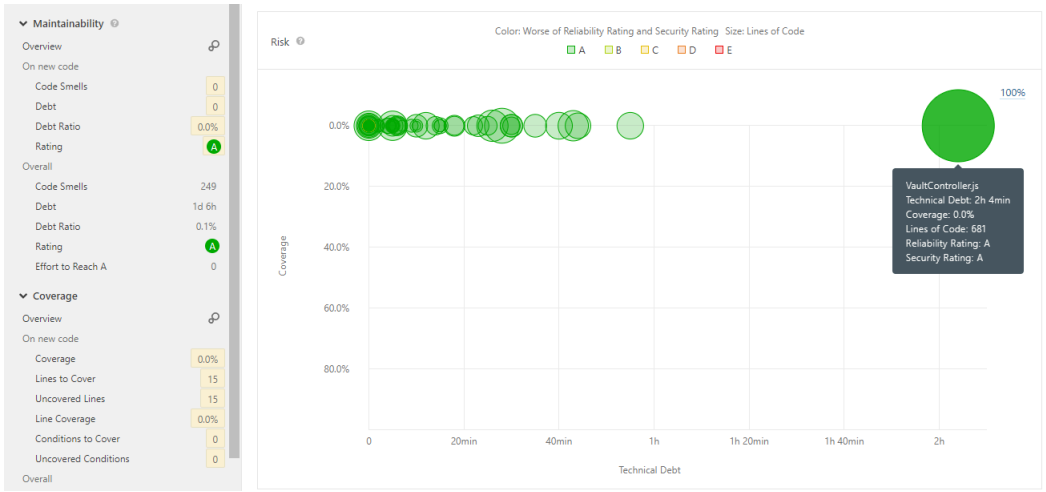


Figura 24: SonarQube Retrabalho por local

Fonte: Pelepenko, Lucas, 2021

Os projetos monitorados nessa pesquisa, referente a Equipe A, tiveram poucos débitos detectados pela ferramenta e conseguiram controlar para que não aumentem, uma vez que cada novo problema detectado, em suma, a maioria foi corrigido logo na sequência, mantendo assim, poucas horas de débito no total do projeto, conforme visto na Figura 16, e na mesma imagem, podemos analisar que a Equipe B havia muitas horas de débitos pendentes nos projetos de sua autoria, os quais foram realizadas algumas melhorias ao decorrer das semanas.

Porém, no dia a dia, é muito difícil alocar melhorias focadas no aumento da qualidade do código, onde as que foram realizadas durante essa pesquisa foi apenas uma coincidência, pois o módulo principal de API's estava sendo refatorado e centralizado, sendo possível assim a refatorar conforme demostrado no projeto 4 da figura 16. Contudo, foi percebido que é mais fácil evitar que novas refatorações sejam introduzidas no código durante o dia a dia, do que corrigir os antigos, dessa forma, foram realizadas duas ações principais que buscam melhorar o código, sendo elas as seguintes:

- Evitar novos débitos: Com essa ação, evita-se que sejam introduzidos novos débitos para que não acumule mais alterações, e que cada débito cause outros novos débitos, o que provou ser viável e não afeta muito as equipes durante o desenvolvimento.
- Corrigir sempre que tiver oportunidade: Com essa ação sempre que tiver alguma demanda nova de manutenção ou melhoria em uma parte do código que possui

débitos já mapeados, pode ser feita uma revisão pelo próprio programador para melhorar a mesma, uma vez que já será testado o local da alteração, então mesmo que refatorado, são poucas as chances de originar novos *bugs*.

Seguindo essas duas regras citadas acima, é possível evitar que seja introduzido novos débito e também ser corrigido os antigos de forma gradual, sem afetar a quantidade de entregas e desempenho da equipe, mas por ser feito pouco a pouco, o seu efeito só poderá ser medido a longo prazo ou em projetos pequenos.

5.4 AMEAÇAS À VALIDADE

As ameaças à validade do estudo podem ser analisadas sob os seguintes aspectos:

- *Ameaças à validade de conclusão*: que se refere ao quão confidente o estudo é em relação ao uso de técnicas/ferramentas de predição de falhas resultaram nas métricas. Em relação a esse aspecto considera-se que as ameaças são *baixas*, haja vista que as correções são realizadas no dia a dia, conforme é detectada pela ferramenta.

