UNIVERSIDADE ESTADUAL DA PARAÍBA
CAMPUS I
CENTRO DE CIÊNCIA E TECNOLOGIA
DEPARTAMENTO DE CIÊNCIAS DA COMPUTAÇÃO
CURSO DE CIÊNCIAS DA COMPUTAÇÃO

LÍVIA CAROLINE TOMAZ SANTOS

PROCESSO DE TESTE PARA VALIDAÇÃO FUNCIONAL DE PROJETOS COM DESENVOLVIMENTO DISTRIBUÍDO DE SOFTWARE: UMA APLICAÇÃO NO PROJETO OCARIOT

CAMPINA GRANDE - PB
2020

LÍVIA CAROLINE TOMAZ SANTOS

PROCESSO DE TESTE PARA VALIDAÇÃO FUNCIONAL DE PROJETOS COM DESENVOLVIMENTO DISTRIBUÍDO DE SOFTWARE: UMA APLICAÇÃO NO PROJETO OCARIOT

Trabalho de Conclusão de Curso apresentado a Universidade Estadual da Paraíba, como requisito parcial à obtenção do título em bacharel em Computação.

Área de concentração: Engenharia de Software.

Orientadora: Profa. Dra. Sabrina de Figueirêdo Souto

CAMPINA GRANDE - PB
2020

S237p Santos, Livia Caroline Tomaz.

Processo de teste para validação funcional de projetos com desenvolvimento distribuído de software [manuscrito] : uma aplicação no Projeto OCARIoT / Livia Caroline Tomaz Santos. - 2020.

67 p. : il. colorido.

Digitado.

Trabalho de Conclusão de Curso (Graduação em Computação) - Universidade Estadual da Paraíba, Centro de Ciências e Tecnologia , 2020.

"Orientação : Profa. Dra. Sabrina de Figueirêdo Souto , Coordenação do Curso de Computação - CCT."

1. Desenvolvimento Distribuído de Software. 2. Engenharia de software. 3. Projeto OCARIoT. I. Título

21. ed. CDD 005.1

Elaborada por Giulianne M. Pereira - CRB - 15/714 BC/UEPB

LÍVIA CAROLINE TOMAZ SANTOS

PROCESSO DE TESTE PARA VALIDAÇÃO FUNCIONAL DE PROJETOS COM DESENVOLVIMENTO DISTRIBUÍDO DE SOFTWARE: UMA APLICAÇÃO NO PROJETO OCARIOT

Trabalho de Conclusão de Curso de Graduação em Ciência da Computação da Universidade Estadual da Paraíba, como requisito à obtenção do título de Bacharel em Ciência da Computação.

Aprovada em 10 de Dezembro de 2020.

Sabrina de F. Souto.

Profa. Dra. Sabrina de Figueiredo Souto (DC - UEPB)
Orientador(a)

Kézia de Vasconcelos Oliveira Dantas

Profa. Dra. Kézia Vasconcelos Dantas (DC - UEPB)
Examinador(a)

Ana Isabella M. Leite

Profa. MSc. Ana Isabella Muniz Leite (DC - UEPB)
Examinador(a)

A Deus e aos meus pais, pelo apoio, incentivo e amor, DEDICO.

AGRADECIMENTOS

Ao longo do meu trajeto na graduação não caminhei sozinha, cada pessoa teve um papel fundamental nesse caminho.

Agradeço primeiramente a Deus, por me dar forças a cada dia para continuar e me possibilitar esta conquista.

Aos meus pais, Antonio Carlos e Socorro Palitó, por me dar todo apoio, amor e dedicação em todos os momentos da minha vida e por me mostrar o quanto a educação é importante.

Aos meus irmãos, Carlos Augusto e Bruna Carla, por ser meus melhores amigos ao longo da vida.

Ao meu cunhado, Carlos Alejandro, por ser como um irmão e amigo.

Ao meu sobrinho, Alejandro, por proporcionar tanta alegria e aprendizagem a cada dia.

Aos meus amigos, Bruno, Manuella e Vitor, por ser um presente que a UEPB me proporcionou e tornaram esta jornada mais leve. Uma amizade que será perpetuada ao longo da minha vida.

Aos professores da instituição por compartilhar todo ensinamento e em especial ao professor Paulo Eduardo, pelas oportunidades que me ofereceu durante os anos de projeto de pesquisa no NUTES.

A minha orientadora, Sabrina Souto, por me acolher e encaminhar na reta final do curso e ajudar a tornar esse sonho realidade, muito obrigada.

A todos que contribuíram de forma direta ou indireta para a conclusão desta pesquisa.

“Ser feliz é encontrar força no perdão, esperanças nas batalhas, segurança no palco do medo, amor nos desencontros. É agradecer a Deus a cada minuto pelo milagre da vida”

Augusto Cury

RESUMO

Com a necessidade de softwares de qualidade, atividades de teste são importantes para a indústria, como forma de garantir a qualidade do produto. Quando não aplicado um processo de teste, um maior número de falhas pode ser encontrado por usuários finais do produto. Atualmente há um aumento de projetos cujo modelo utilizado é o Desenvolvimento Distribuído de Software (DDS), que permitem benefícios como interação de equipes distribuídas geograficamente e redução de custos. Este modelo também apresenta alguns desafios relacionados a comunicação, falta de clareza nas informações, questões culturais e problemas de gestão e processo, que impactam no processo de teste. Como solução proposta aos pontos apresentados fazem-se necessário a adaptação de um processo de teste único para todos os envolvidos de forma que facilite a identificação dos erros do software. Desta forma, o objetivo deste trabalho é propor um processo de teste para projetos com DDS, de forma a promover a qualidade do software em desenvolvimento e o cumprimento dos requisitos especificados. A abordagem foi aplicada no projeto no OCARIoT. Obteve-se como resultado artefatos de teste e a identificação de bugs, proporcionando a correção de erros antes do uso dos sistemas. Também foi possível identificar a cobertura de testes das funcionalidades implementadas e o índice de aprovação dos testes.

Palavras-Chave: Desenvolvimento Distribuído de Software. Engenharia de Software. Projeto OCARIoT.

ABSTRACT

With the need for quality software, testing activities are important for the industry, as a way to guarantee product quality. When it is not a testing process, a greater number of failures can be found by end users of the product. Currently, there is an increase in projects whose model used is Distributed Software Development (DSD), which allows benefits such as the interaction of teams distributed geographically and cost reduction. This model also presents some challenges related to communication, lack of clarity in information, cultural and management and process issues, which impact the testing process. As a solution proposed to the points, it is necessary to adapt a single test process for all involved in a way that facilitates the identification of software errors. Thus, the objective of this work is to propose a testing process for projects with DDS, in order to promote the quality of the software under development and the fulfillment of the required requirements. The approach was applied to the project at OCARIoT. As a result, test artifacts and the identification of bugs were obtained, providing the correction of errors before using the systems. It was also possible to identify the test coverage of the implemented characteristics and the test approval index.

Keywords: Distributed Software Development. Software Engineering. OCARIoT Project.

LISTA DE ILUSTRAÇÕES

LISTA DE TABELAS

SUMÁRIO

1 INTRODUÇÃO

Testes de software são utilizados na indústria para garantir a qualidade do sistema. Desta forma, a validação de software tem por objetivo assegurar que o software esteja de acordo com o que foi especificado. Garantir que o software em desenvolvimento passe por testes permite a redução dos custos de correções que podem ocorrer à medida que os erros vão passando para as etapas seguintes do ciclo de desenvolvimento. Desta forma, "quanto mais cedo erros forem encontrados, menores serão os custos de correção dos erros e maior a probabilidade de corrigi-los corretamente" (MYERS, SANDLER e BADGETT, 2011, p. 20, tradução nossa)[1].

Para que o produto final esteja de acordo com o esperado é necessário um processo de qualidade bem definido, que se adapte as fases do ciclo de vida de desenvolvimento do produto. Portanto, o teste de software é apenas uma parte do processo de garantia de qualidade, juntamente com as boas práticas durante as atividades de análise de requisitos, projeto de sistema, codificação e operação. "O teste proporciona o último elemento a partir do qual a qualidade pode ser estimada e, mais pragmaticamente, os erros podem ser descobertos" (PRESSMAN, 2011, p. 402).

Processo é uma ação contínua e estruturada que visa definir um caminho a ser seguido para se obter um certo resultado. As etapas do processo de teste podem variar de acordo com a metodologia utilizada (DEVMEDIA, 2012). Assim, a adaptação de um processo de teste que funcione em paralelo com o processo de desenvolvimento de software garantirá que a equipe de teste possa encontrar o maior número de erros durante o desenvolvimento, a fim de serem mitigados antes que o produto seja finalizado. A qualidade do software está diretamente relacionada com a qualidade do processo de teste utilizado.

No contexto de equipes geograficamente distribuídas, cujo modelo utilizado é o Desenvolvimento Distribuído de Software (DDS), também conhecido por Desenvolvimento Global de Software (GSD), as principais características que o diferenciam do desenvolvimento tradicional são "dispersões geográficas (distância física), dispersão temporal (diferenças de fuso horário) e diferenças socioculturais (idioma, tradições, costumes, normas e comportamento)" (AUDY e PRIKLADNICKI, 2008, p.44).

[1] "the earlier errors are found, the lower the costs of correcting the errors and the higher the probability of correcting them correctly."