- *Ameaças à validade de construção*: que foca na relação entre as hipóteses e teorias por trás de se utilizar ferramentas de predição de defeitos e as métricas coletadas. De modo similar ao item anterior, pode-se afirmar que a ameaça nesse sentido é *média* devido ao aculturamento de uso de estratégia de predição de falhas ser novo para times e ao curto prazo disponível;

- *Ameaças à validade externa*: tal análise é associada à generalização dos resultados obtidos no estudo. Devido ao fato de ter sido realizado em apenas dois times e por um número limitado de *sprints*, esses autores julgam essas ameaças como sendo *altas*. Estudos complementares são necessários para generalizar os resultados obtidos. Adicionalmente, estudos com variados escopos de projetos, times com expertises diferentes e variados tamanhos de times são alguns dos elementos a serem considerados; e

- *Ameaças à validade interna*: em relação às ameaças de validades internas, que se referem à confidência associada às métricas coletadas, destaca-se que o nível de ameaça identificado é *médio*, haja vista que a aplicação da ferramenta e a análise dos dados foi realizado pela mesma pessoa, houve parcialidade na avaliação dos dados.

6 CONSIDERAÇÕES FINAIS

Atividades de predição de defeitos têm sido negligenciadas na indústria por conta da escassez de recomendações e estudos teórico-práticos sobre a aplicabilidade do tema em projetos contemporâneos, porém cada vez mais os clientes exigem que o software funcione da forma correta e que as novas melhorias sejam entregues o mais rápido possível, o que na maioria das vezes são duas ações que não funcionam juntas. Dessa forma, faz-se necessário aplicar ferramentas que não afetem de forma considerável o tempo investido para desenvolver novas alterações e que tragam melhoras na qualidade do código evitando a inserção de novos problemas.

Um dos pontos mais importantes para poder aplicar ferramentas de predição de defeito e de qualidade de código como o SonarQube é ter-se um alto nível de maturidade de processos e de gerenciamento. Tais níveis são necessários para que os times tenham a aplicabilidade correta, a qual demanda de gastos extras para configurar a ferramenta e utilizá-la no dia a dia, por meio de correções detectadas durante a entrega do código, para assim ter-se um uso adequado que irá refletir em menores quantidades de defeitos e facilitar novas alterações a longo prazo.

Como cada processo alterado pode causar reações negativas na equipe, um uso de implantação mais inteligente é a sua aplicação em poucas equipes, realizando pequenas mudanças no processo de forma gradual. Por isso, a principal preocupação durante toda a implantação do SonarQube nos times desse trabalho, foi como implantar sem afetar significativamente o processo já utilizado, considerando que se houvessem mudança críticas, não seriam nem aprovadas para implantação e estudo nos times, pois, aplicaram-se as ferramentas para cada equipe respeitando as singularidades dos processo, não apenas para comparar as duas técnicas, mas também para evitar atrapalhar o dia a dia da equipe, dessa forma a ferramenta executa e da o feedback de forma automática, sem a necessidade de treinamentos auxiliares, exceto pelo gerente e líder que possuíam acesso aos dados gereis e métricas.

Uma vez que se deseje implantar ferramentas preditivas, deve-se entender que um cenário real de desenvolvimento está sempre sujeito a grandes mudanças, como durante a aplicação do estudo, onde houve demissões e contratações nos times, ações da direção da empresa para o alcance de metas, onde houve a necessidade de liberar códigos que não passaram na validação da ferramenta devido ao curto prazo de entrega, ignorando pequenos problemas de qualidade, e com isso, afetou-se também o *backlog* estipulado para correções e refatorações de *bugs* antigos encontrados pelo *SonarQube* que acabaram indo para o próximo trimestre.

Porém com o curto tempo de aplicação do estudo desse trabalho, não foi possível visualizar nas métricas o resultado dos esforços investidos pelas equipes, uma vez que muitas das alterações realizadas devido ao processo longo, que acabou sendo liberado para clientes após poucas semanas, não sendo possível analisar os dados da quantidade de falhas encontradas após a aplicação da ferramenta e suas causas. Por fim, foi identificada uma grande melhora na qualidade de código no quesito de refatorações no decorrer do período do estudo, apesar do tempo reduzido.