Estas características quando não consideradas podem acarretar problemas na qualidade do software em desenvolvimento. Como apontado por Jain e Suman (2015), alguns dos impactos negativos do DDS durante as atividades de teste estão relacionados a comunicação, por ser insuficiente ou inadequada, causando principalmente problemas entre testadores e desenvolvedores. Desta forma, é diante dos problemas identificados no uso do DDS, que se faz necessário a adaptação de um processo de teste que se adeque ao modelo de desenvolvimento.

1.1 Problema

Apesar dos benefícios do uso de DDS, com a redução nos custos de mão de obra e a troca de conhecimento, dificuldades com relação aos processos podem interferir no resultado final do software desenvolvido.

De acordo com Audy e Prikladnicki (2008), ainda existem lacunas nos estudos de testes em ambientes distribuídos. Entre os problemas apontados pelos autores está a “falta de precisão nos documentos de teste que são trocados entre as equipes distribuídas” (AUDY e PRIKLADNICKI, 2008, p.187).

É comum que em equipes distribuídas não exista clareza na troca de informações, dados e documentos, como requisitos e regras de negócios. Além das informações provenientes de outras equipes, também é importante que os artefatos e resultados obtidos através do processo de teste sejam passados de forma clara para as demais equipes. Estes são sem dúvida, os principais desafios a serem enfrentados pela equipe de teste.

1.2 Solução

São necessárias que atividades de teste sejam realizadas durante o ciclo de desenvolvimento de software, principalmente em projetos cujo modelo utilizado é o DDS, nos quais são identificados problemas relacionados ao gerenciamento de projetos e processos. Para se obter um bom resultado deste processo, a solução proposta é a adaptação de um processo de teste único para todos os envolvidos no processo de teste, definindo padrões de documentação e ferramentas utilizadas.

O processo de teste proposto visa facilitar a troca de informações com os demais membros da equipe, principalmente informações relacionadas aos erros identificados, de forma que possam ser mitigados antes que o produto esteja em produção.

A solução proposta será aplicada no projeto OCARIoT (Smart Childhood Obesity CARing Solution Using IoT Potencial)[2], que será apresentado no capítulo 4, por ser um projeto em parceria com instituições localizadas em países da Europa e no Brasil, desta forma, seguindo o modelo DDS.

1.3 Objetivos

O objetivo geral deste trabalho é propor um processo de teste para projetos com desenvolvimento distribuído, de forma a melhorar a qualidade do sistema e o cumprimento dos seus requisitos.

Com base no objetivo geral, os objetivos específicos são:

- Propor um processo de teste que se adeque com o desenvolvimento distribuído de software;
- Contribuir para diminuição dos problemas de comunicação entre as equipes de teste e demais *stakaholders* envolvidos nesses tipos de projetos, de forma a minimizar os problemas comumente encontrados nesse modelo de desenvolvimento de software;
- Aplicar o processo proposto no contexto do OCARIoT.

1.4 Estrutura da monografia

A monografia está estruturada nos seguintes capítulos:

- Capítulo 2: apresenta os conceitos básicos para compreensão do problema contextualizado neste trabalho;
- Capítulo 3: apresenta o processo de teste para o desenvolvimento distribuído de software e como deve ser aplicado;
- Capítulo 4: descreve a aplicação do processo no OCARIoT;

[2] http://www.ocariot.com.br/

- Capítulo 5: apresenta as conclusões, contribuições e propostas de continuação da pesquisa.

2 FUNDAMENTAÇÃO TEÓRICA

Este capítulo fornece embasamento teórico para a compreensão da pesquisa realizada. Nesta seção, são introduzidos os conceitos relacionados ao desenvolvimento distribuído de software e teste de software e, posteriormente, as técnicas e tipos de testes que auxiliam na aplicação do processo de teste. Por fim, discutiremos sobre o gerenciamento dos *bugs* encontrados durante o processo de teste.

2.1 Desenvolvimento Distribuído de Software

O desenvolvimento distribuído de software (DDS) tem se transformado em uma prática comum no desenvolvimento de software. Com o uso do DDS os membros da equipe de desenvolvimento, teste, análise e gestão podem estar localizados em vários lugares diferentes durante o ciclo de vida do software (JIMÉNEZ, PIATTINI e VIZCAÍNO, 2009). O DDS pode ser definido como desenvolvimento global de software (GSD, do inglês *Global Software Development*), quando a dispersão das equipes envolvidas é de escala global.

O DDS permite a interação de pessoas e organizações, economia de custos, troca de tecnologias e estilos de trabalho. Alguns dos fatores que contribuíram para o aumento deste modelo de trabalho são apresentados por Prikladnicki et al. (2004),

> - Custo mais baixo e disponibilidade de mão de obra;
> - Evolução e maior acessibilidade a recursos de telecomunicação;
> - Evolução das ferramentas de desenvolvimento;
> - A necessidade de possuir recursos globais para utilizar a qualquer hora;
> - As vantagens de estar perto do mercado local;
> - A formação de equipes virtuais para explorar as oportunidades de mercado;
> - A pressão para o desenvolvimento *time-to-market*, utilizando as vantagens proporcionadas pelo fuso horário diferente, no desenvolvimento conhecido como *round-the-clock*, ou seja, o desenvolvimento quase que contínuo. (PRIKLADNICKI et al., 2004, p. 154)

Apesar dos benefícios apresentados, este modelo apresenta um desafio de liderança e gerenciamento. Os principais desafios são apontados por Herbsleb e Moitra (2001):

Problemas estratégicos: dificuldades com a divisão das tarefas entre as equipes e a resistência da organização ao uso deste modelo de trabalho.

Problemas culturais: diferenças culturais podem aumentar os problemas de comunicação, sendo necessário a estruturação da equipe, padronização no estilo de comunicação e o senso de tempo.

Comunicação inadequada: problemas relacionados a falta de formalidade na comunicação durante as fases iniciais do projeto, na definição das responsabilidades, escalonamento de problemas e atualização do status do projeto. Outro problema de comunicação apontado pelos autores é a falta de conversa informal entre os membros das equipes, o que prejudica no acompanhamento das atividades em andamento causando desalinhamento e retrabalho.

Gestão de conhecimento: O compartilhamento das informações permite que a equipe saiba as prioridades das tarefas a serem desenvolvidas. Com a gestão do conhecimento, informações e conhecimentos podem ser reutilizados de forma a diminuir tempo e custo. Sendo assim, as informações fornecidas através dos artefatos devem estar sempre atualizadas.

Problemas de gerenciamento de projetos e processos: Podem acarretar diferentes entendimentos e definições das etapas do processo, assim como dos critérios de entrada e saída das atividades pelos membros da equipe que estão separados.

Problemas técnicos: Os problemas relacionados a técnicos estão relacionados com a infraestrutura, redes e dados. Sendo necessário o planejamento nas atividades de gerenciamento de configuração.

Sendo assim, não se pode ignorar a necessidade de um nível mais alto de comunicação e clareza para compensar a falta de conhecimento entre as pessoas envolvidas, tempo, diferenças culturais e os problemas relacionados ao processo e tecnologia, como apontados por Herbsleb e Moitra (2001).

O DDS pode ser caracterizado por vários níveis de dispersões de acordo com a distâncias entre os envolvidos, assim como retratados por AUDY e PRIKLADNICKI (2008):

Mesma localização física: os membros da equipe estão na mesma localização física.

Distância nacional: os membros da equipe estão em diferentes localizações de um mesmo país.

Distância continental: os membros da equipe estão em países diferentes de um mesmo continente.

Distância global: como apontado anteriormente com o GSD, membros da equipe estão localizados em continentes diferentes.

O trabalho com DDS fornece alguns desafios para as equipes envolvidas, entretanto, ainda se apresenta como uma solução na redução de custos e tempos de produção para as empresas de software.

2.2 Teste de Software

Mesmo com o planejamento durante as atividades de desenvolvimento de um software falhas podem acontecer, por isso é importante que durante o processo atividades de testes sejam realizadas. Para Myers (1979), teste de software é o processo de executar um programa ou sistema com intenção de encontrar defeitos. No entanto, também podemos definir como o processo de verificar se um software está fazendo o que deveria fazer, de acordo com seus requisitos. A partir dos resultados da execução dos testes é possível verificar o grau de qualidade do software, de forma a reduzir falhas quando o sistema esteja em produção.

De acordo com Pressman,

> O teste de software é um elemento de um tópico mais amplo, muitas vezes conhecido como verificação e validação (V&V). Verificação refere-se ao conjunto de tarefas que garantem que o software implementa corretamente uma função específica. Validação refere-se a um conjunto de tarefas que asseguram que o software foi criado e pode ser rastreado segundo os requisitos do cliente. (PRESSMAN, 2011, p. 402).

Desta forma, a validação funcional será responsável por garantir que o software que está sendo testado supra as necessidades dos *stakeholders* (partes interessadas), quando esteja em uso.

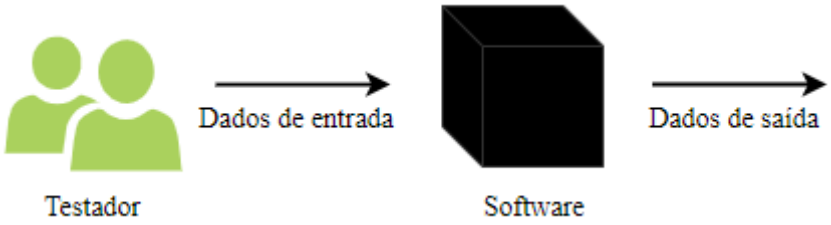
2.2.1 *Tipos de Teste*

Existem diversas formas de testar um software de acordo com o objetivo do testador e o que será testado. Durante o teste de software, os tipos e técnicas de teste podem ser aplicados de forma complementar para garantir a qualidade esperada.

2.2.1.1 Funcional

O teste funcional tem por objetivo constatar que o que foi desenvolvido está de acordo com o especificado. Uma das principais técnicas para realização do teste funcional é a de caixa-preta, que permite que o testador atue como usuário verificando a funcionalidade e a aderência aos requisitos sem se basear no conhecimento lógico do código (RIOS e MOREIRA, 2013), como mostra a Figura 1 a seguir:

Figura 1 - Teste caixa-preta



Fonte: Elaborada pelo autor (2020).

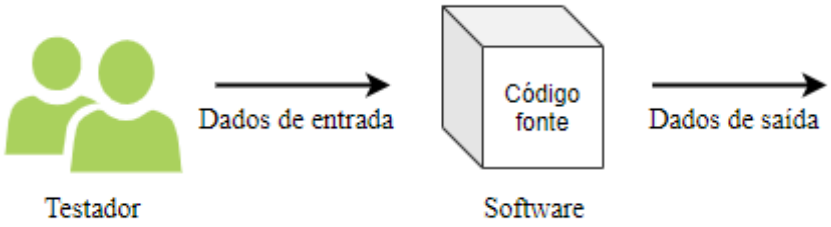
2.2.1.2 Não-Funcional

De acordo com o *Brazilian Software Testing Qualifications Board* (BSTQB), representante no Brasil do *International Software Testing Qualifications Board* (ISTQB), "os testes não funcionais de um sistema avaliam as características de sistemas e de softwares, como usabilidade, eficiência de desempenho ou segurança. " (BSTQB, 2018, p. 44). Assim os testes poderão garantir alguns aspectos como estabilidade após alguma falha, sigilo dos dados e escalabilidade. Ainda segundo o BSTQB, o uso de técnicas de teste caixa-preta também pode ser utilizado para validação dos requisitos não-funcionais.

2.2.1.3 Estrutural

Os testes estruturais são realizados a partir da análise do código fonte do software. Ao contrário dos testes anteriores, que utilizavam técnicas de teste caixa-preta, o teste estrutural também é conhecido como teste de caixa-branca, que têm por objetivo verificar o comportamento interno do software como apresentado na Figura 2 abaixo:

Figura 2 - Teste caixa-branca



Fonte: Elaborada pelo autor (2020).

Durante os testes de caixa-branca é avaliado a porcentagem do código que foi coberta durante os testes. Existem limitações ao uso apenas dos testes de caixa-branca, já que testar apenas o código e a estrutura do software não conseguem determinar a ausência de recursos. (WHITE, 1987).

2.2.1.4 Teste de Regressão

Os testes de regressão são realizados em uma parte do software que já havia sido previamente testada para garantir o perfeito funcionamento do mesmo após incremento ou alterações em outra parte do código. O objetivo é garantir que após a realização de mudanças o software permaneça funcionando corretamente (RIOS e MOREIRA, 2013).

Realizar testes de regressão em todo o software pode ser exaustivo, sendo assim a opção em muitos casos é apenas retestar partes do software onde ocorreu uma maior concentração de bugs anteriormente.

2.2.2 *Estratégias de Testes*

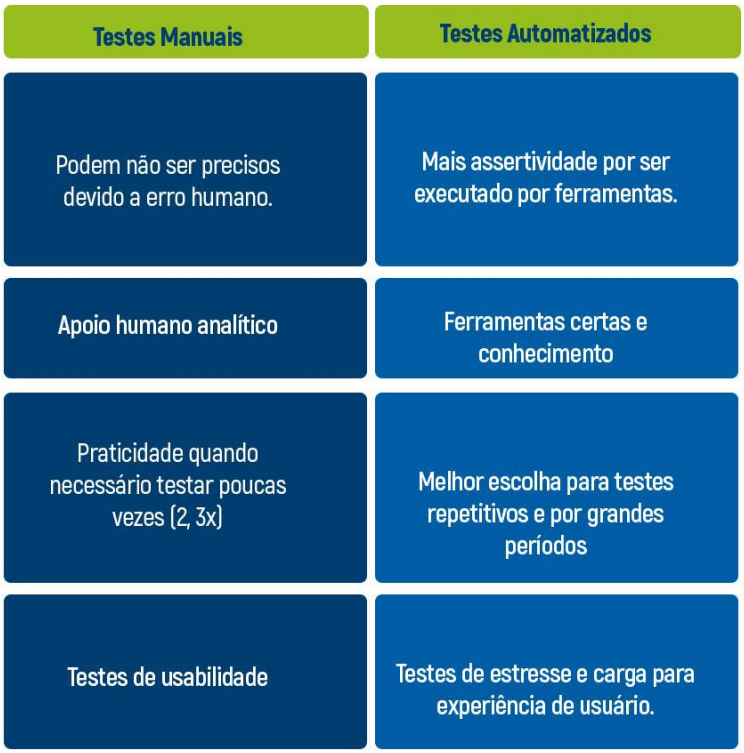
A escolha da estratégia utilizada durante o processo de teste dependerá de fatores como: objetivo do teste, o que será testado, tecnologia utilizada, prazos e custos. A partir da análise dos fatores, a equipe responsável optará pelo uso de estratégias de teste manual, automatizado ou as duas de forma complementar.

Os testes manuais como o próprio nome sugere são realizados manualmente pelo testador. Esta estratégia de teste é recomendada quando são testados aspectos de design e usabilidade, onde é necessário a avaliação de um usuário real. Testes manuais são utilizados em projetos cujos requisitos sofrem constantes alterações. Não são recomendados testes

manuais quando há uma maior necessidade de repetição de testes e o prazo do projeto é curto, pois requer mais tempo de trabalho.

Testes automatizados ao contrário dos manuais são executados pela máquina, ou seja, o testador implementa scripts de testes e coloca para que estes sejam executados. Os testes automatizados são mais utilizados quando há necessidade de testes de regressão ou de carga, já que permitem a redução custos e tempo de execução. Diferente dos testes manuais, esta estratégia de teste não é recomendada em projetos cuja aplicação sofre constante alterações por ser menos viável a mudança de scripts de teste.

Figura 3 - Comparativo entre os Testes Manuais e Testes Automatizados



Fonte: https://blog.cedrotech.com/testes-de-software-manuais-x-automatizados/

Como observado na Figura 3, nenhum dos testes oferece um nível completo de cobertura, desta forma, as duas estratégias podem ser utilizadas de forma complementar para aumentar a qualidade do software. Para Aniche (2015, p. 153), “Testar não é ter simplesmente ter testes automatizados. É maximizar o feedback que essa bateria pode lhe dar. E para isso, você precisa balancear entre todos os diferentes níveis de teste.”.

2.2.2.1 Teste de Aplicações Web

Aplicações *Web* são sistemas que podem ser acessados por navegadores *web*. Estes sistemas são executados em um servidor remoto, sendo assim, não precisam ser baixados, pois são acessados por meio de uma rede.

Muitos tipos de teste podem ser utilizados para Aplicações *Web*, de acordo com as características do sistema, complexidade, risco ou até requisitos contratuais. Como exemplo de testes que podem ser utilizados para Aplicações *Web* podemos citar:

- **Teste de Usabilidade:** tem por objetivo avaliar a facilidade de uso do sistema da perspectiva dos usuários, principalmente aspectos ligados a interface;
- **Teste de Carga:** avalia o comportamento do sistema com o aumento da carga a ser processada;
- **Teste de Desempenho:** busca encontrar o limite de carga em que o sistema funciona em perfeito desempenho;
- **Teste de Segurança:** visa avaliar a vulnerabilidade do sistema a possíveis ataques e invasões.

Uma das ferramentas mais comuns para criação de testes automatizados para Aplicações *Web* é o Selenium. O framework de código aberto oferece diferentes componentes, entre eles o Selenium WebDriver.

> WebDriver é uma API e protocolo que define uma interface com neutralidade de linguagem para controlar o comportamento dos navegadores da web. Cada navegador é apoiado por uma implementação WebDriver específica, chamada de driver. O driver é o componente responsável por delegar ao navegador e controla a comunicação de e para o Selenium e o navegador. (SELENIUM, 2020, tradução nossa).

O Selenium WebDriver consegue automatizar os principais browsers disponíveis no mercado, sendo compatível com diversas linguagens de programação como apresentado na Figura 4 abaixo:

Figura 4 - Selenium WebDriver

Fonte: https://www.quora.com/What-are-the-Components-of-Selenium

2.2.2.2 Teste de API

API - "*Application Programming Interface*" é um conjunto de protocolos e padrões de programação que permitem a comunicação entre diferentes softwares sem interferência de usuários.

O teste de API tem por objetivo assegurar o correto funcionamento, desempenho e segurança na troca de dados e comunicação entre sistemas. Segundo De (2017, p. 153, tradução nossa) "O teste de API é como o teste de caixa branca. O teste de uma API envolve um software especial que envia mensagens para um *endpoint* da API conforme definido na interface, obtém a saída, registra e analisa a resposta.".