O modelo GQM foi essencial para o bom andamento do estudo, utilizando como base principal as quatro questões de pesquisa estipuladas com o intuito de responder as duas principais metas desse estudo. Não foi possível notar uma redução na quantidade de defeitos encontrados em produção, em virtude do curto espaço de tempo. Em contrapartida, notou-se uma melhora significativa na qualidade do código conforme indicou os dados coletados, notando-se que houve uma grande melhora refente aos retrabalhos do código fonte, devendo-se principalmente ao fato de que as alterações realizadas possuíram um rápido *feedback*, enquadrando-se no escopo de tempo da pesquisa.

Assim sendo, o presente trabalho contribui de forma expressiva para as equipes que foram aplicadas, mostrando diversos pontos chaves a se atentar, além de mostrar que é possível aplicar ferramentas preditivas sem afetar demais o fluxo já existente e obter resultados mesmo em um ambiente volátil e cheio de mudanças. Com base em todos os dados analisados, em um próximo estudo, recomenda-se aplicar a ferramenta por maiores períodos de tempo, como pelo menos oito ciclos de desenvolvimento e entrega pro usuário final, pois mesmo após o fim do presente trabalho a empresa em questão irá continuar a utilizar a ferramenta e os dados do mesmo podem vir a serem utilizados em projetos futuros.

REFERÊNCIAS

12200, I. Iso 12200:1999. IEEE, 1999.

ADKAR, S. S. P. A modern review on scrum: Advance project management method. **International Journal of Trend in Scientific Research and Development**, v. 2, n. 4, p. 87–95, june 2018. ISSN 2456-6470. Disponível em: <http://www.ijtsrd.com/computer-science/other/12864/a-modern-review-on-scrum-advance-project-n

BARLOW, H. **Unsupervised Learning**. [S.l.]: Massachusetts Institute of Technology, 2008.

BASILI, V. R.; CALDIERA, G.; ROMBACH, H. D. The goal question metric approach. In: **Encyclopedia of Software Engineering**. [S.l.]: Wiley, 1994.

BECK, K. et al. **Manifesto for Agile Software Development**. 2001. Disponível em: <http://agilemanifesto.org/iso/en/manifesto.html>.

BEIZER, B. **Black-Box Testing: Techniques for Functional Testing of Software and Systems**. 1. ed. [S.l.]: Verlag John Wiley Sons, Inc, 1995. ISBN 0471120944.

BERKI, E.; GEORGIADOU, E.; HOLCOMBE, M. Requirements engineering and process modelling in software quality management— towards a generic process metamodel. **Software Quality Journal**, v. 12, n. 3, p. 265–283, sep 2004. Disponível em: <http://dx.doi.org/10.1023/B:SQJO.0000034711.87241.f0>.

BERNARDO, F. K. P. C. A importância dos testes automatizados. 2008.

BERTOLINO, A. Software testing research: Achievements, challenges, dreams. In: **2007 Future of Software Engineering**. Washington, DC, USA: IEEE Computer Society, 2007. (FOSE '07), p. 85–103. ISBN 0-7695-2829-5. Disponível em: <http://dx.doi.org/10.1109/FOSE.2007.25>.

BINDER, R. V. Testing object-oriented software: A survey. **Softw. Test., Verif. Reliab.**, v. 6, n. 3/4, p. 125–252, 1996.

BOBROV, E. et al. Teaching devops in academia and industry: Reflections and vision. In: BRUEL, J.-M.; MAZZARA, M.; MEYER, B. (Ed.). **DEVOPS**. Springer, 2019. (Lecture Notes in Computer Science, v. 12055), p. 1–14. ISBN 978-3-030-39306-9. Disponível em: <http://dblp.uni-trier.de/db/conf/devops-ws/devops-ws2019.htmlBobrovBCGMM19>.

BOBROVSKIS, S.; JURENOKS, A. A survey of continuous integration, continuous delivery and continuos deployment. In: ZDRAVKOVIC, J. et al. (Ed.). **BIR Workshops**. CEUR-WS.org, 2018. (CEUR Workshop Proceedings, v. 2218), p. 314–322. Disponível em: <http://dblp.uni-trier.de/db/conf/bir/bir2018w.htmlBobrovskisJ18>.