Entre os frameworks JavaScript disponíveis para o teste de API está o Mocha e a biblioteca Chai, que são executados no Node.js e no browser. O Mocha fornece um ambiente de execução de testes que necessita de uma biblioteca de asserções como o Chai. O Chai permite que sejam utilizadas as técnicas de desenvolvimento TDD (do inglês *Test Driven Development*) com o uso da interface *assert* e a técnica BDD (do inglês *Behavior Driven Development*) com as interfaces *should* e *expec*, como observado na Figura 5 a seguir:

Figura 5 - Interfaces da biblioteca Chai

Fonte: https://www.chaijs.com/

2.2.2.3 Teste de Aplicações Móveis

Aplicações Móveis, também conhecidas como App, são programas desenvolvidos para aparelhos móveis. Os aplicativos móveis estão divididos em três tipos:

Aplicativos Nativos: são aplicativos desenvolvidos em uma linguagem de programação específica para o sistema operacional do dispositivo móvel no qual será instalado. Este tipo de aplicativo está disponível para download em plataformas como Play Store para dispositivos com sistema operacional Android e App Store para o sistema operacional iOS.

Web Apps: diferentemente dos Aplicativos Nativos, os Web Apps não precisam de download para funcionar. Os Web Apps são desenvolvidos utilizando as tecnologias para desenvolvimento de aplicações web se adaptando ao uso em dispositivos móveis.

Aplicativos Híbridos: são formados pela combinação de Aplicativos Nativos e Web Apps. Os Aplicativos Híbridos são desenvolvidos utilizando tecnologias web e encapsulado em uma aplicação nativa. Este tipo de aplicativo pode ser baixado na plataforma correspondentes a cada sistema operacional.

O ambiente no qual as Aplicações Móveis são executadas é limitado, por esta razão devem ser levados em conta fatores como: “Resolução da tela, Limitações de hardware, Uso caro de dados, Problemas de conectividade, Possibilidades de interação limitada” (ISLAM, ISLAM e MAZUMDER, 2010, p. 72, tradução nossa). Desta forma, estes fatores devem ser levados em conta durante o teste das aplicações móveis, validando o funcionamento correto da aplicação mesmo com essas limitações.

Entre os frameworks de testes automatizados de Aplicações Móveis está o Appium. O framework tem código aberto e permite teste de Aplicativos Nativos, Web Apps e Aplicativos

Híbridos. O Appium funciona com base no Selenium WebDriver, executando assim testes em diversas linguagens de programação como: Java, Ruby, C, Python, Perl (SINGH, GADGIL e CHUDGOR, 2014) como apresentado na Figura 6 abaixo:

Figura 6 - Appium

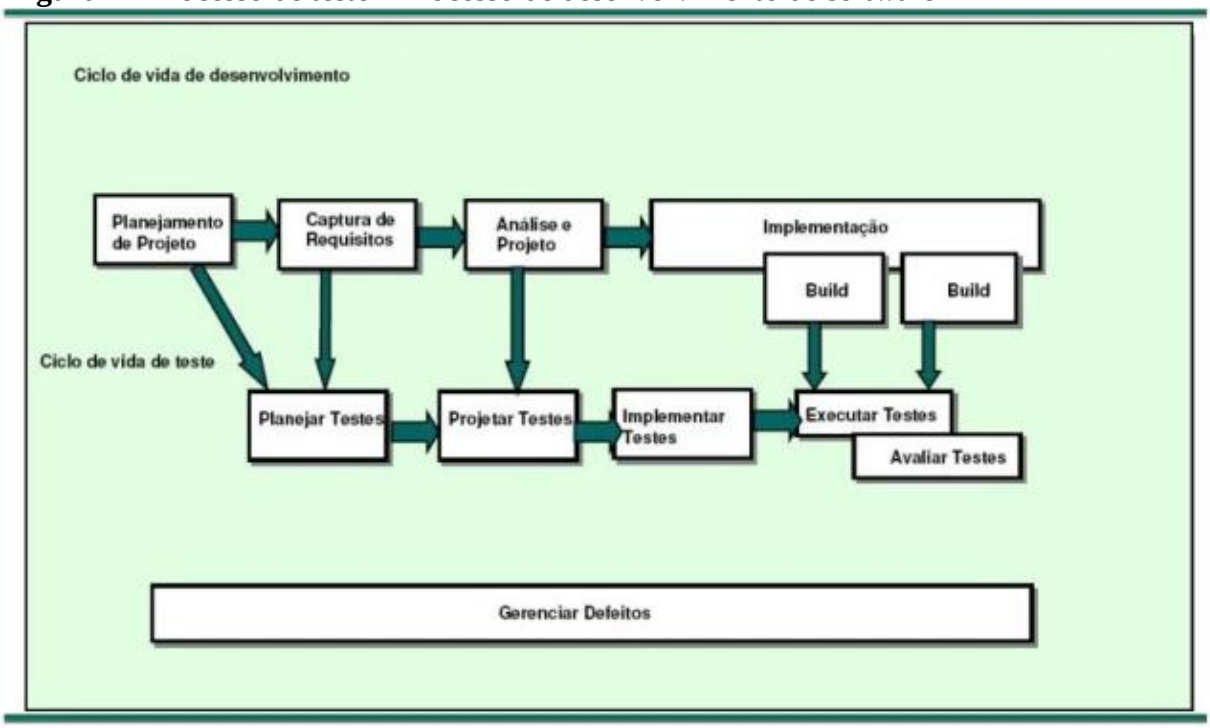
Fonte: https://www.qedgetech.com/blog/appium-mobile-application-testing/

O framework Appium é multiplataforma, podendo executar os testes em aplicativos de diferentes sistemas operacionais como Android, iOS e Windows.

2.3 Processo de Teste

O processo de teste define o conjunto de atividades, fases, artefatos e atribuições definidas com o propósito de testar um software de forma a garantir a qualidade do produto. Para o BSTQB “Não existe um processo universal de teste de software, mas há conjuntos comuns de atividades de teste sem as quais os testes terão menor probabilidade de atingir seus objetivos estabelecidos” (BSTQB, 2018, p. 18). As principais atividades definidas durante o processo incluem planejar, projetar, implementar, executar e posteriormente avaliar os resultados até a correção dos defeitos encontrados, como apresentado na figura 7 a seguir:

Figura 7 - Processo de teste x Processo de desenvolvimento de software



Fonte: https://testesw.wordpress.com/processo-de-testes/

Ainda como observado na figura 7, as atividades do processo de teste devem ir em paralelo ao processo de desenvolvimento do software e se adaptar a metodologia de desenvolvimento, seja ela tradicional ou ágil. “Ter um processo de garantia da qualidade em todo o ciclo de desenvolvimento do software permite que um número maior de defeitos seja descoberto antecipadamente, evitando a migração destes para as fases seguintes” (BARTIÉ, 2002, p. 27).

Durante cada etapa do processo de teste vários artefatos podem ser produzidos, entre os principais estão o Plano de Teste e Caso de Teste.

2.3.1 *Plano de Teste*

O Plano de teste é um documento escrito durante a fase de planejamento e tem entre as suas informações as funcionalidades que serão testadas, cronograma, equipamentos, papéis, riscos e critérios nos quais os testes podem ser considerados como finalizado (BLANCO, 2012). “O planejamento pode ser documentado em um plano de teste principal e em planos de teste separados para cada nível de teste” (BSTQB, 2018, p.75).

As informações contidas no plano de teste podem ser alteradas em função da evolução do projeto que está sendo testado, por razões como:

- Alterações no escopo de teste;
- Definições de novos tipos de teste;
- Novos critérios de aceite;
- Identificação de novos riscos a partir de resultados de testes.

O principal objetivo do plano de teste é a definição de metas a serem obtidas a partir das atividades de teste. As informações contidas no documento podem ser compartidas com gerentes, analistas e equipe de desenvolvimento provendo o entendimento e a transparência das atividades realizadas pela equipe de teste.

2.3.2 *Caso de Teste*

A partir da definição do plano de teste serão criados casos de teste para validação das funcionalidades e requisitos do software. Para validação dos requisitos pode ser necessária a criação de vários casos de teste, de forma que todos os cenários sejam cobertos por testes. As informações necessárias para especificação dos casos de testes são fornecidas através do documento de requisitos.

Entre as informações contidas no caso de teste estão a descrição breve do teste, roteiro com a descrição passo a passo a ser seguido, resultados esperados e os resultados obtidos pelo responsável pela execução do teste. O caso de teste visa identificar se o que foi implementado cumpre com os critérios do que foi requisitado. (BLANCO, 2012).

Existem diversas ferramentas que auxiliam na especificação de casos de teste, a partir do uso de tabelas, mapa mental e sistemas de gerenciamento de testes, ou do uso combinado destes recursos.