BOURQUE, R. E. F. P. **Guide to the Software Engineering Body of Knowledge (SWEBOK(R)): Version 3.0**. 3rd. ed. Washington, DC, USA: IEEE Computer Society Press, 2014. ISBN 0769551661.

BROY, M. Yesterday, today, and tomorrow: 50 years of software engineering. IEEE, 2018.

CAMPBELL, P. P. P. G. A. **SonarQube in Action**. [S.l.]: Manning Publications Co, 2013.

CAPIZZI, A. et al. Anomaly detection in devops toolchain. In: BRUEL, J.-M.; MAZZARA, M.; MEYER, B. (Ed.). **DEVOPS**. Springer, 2019. (Lecture Notes in Computer Science, v. 12055), p. 37–51. ISBN 978-3-030-39306-9. Disponível em: <http://dblp.uni-trier.de/db/conf/devops-ws/devops-ws2019.htmlCapizziDAMAB19>.

CARNIELLO, A.; JINO, M.; CHAIM, M. L. Structural testing with use cases. **Journal of Computer Science Technology**, 2005.

CATAL, C.; DIRI, B. A systematic review of software fault prediction studies. **Expert Systems with Applications**, v. 36, n. 4, p. 7346 – 7354, 2009. ISSN 0957-4174. Disponível em: <http://www.sciencedirect.com/science/article/pii/S0957417408007215>.

CAVEZZA ROBERTO PIETRANTUONO, S. R. D. G. Performance of defect prediction in rapidly evolving software. São Carlos, 2015.

CHAKRABORTY, U. D.; SUR, B. Random forest based fault classification technique for active power system networks. IEEE, Bangalore, India,, 2019.

CHEN, B. Improving the software logging practices in devops. IEEE, Montreal, Canada, 2019.

CHIU YUH-RU YU, H. L. L. L. C.-H. M.-H. The use of facial micro-expression state and tree-forest model for predicting conceptual-conflict based conceptual change. ESERA, Taipei, Taiwan, 2016.

CONRADI, R.; WESTFECHTEL, B. Version models for software configuration management. **ACM Comput. Surv.**, ACM, New York, NY, USA, v. 30, p. 232–282, June 1998. ISSN 0360-0300. Disponível em: <http://doi.acm.org/10.1145/280277.280280>.

CRUZ, A. B. A. V. L. A devops introduction process for legacy systems. IEEE, São Paulo, Brazil, 2018.

DELAMARO J. C. MALDONADO, M. J. M. E. **INTRODUÇÃO AO TESTE DE SOFTWARE**. 2. ed. São Paulo: Elsevier, 2016.

EBERT, A. H. C. Requirements engineering – industry needs. IEEE, Catalunya, Spain, 2008.

EBERT, C. et al. DevOps. **IEEE Software**, Institute of Electrical and Electronics Engineers (IEEE), v. 33, n. 3, p. 94–100, maio 2016. Disponível em: <https://doi.org/10.1109%2Fms.2016.68>.

FENTON, M. N. N. A critique of software defect prediction models. IEEE, 1999.

GAVANKAR, S. D. S. S. S. Eager decision tree. IEEE, Mumbai, India, 2017.

GIGER, E. et al. Method-level bug prediction. In: RUNESON, P. et al. (Ed.). **ESEM**. ACM, 2012. p. 171–180. ISBN 978-1-4503-1056-7. Disponível em: <http://dblp.uni-trier.de/db/conf/esem/esem2012.htmlGigerDPG12>.

GUSTAFSSON, T. Machine learning and sonarqube kpis to predict increasing bug resolution times. Finland, 2019.

HARAHAP, E. F.; EKADIANSYAH, R.; SARI, A. Implementation of naïve bayes classification method for predicting purchase. IEEE, Parapat, Indonesia,, 2018.

HASSAN, A. E. **Predicting Faults Using the Complexity of Code Changes**. Vancouver, Canada: IEEE, 2009.

HEARST S.T. DUMAIS, E. O. J. P. B. S. M. Support vector machines. IEEE, 1998.

HUDSON, A. **Program errors as a birth and death proces**. Santa Monica, CA, EUA: Technical Report SP-3011, 1967.

IEEE. **IEEE standard glossary of software engineering terminology**. New York, USA: IEEE, 1990.