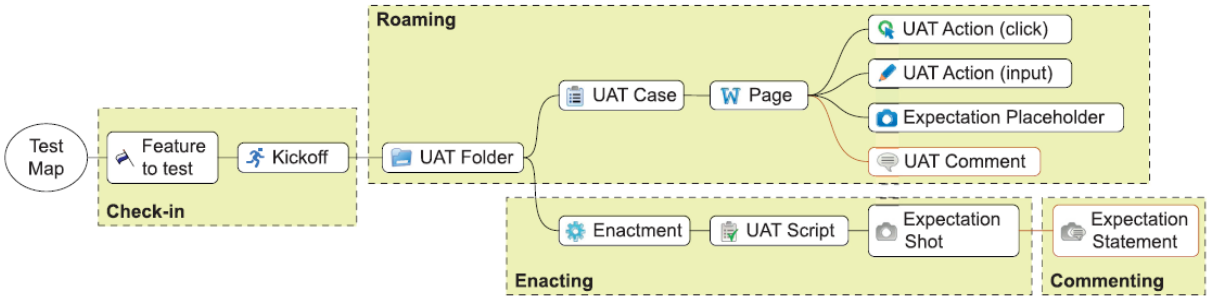
2.3.2.1 Mapa Mental

A realização de testes de aceitação é necessária para que os clientes avaliem se o software funciona de forma correta atendendo a suas expectativas. Otaduy e Diaz (2017) apontam os benefícios do uso de mapa mental na criação de teste para realização dos testes de aceitação, por fornecer uma notação mais familiar aos clientes. “Mapa mental é uma técnica gráfica usada para representar palavras, conceitos, tarefas ou outros itens conectados ou

organizados em torno do tópico central ou ideia" (SIOCHOS e PAPATHEODOROU, 2011, p. 39, tradução nossa).

Ainda segundo Otaduy e Diaz (2017), esta notação de teste é benéfica quando é necessária a colaboração de equipes que estão geograficamente distribuídas, desta forma, podendo ser aplicada em projetos com DDS. Os testes elaborados a partir de mapas mentais são chamados de Mapa de Teste (*Test Map*), como observado na Figura 8 abaixo:

Figura 8 - Mapa de teste para capturar as preocupações de testes de aceitação



Fonte: OTADUY e DIAZ (2017)

O uso de mapas mentais na criação de casos de teste tem aumentado principalmente em projetos com metodologia ágil, já que fornecem uma forma mais simples e objetiva de documentar o que será testado e critérios de aceitação. Atualmente existem diversas plataformas para criação de mapas mentais, entre elas Xmind, FreeMind e MindMeister.

O MindMeister é uma plataforma online que permite a colaboração entre as equipes para criação dos mapas mentais, fornecendo também visualização de históricos, brainstorm colaborativo e designs personalizados.

2.4 Gerenciamento de Teste

Durante o processo de teste o uso de ferramentas de gerenciamento de testes auxilia testadores na criação e gestão de artefatos de teste. Um dos principais fatores para o uso destas ferramentas é a possibilidade de colaboração entre os testadores, permitindo também que gestores e analistas possam acompanhar as atividades de teste.

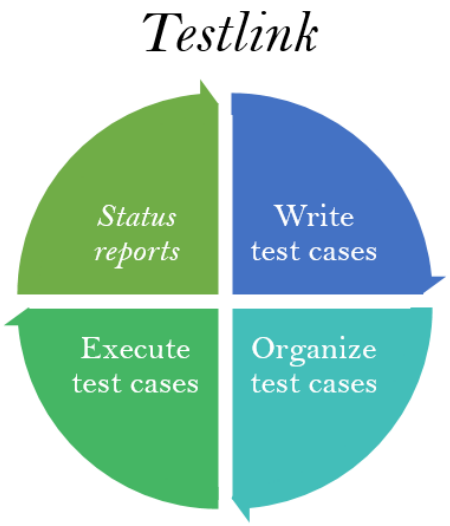
As ferramentas de gerenciamento de teste contribuem desde a criação de planos de teste, casos de teste, planos para execuções de diferentes releases do software e geração de relatórios. Podemos encontrar no mercado diversas ferramentas gratuitas e pagas, que

oferecem diversos recursos de acordo com as necessidades da equipe de teste e o tamanho do projeto. Como exemplo ferramentas gratuitas temos o Testlink, TestiTool e TestMaster.

2.4.1 *Testlink*

O Testlink é uma ferramenta de código aberto criada em 2005, que permite a gestão de vários projetos. Entre as funcionalidades do Testlink está a criação de planos de testes, casos de teste e reporte de execução de testes manuais e automatizados, como observado na Figura 9. Com a ferramenta também é possível criar requisitos do sistema e associar aos casos de teste permitindo a rastreabilidade entre os artefatos (DE AMORIM et al., 2016).

Figura 9 - Funcionalidades do Testlink



Fonte: http://www.qaondemand.ca/testlink-centralized-test-case-management-system/

Membros da equipe de teste podem trabalhar de forma paralela, já que o Testlink é acessado através da *web* "favorecendo sua utilização em equipes distribuídas geograficamente [...]" (Caetano, 2008 apud DE AMORIM et al., 2016, p. 298). É possível realizar a integração do Testlink com ferramentas de gerenciamento de *bugs*, como Mantis e Bugzilla.

2.5 Ambiente de Teste

Para que os resultados obtidos na execução dos testes possam garantir a qualidade do software é necessária que sua realização seja em um ambiente o mais parecido possível ao de produção, assim, é maior a probabilidade de que o software funcione corretamente durante o uso por parte de usuários finais. Um ambiente de teste controlado requer que sejam definidos recursos de hardware, software e massa de dados necessários para a realização dos testes.

Os benefícios do uso de ambientes de teste é o aumento da produtividade e eficiência, redução de custos em produção e aumento de confiabilidade na cobertura de um maior número de cenários (EDUCBA, 2020).

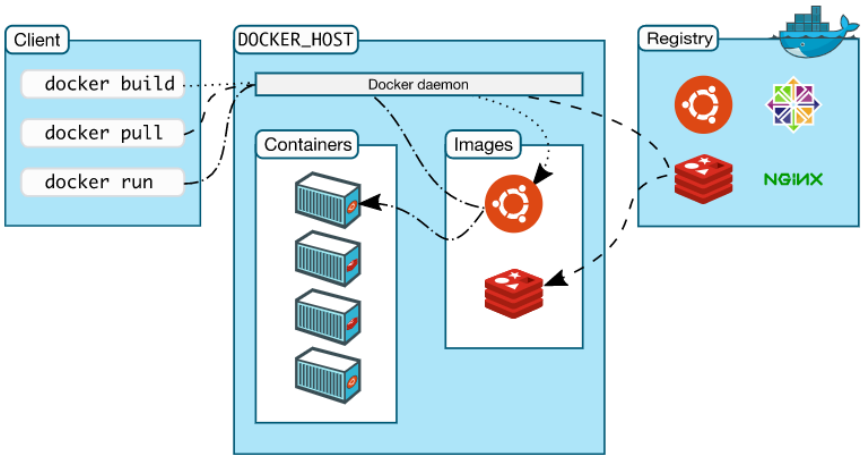
Algumas ferramentas podem auxiliar no processo de criação de um ambiente de teste, entre elas: ferramentas de controle de versão, máquinas virtuais, VPN (Rede Privada Virtual, do inglês *Virtual Private Network*), *containers* ou ferramentas de automação.

2.5.1 *Docker*

O Docker é uma plataforma aberta desenvolvida na linguagem Go lançada em 2013. A plataforma "oferece a capacidade de empacotar e executar um aplicativo em um ambiente livremente isolado denominado *container*" (DOCKER, 2020, tradução nossa).

Como apresentado na Figura 10, da arquitetura do Docker, as imagens fornecem as instruções necessárias para a criação dos *containers*, podendo ser geradas diferentes imagens quando são necessárias novas dependências para o aplicativo que será executado.

Figura 10 - Arquitetura do Docker



Fonte: https://docs.docker.com/get-started/overview/

O uso do Docker permite que testes realizados em máquinas com diferentes configurações sejam executados em um mesmo ambiente de teste, já que a plataforma usa virtualização a nível de sistema operacional e as imagens contêm todas as dependências necessárias para a criação do *container* da aplicação.

2.6 Gerenciamento de Bugs

De acordo com o Institute of Electrical and Electronics Engineers (IEEE), norma número 610.12-1990, um Bug é um erro ou defeito (IEEE, 1990). O Gerenciamento de Bugs tem como propósito documentar e rastrear os erros encontrados durante o ciclo de vida do software. A partir da execução dos testes, os bugs identificados devem ser documentados contendo informações como o passo a passo para reprodução, versão do software testado, nível de criticidade e prioridade. Informações como criticidade e prioridade permitem que a equipe de desenvolvimento atue de forma rápida em bugs de maior impacto. Assim, podemos afirmar que o Gerenciamento de bugs permite que riscos sejam mitigados.

O ciclo de vida do bug se inicia a partir de sua identificação, passando por vários estados até o momento em que são encerrados e não seja mais possível sua reprodução. Os estados definidos no ciclo de vida do bug podem variar de acordo com a metodologia de desenvolvimento e ferramentas de gerenciamento de bugs utilizadas. É importante que os membros da equipe responsável pelo sistema tenham conhecimento dos estados e fluxos, assim como sua responsabilidade durante o ciclo de vida do bug.

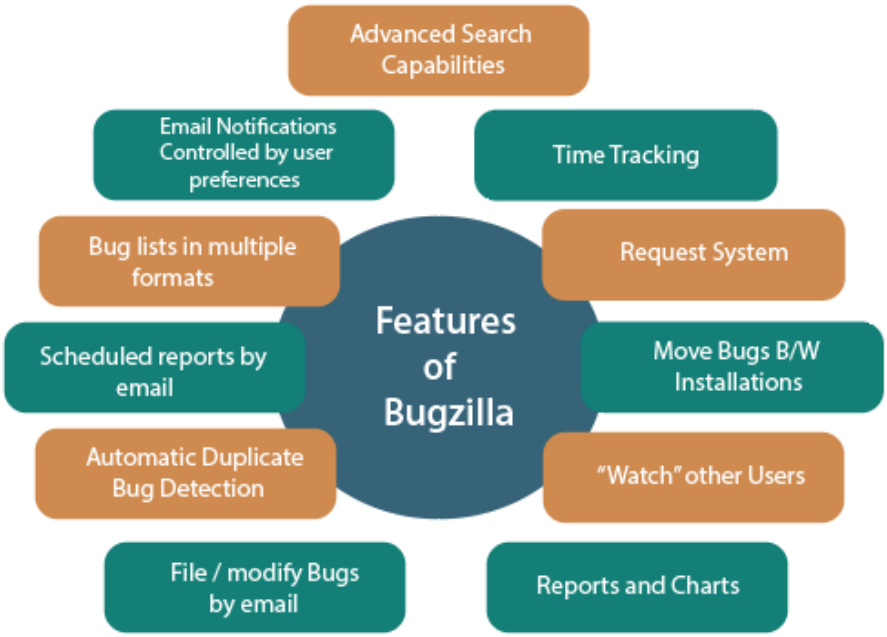
Os erros encontrados durante o processo de teste podem ser cadastrados em ferramentas de gerenciamento de bugs. Estes tipos de ferramentas permitem que: informações dos bugs estejam disponibilizadas a todos, padronização nas informações cadastradas e geração de relatórios. Entre as ferramentas gratuitas de rastreamento de bugs disponíveis no mercado está o Bugzilla, Jira e Redmine.

2.6.1 *Bugzilla*

O Bugzilla é uma aplicação web que foi criada para dar suporte ao desenvolvimento do browser Mozilla por sua equipe de desenvolvedores (SERRANO e CIORDIA, 2005). Alguns dos benefícios apontados no uso da ferramenta são: filtros avançados de busca,

notificação por e-mail, relatórios e gráficos, controle de tempo, detecção automática de bugs duplicados, como observado na Figura 11 abaixo:

Figura 11 - Funcionalidades do Bugzilla



Fonte: https://www.javatpoint.com/bugzilla

No cadastro dos bugs devem ser fornecidas informações relevantes como passo a passo para reprodução do bug, descrição do erro, ambiente de teste, versão do software, gravidade, prioridade e quais os membros responsáveis pela correção do erro encontrado.

Como citado anteriormente, os estados do ciclo de vida do bug podem variar de acordo com a ferramenta de rastreamento de bugs utilizada, sendo assim também é possível definir no Bugzilla quais são as transições possíveis entre cada estado. Por ser uma ferramenta colaborativa de gerenciamento de bugs, como apontado por Aljedaani e Javed (2018), testadores e usuários geograficamente distribuídos podem cadastrar bugs no Bugzilla.

2.7 Considerações Finais

Neste capítulo foram apresentados os fundamentos teóricos que contextualizam a pesquisa elaborada, proporcionando conceitos sobre o desenvolvimento distribuído de software, e por fim o teste de software (tipos de teste, estratégias, processo, gerenciamento de teste, ambiente de teste e gerenciamento de bugs). Nos próximos capítulos serão utilizados os

conceitos abordados para definição de um processo de teste adequado para projetos com metodologia DDS e a aplicação no projeto OCARIoT.

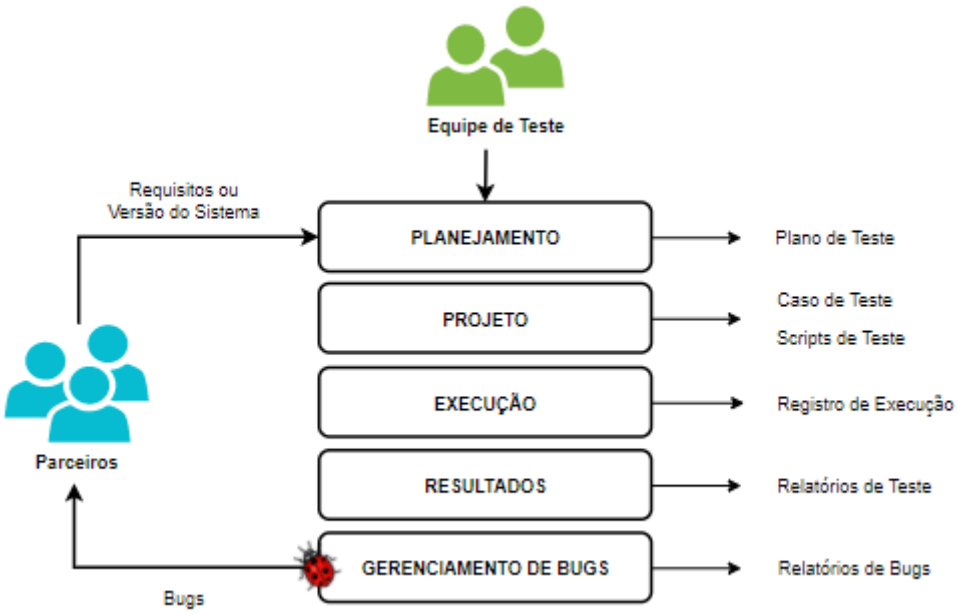
3 PROCESSO DE TESTE PARA VALIDAÇÃO FUNCIONAL

Neste capítulo, apresentamos as etapas do processo de teste que tem como objetivo garantir que o software atenda a todos os requisitos funcionais. Com os desafios do DDS definir um processo de teste que se adeque a esta realidade pode ser difícil. Para Audy e Prikladnicki (2008), os desafios do DDS podem ser categorizados com relação a pessoas, tecnologias, comunicação, gestão e processos. Para esses mesmos autores, os desafios relacionados ao processo fazem relação a arquitetura do software, engenharia de requisitos, gerência de configuração e processo de desenvolvimento.

O impacto do DDS no desenvolvimento de software não interfere apenas no processo de teste, mas também na garantia de qualidade do produto que está sendo desenvolvido sob essa metodologia. De forma a minimizar os impactos negativos do DDS é necessário adotar um processo de teste único para todos os envolvidos. Como apontado no capítulo anterior, o processo de teste não possui um modelo universal, desta forma, torna-se importante também o uso de normas, padrões de documentação e ferramentas.

Para elaboração do processo de teste utilizado na validação funcional, foi tomado como parâmetro as atividades e tarefas estabelecidas pelo BSTQB no *Certified Tester Foundation Level Syllabus*, e como base para documentação de teste a norma ISO/IEC/IEEE 29119 *Software and systems engineering - Software testing*, que é referenciada no Syllabus. O processo estabelecido segue o modelo apresentado na Figura 12 abaixo:

Figura 12 - Fases do processo de teste para validação funcional



Fonte: Elaborada pelo autor (2020).

Conforme descrito na Figura 12, o processo de teste foi dividido em cinco fases gerenciadas pela equipe de teste: Planejamento, Projeto, Execução, Resultados e Gerenciamento de Bugs. O planejamento é a fase inicial para validação funcional, onde é elaborado o Plano de Teste no qual são definidos o escopo, atividades e recursos necessários para realização dos testes. Com as informações definidas na fase de planejamento, serão projetados os cenários e casos de testes a serem executados, e scripts de teste para os testes automatizados. Todos os resultados obtidos com a execução dos casos de teste, contidas no registro de execução, são relatados, assim como os bugs identificados para posterior correção. Todas as informações dos resultados dos testes e do gerenciamento de bugs podem ser gerenciadas através de relatórios.

A partir de mudanças no escopo ou correções feitas por parte dos parceiros responsáveis pelas correções a equipe de teste avaliara mudanças no planejamento e posteriormente a necessidade de criação de novos testes. Durante todas as etapas serão apresentados os recursos que podem auxiliar na mitigação dos impactos causados pelo DDS.

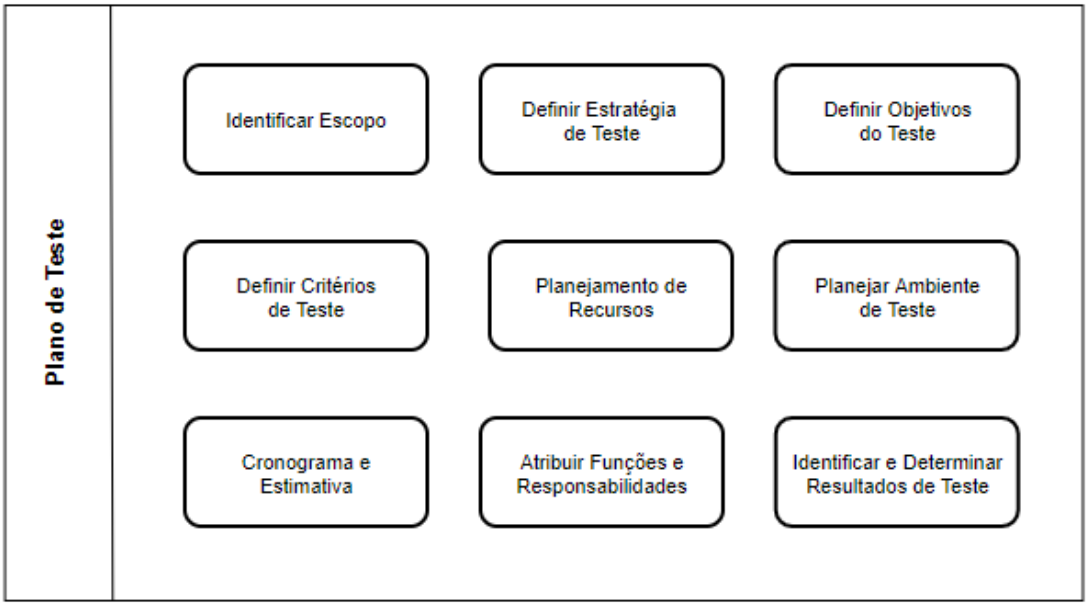
3.1 Planejamento

Antes de dar início às atividades de teste, é necessário um entendimento dos requisitos, objetivos e estrutura do projeto. Sendo assim, a equipe de teste é responsável por criar o plano de teste que fornecerá a abordagem estrutural de como testar o software para alcançar os resultados esperados.