JI, J. Y. S. Generalized linear discriminant analysis: A unified framework and efficient model selection. IEEE, 2008.

KAMEI, E. S. Y. Defect prediction: Accomplishments and future challenges. IEEE, 2016.

KENEFICK, S. **Real World Software Configuration Management (Expert's Voice)**. Direct and 1-si. Springer, 2008. ISBN 9781590590652. Disponível em: <http://www.amazon.de/World-Software-Configuration-Management-Experts/dp/1590590651>.

KEYES, J. **Software Configuration Management**. 1. ed. Taylor Francis, 2007. Disponível em: <http://www.amazon.de/Software-Configuration-Management/dp/B000Q36ELA/ref=

KUMAR, S. S. R. S. **Software Fault Prediction A Road Map**. Roorkee, India: Springer, 2018.

LANGE, C.; CHAUDRON, M. Managing Model Quality in UML-Based Software Development. In: **Software Technology and Engineering Practice, 2005. 13th IEEE International Workshop on**. [S.l.: s.n.], 2005. p. 7–16.

LOELIGER, J. **Version Control with Git - Powerful Tools and Techniques for Collaborative Software Development**. Sebastopol: O'Reilly Media, Inc., 2009. Disponível em: <https://www.amazon.de/Version-Control-Git-Jon-Loeliger/dp/0596520123/ref=sr_1_cc_1?s=aps&ie=

MALDONADO, e. Introduçao ao teste de software. 2004.

MAXIMINI, D. V. text. **The Scrum Culture. Introducing Agile Methods in Organizations**. Cham: Springer International Publishing, 2015. Buch. (Management for Professionals). Vetrieb: Ann Arbor, Michigan: ProQuest ; Materialart: Computer file ; Andere Veröffentlichungsform: ISBN 9783319118260 ; Inhaltstyp: Text ;

Medientyp: Computermedien ; Datenträgertyp: Online-Ressource ; Sprache: Englisch ; DDC-Sachgruppe: 650 ; Description based on publisher supplied metadata and other sources ; Quelldatenbank: HEBIS ; Format:marcform: print ; Umfang: 1 Online-Ressource (315 pages). ISBN 978-3-319-11827-7.

MENZIES, Z. T. et al. Defect prediction from static code features: current results, limitations, new approaches. **Automated Software Engineering**, v. 17, n. 4, p. 375 – 407, 2010.

MUNIZ, F. **DevOps – o que é e quais são seus benefícios?** 2018. Disponível em: <https://blog.4linux.com.br/beneficios-do-devops/>.

MURPHY, G. C.; KERSTEN, M. Towards bridging the value gap in devops. In: BRUEL, J.-M.; MAZZARA, M.; MEYER, B. (Ed.). **DEVOPS**. Springer, 2019. (Lecture Notes in Computer Science, v. 12055), p. 181–190. ISBN 978-3-030-39306-9. Disponível em: <http://dblp.uni-trier.de/db/conf/devops-ws/devops-ws2019.htmlMurphyK19>.

NAM, J. Survey on software defect prediction. 2014.

ORTEGA, M.; PÉREZ, M.; ROJAS, T. Construction of a systemic quality model for evaluating a software product. **Software Quality Control**, v. 11, n. 3, p. 219–242, 2003. ISSN 0963-9314.

PAVLYSHENKO, B. Machine learning, linear and bayesian models for logistic regression in failure detection problems. IEEE, Washington, USA, 2016.

PIKKARAINEN, M. et al. The impact of agile practices on communication in software development. **Empirical Software Engineering**, 2008. Disponível em: <http://dx.doi.org/10.1007/s10664-008-9065-9>.

PRESSMAN, B. R. M. R. S. **Engenharia de Software uma abordagem profissional**. 8. ed. New York: McGraw-Hill, 2016.

PRESSMAN, R.; MAXIM, B. **Engenharia de Software-8ª Edição**. [S.l.]: McGraw Hill Brasil, 2016.

PRESSMAN, R. S. **Engenharia de Software**. 6. ed. Rio de Janeiro: McGraw-Hill, 2002.

PRESSMAN, R. S. **Engenharia de Software**. 6. ed. New York: Mc-Graw Hill, 2005.