Durante a etapa de planejamento é fundamental o entendimento de como os desafios do DDS podem impactar no desenvolvimento do software, de modo que o plano de teste elaborado contribua para garantir a qualidade do produto. Alguns desafios e lições podem ser levados em consideração com vista a minimizar os impactos do DDS nas equipes de teste como: definição de meios de comunicações entre os membros do projeto e equipe de teste; disponibilidade de requisitos e critérios de aceitação para equipe de teste; uso de testes automatizados; escolha de ferramentas que deem suporte a atividades colaborativas e, principalmente, a organização durante todo o processo de teste. (COLLINS et al., 2012).

A seguir são descritas as informações que devem estar contidas no plano de teste, de acordo com a Figura 13, a seguir:

Figura 13 - Tarefas do plano de teste



Fonte: Adaptado de https://www.guru99.com/what-everybody-ought-to-know-about-test-planing.html

Identificar escopo: a partir dos requisitos do software será estabelecido o que deve ser testado, de forma que a equipe saiba os caminhos que precisam ser cobertos durante a execução dos testes.

Definir estratégia de teste: determinar como os testes precisam ser realizados e que tipos de testes serão realizados.

Definir objetivos do teste: identificar quais os objetivos esperados da execução dos testes, para garantir que o software liberado esteja livre de erros.

Planejar ambiente de teste: configurar softwares, hardware e rede necessária para testar o software.

Planejamento de recursos: determinar os recursos humanos, de sistema e equipamento necessários para o teste.

Definir critérios de teste: especificar os critérios que determinam que a fase de teste estará concluída.

Cronograma e estimativa: estimar e dividir a fase de teste em pequenas tarefas.

Atribuir funções e responsabilidades: definir as funções e responsabilidades dos membros da equipe de teste.

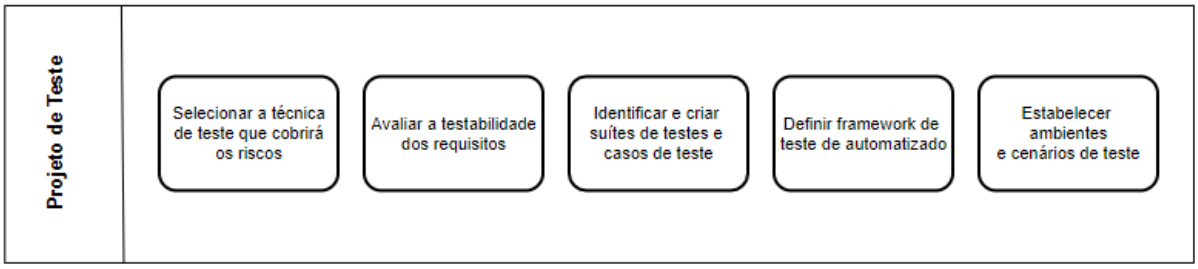
Identificar e determinar resultados de teste: listar todos os documentos que devem ser desenvolvidos e mantidos durante a fase de teste.

Durante o processo de validação é necessário ir comparando os resultados obtidos das atividades de teste com a estratégia definida no Plano de Teste. Caso sejam identificadas mudanças no escopo serão realizadas adaptações no planejamento, garantindo o controle de teste.

3.2 Projeto

Uma vez identificadas e estabelecidas as estratégias e os objetivos para validação do software são necessários descrever como o teste deve ser realizado e quantos são necessários. As atividades estabelecidas durante o projeto de teste são apontadas na figura 14 abaixo:

Figura 14 - Tarefas do projeto de teste



Fonte: Elaborada pelo autor (2020).

Selecionar a técnica de teste que cobrirá os riscos: determinar as técnicas de teste que mitigam o risco de erros no software lançado.

Avaliar a testabilidade dos requisitos: analisar todos os requisitos e casos de uso para identificar a testabilidade.

Identificar e criar suítes de testes e casos de teste: estudar os requisitos e casos de uso para definir as suítes de teste e elaborar casos de teste.

Definir framework de teste de automatizado: estabelecer o melhor framework de automação de teste e criar scripts.

Estabelecer ambientes e cenários de teste: preparar o ambiente e infraestrutura necessária para a execução dos casos de teste.

Como citado anteriormente, o uso de ferramentas de suporte colaborativas pode auxiliar na organização da equipe de teste e também minimizar os impactos do DDS. Deste modo, durante a etapa de projeto podem ser utilizadas ferramentas de criação de mapas mentais com o MindMeister[3] para auxiliar na definição de suítes de teste e casos de teste e de forma complementar ferramentas de gerenciamento de teste, como Testlink, para que posteriormente sejam criados planos de execuções dos casos de teste cadastrados e geração de relatórios para acompanhamento dos resultados do processo de teste.

Como apontado por Collins et al. (2012), o uso de testes automatizados pode reduzir os impactos do DDS, sendo assim, durante a fase de projeto é importante que, quando possível, sejam utilizados frameworks de testes automatizados para auxiliar nos testes do software.

[3] www.mindmeister.com

3.3 Execução

A execução de teste é uma das fases mais importantes do ciclo de vida do teste. Nesta fase, a equipe de teste começa a executar os casos de teste preparados na fase de projeto. Em estratégias de testes manuais, o testador irá seguir de forma manual o passo a passo descrito no caso de teste. Em estratégias de testes automatizados, o testador será responsável por dar início a execução dos scripts de teste criados na fase anterior. Durante a execução, os resultados obtidos são comparados aos resultados esperados.

Um dos principais desafios durante a execução de testes é a configuração do ambiente de teste, já que este deve ser o mais parecido possível ao ambiente de produção. Os problemas com o DDS podem estar relacionados a:

> [...] falta ou diferenças de infraestrutura em diferentes locais de desenvolvimento, incluindo conectividade de rede, ambiente de desenvolvimento, laboratórios de teste e *build*, e sistemas de gerenciamento de mudança e versão (MOCKUS e HERBSLEB, p. 182, 2001, tradução nossa).

De forma a mitigar estes problemas o uso de ferramentas como o Docker permite que sejam criados *containers* com as mesmas configurações, fornecendo um ambiente de teste adaptável as necessidades dos membros da equipe de teste.

3.4 Resultados

Os resultados obtidos na execução dos testes devem ser relatados incluindo informações sobre a versão do software em teste, ferramentas utilizadas e evidências. Sendo fundamental o uso das ferramentas de gerenciamento de testes, como o Testlink, para que resultados dos testes estejam disponíveis para todos os envolvidos no projeto e relatórios possam ser gerados. É importante que inconsistências também sejam documentadas para posterior análise da causa.

Com os resultados obtidos, a equipe de teste identificará se testes adicionais são necessários ou se os critérios de saída especificados devem ser alterados, sendo assim, os resultados dos testes podem influenciar diretamente em alterações no plano de teste caso seja observado que a cobertura dos testes não é suficiente para garantir a qualidade do software.