Punitha, K.; Chitra, S. Software defect prediction using software metrics - a survey. In: **2013 International Conference on Information Communication and Embedded Systems (ICICES)**. [S.l.: s.n.], 2013. p. 555–558.

PUNITHA, S. C. K. Software defect prediction using software metrics - a survey. IEEE, Chennai, India, 2013.

RADJENOVIć, D. et al. Software fault prediction metrics: A systematic literature review. **Information and Software Technology**, v. 55, n. 8, p. 1397 – 1418, 2013. ISSN 0950-5849. Disponível em: <http://www.sciencedirect.com/science/article/pii/S0950584913000426>.

ROSEN BEN GRAWI, E. S. C. Commit guru: Analytics and risk prediction of software commits. ESEC/FSE, NY, USA, 2015.

ROSSO, S. P. D.; JACKSON, D. What's wrong with git?: A conceptual design analysis. In: **Proceedings of the 2013 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming & Software**. New York, NY, USA: ACM, 2013. (Onward! '13), p. 37–52. ISBN 978-1-4503-2472-4. Disponível em: <http://doi.acm.org/10.1145/2509578.2509584>.

RUBART, J.; FREYKAMP, F. Supporting daily scrum meetings with change structure. In: **HT '09: Proceedings of the Twentieth ACM Conference on Hypertext and Hypermedia**. New York, NY, USA: ACM, 2009.

SANCTIS, M. D.; BUCCHIARONE, A.; TRUBIANI, C. A devops perspective for qos-aware adaptive applications. In: BRUEL, J.-M.; MAZZARA, M.; MEYER, B. (Ed.). **DEVOPS**. Springer, 2019. (Lecture Notes in Computer Science, v. 12055), p. 95–111. ISBN 978-3-030-39306-9. Disponível em: <http://dblp.uni-trier.de/db/conf/devops-ws/devops-ws2019.htmlSanctisBT19>.

SCHWABER, K. Scrum development process. 2006. Disponível em: <http://jeffsutherland.com/oopsla/schwapub.pdf>.

SMART, J. F. **Jenkins - The Definitive Guide: Continuos Integration for the Masses: also Covers Hudson.** [S.l.]: O'Reilly, 2011. I-XXII, 1-380 p. ISBN 978-1-449-30535-2.

SOMMERVILLE, I. **Engenharia de Software**. São Paulo: Pearson, Edição: 9ª, 2011.

SON, L. H. et al. Empirical study of software defect prediction: A systematic mapping. **Symmetry**, v. 212, n. 11, p. 1–28, 2019.

VASSALLO, C. Enabling continuous improvement of a continuous integration process. IEEE, San Diego, USA, 2019.

VERHAEG, R. F. Uma abordagem de predição de falhas de software no contexto de desenvolvimento ágil. São Carlos, 2016.

VERNER, J. M. Quality software development: what do we need to improve in the software development process? In: **Proceedings of the 6th international workshop on Software quality**. New York, NY, USA: ACM, 2008. (WoSQ '08), p. 1–2. ISBN 978-1-60558-023-4. Disponível em: <http://0-doi.acm.org.innopac.up.ac.za/10.1145/1370099.1370100>.

Wan, Z. et al. Perceptions, expectations, and challenges in defect prediction. **IEEE Transactions on Software Engineering**, p. 1–1, 2018.

WHEELER, S.; DUGGINS, S. Improving software quality. In: **ACM-SE 36: Proceedings of the 36th annual Southeast regional conference**. New York, NY, USA: ACM, 1998. p. 300–309. ISBN 1-58113-030-9.

YU, A. G. Software crisis, what software crisis? IEEE, Wuhan, China, 2009.

ZHANG, W.; CHALLIS, C. Software component prediction for bug reports. In: LEE, W. S.; SUZUKI, T. (Ed.). **ACML**. PMLR, 2019. (Proceedings of Machine Learning Research, v. 101), p. 806–821. Disponível em: <http://dblp.uni-trier.de/db/conf/acml/acml2019.htmlZhangC19>.

ZHU, A. B. G. X. **Introduction to Semi-Supervised Learning**. [S.l.]: University of Wisconsin, Madison, 2009.

ZOU YONG HU, Z. T. K. S. X. Logistic regression model optimization and case analysis. IEEE, Dalian, China, 2008.