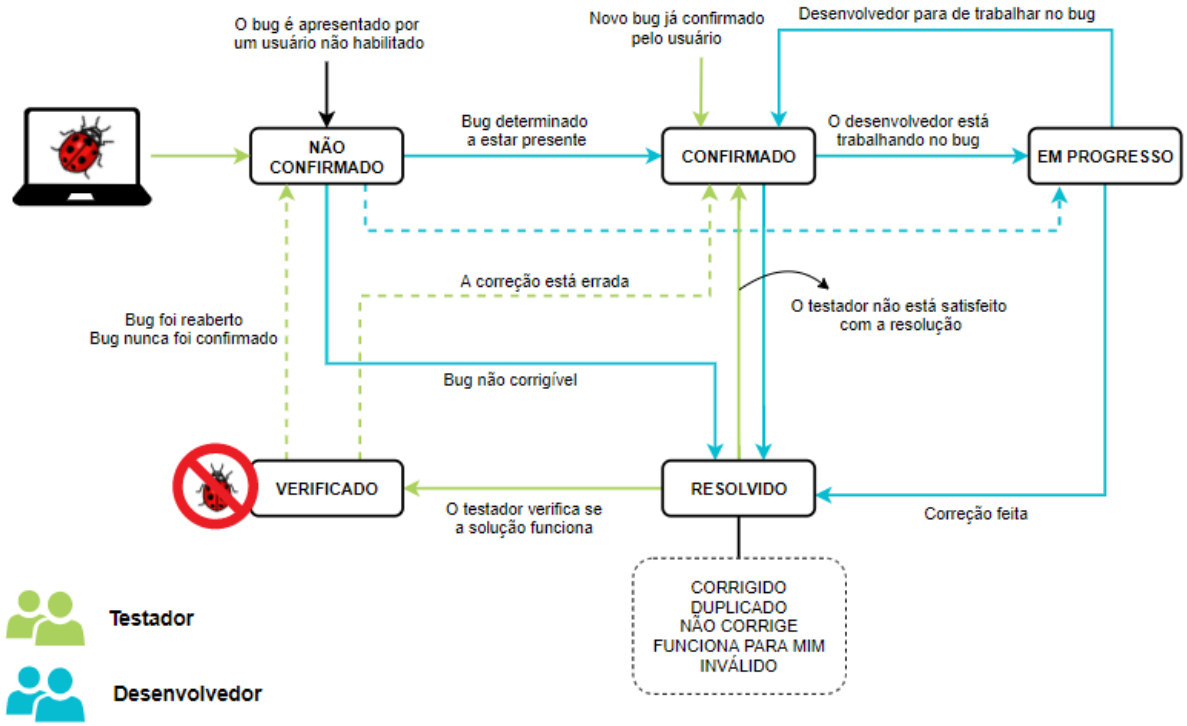
3.5 Gerenciamento de Bugs

Após análise das inconsistências encontradas nos resultados das execuções dos testes, bugs serão cadastrados contendo informações que permitam aos desenvolvedores reproduzir os erros e que a equipe de teste possa rastrear os erros até o fechamento. Nos projetos com DDS, o uso de ferramentas como Bugzilla auxilia que estas informações sejam disponibilizadas para a equipes dispersas e fornecem um meio de comunicação entre testadores e desenvolvedores.

Após as correções dos bugs, a equipe de teste é responsável por executar as atividades de reteste para verificar se a correção do bug foi válida. O gerenciamento de bugs ajuda a equipe de teste identificar se os erros afetam outras áreas do sistema e se existe a necessidade de testes de regressão.

Como citado no capítulo anterior, durante o ciclo de vida do bug o seu status irá mudando de acordo com sua identificação até sua resolução. É necessário que o ciclo de vida do bug estabelecido para o gerenciamento dos bugs seja de conhecimento dos desenvolvedores e testadores, principalmente quando há dispersão geográfica das equipes. A Figura 15, a seguir, apresenta o ciclo de vida do bug cadastrado no Bugzilla.

Figura 15 - Ciclo de vida do bug



Fonte: Adaptado de https://www.bugzilla.org/docs/2.18/html/lifecycle.html

Uma vez alterado o status do bug para resolvido por parte do desenvolvedor é necessário identificar um dos possíveis tipos de resoluções:

Corrigido: o bug foi corrigido e o repositório atualizado.

Duplicado: utilizado quando o mesmo problema já foi relatado anteriormente para outro bug.

Não corrige: pode ser um bug, mas não será corrigido por algum motivo (o motivo deverá ser detalhado em um comentário).

Funciona para mim: para o desenvolvedor o bug não existe ou não pode ser reproduzido usando as informações fornecidas.

Inválido: usado quando o problema identificado não é um bug ou a alteração para corrigi-lo está fora do poder dos desenvolvedores.

3.6 Considerações Finais

Neste capítulo foi apresentado o processo de teste proposto para validação funcional para projetos cujo modelo de desenvolvimento seguido é o desenvolvimento distribuído de software. O objetivo do processo proposto é definir um conjunto de atividades e documentações, que devem ser criados durante as fases do processo e propor ferramentas que podem minimizar os impactos do DDS nas fases de teste.

Os resultados obtidos ao longo das fases devem ser avaliados para que possíveis atualizações no plano de teste sejam realizadas de forma a garantir a qualidade mesmo com mudanças em escopo. O uso de um processo único por todos os membros da equipe permite a diminuição dos problemas ocasionados pelo DDS.

4 APLICAÇÃO DO PROCESSO DE TESTE NO OCARIoT

Este capítulo descreve o estudo de caso da aplicação do processo de teste, proposto no capítulo anterior, no projeto OCARIoT (Smart Childhood Obesity CARing Solution Using IoT Potencial) e os resultados obtidos. Assim como apresentado no capítulo anterior, os resultados obtidos da aplicação do processo foram documentados no **Deliverable 5.12 OCARIoT Functional Validation v1.**

O OCARIoT tem por objetivo o enfrentamento contra a obesidade infantil, a partir do rastreamento de alimentos, atividades físicas, motivação para hábitos saudáveis e atividades através da gamificação, baseado em IoT (Internet das coisas, do inglês *Internet of Things*), como exibido na Figura 16 a seguir:

Figura 16 - Funcionamento do OCARIoT

Fonte: OCARIoT (2018).

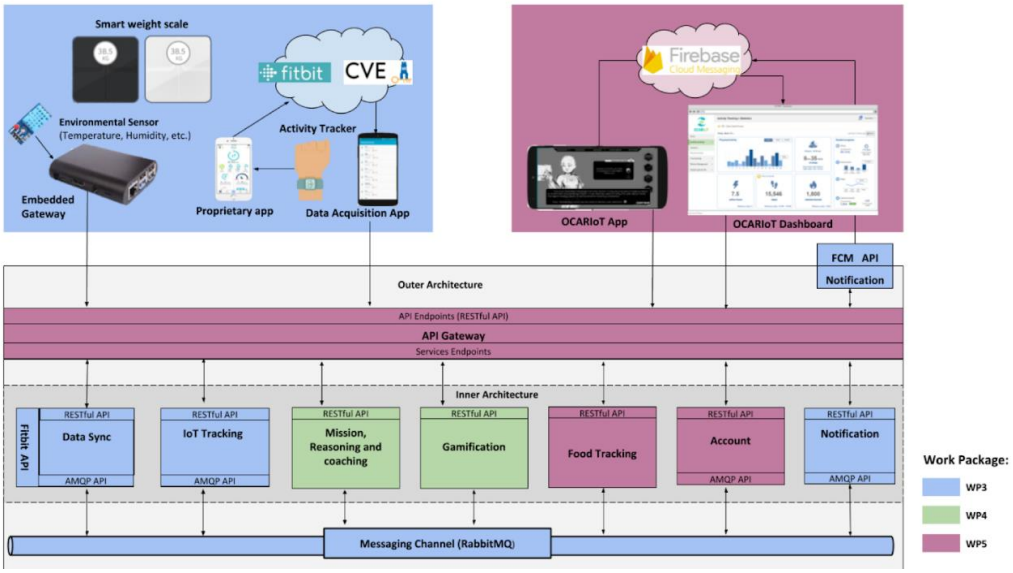
O projeto é “co-financiado pelo Programa HORIZON 2020 da União Européia e pelo Ministério da Ciência, Tecnologia e Inovação do Brasil através da Rede Nacional de Ensino e Pesquisa (RNP)” (OCARIoT, 2018), desenvolvido em parceria com instituições brasileiras e de países da Europa como Espanha, Grécia e Portugal, como apresentado na Figura 17. Por ser desenvolvido em parceria com diferentes instituições, nas quais as equipes estão dispersas, o processo adotado na realização do projeto é o DDS.

Figura 17 - Parcerias do OCARIoT

Fonte: https://ocariot.eu/partners.

A arquitetura do OCARIoT, apresentada na Figura 18, é composta por um conjunto de microsserviços, no qual a API Gateway, será responsável por receber e enviar dados através de requisições para microsserviços externos como o OCARIoT Dashboard e Data Acquisition App.

Figura 18 - Arquitetura do OCARIoT



Fonte: OCARIOT (2020).

O processo de teste proposto foi aplicado de forma separada no OCARIoT Dashboard, API Gateway e Data Acquisition App, onde foram detalhados o funcionamento de cada microsserviço e as ferramentas que auxiliaram cada etapa do processo de teste. Como apresentado no capítulo anterior o processo é composto pelas seguintes fases:

Planejamento: durante esta fase é realizada a análise do escopo do projeto e seus requisitos para que possa ser elaborado o plano de teste, que fornecerá uma abordagem estrutural de como testar o software e recursos necessários.

Projeto: a partir dos requisitos são elaborados os casos de teste, que fornecerão as instruções necessárias para realização dos testes. Caso seja optado pelo uso de testes automatizados deverão ser implementados os scripts.

Execução: serão executados os casos de teste elaborados na fase anterior a partir de estratégias de testes manuais ou automatizados.

Resultado: a partir da execução dos testes os resultados obtidos são documentados.

Gerenciamento de Bugs: os bugs identificados durante a execução são reportados de forma que a equipe de desenvolvimento possa ter acesso as suas informações para posteriores correções.

4.1 OCARIoT Dashboard

O OCARIoT Dashboard fornece um painel de visualização para familiares, educadores e profissionais de saúde para monitorar o desempenho das crianças, como apresentado na Figura 19.

Figura 19 - OCARIoT Dashboard

Fonte: OCARIOT (2020).

O painel é dividido em seis funcionalidades diferentes:

Acompanhamento (*Follow-up*): Informações sobre o progresso da criança nos jogos, alimentação e atividade física.

Missões (*Missions*): Lista as missões na quais a criança deve atuar como um agente secreto. Fornece informações do progresso das respostas às missões e suas estatísticas.

Dispositivos IoT (*IoT Devices*): Para rastreabilidade dos dados de dispositivos sincronizados.

Consumo de alimentos (*Food Intake*): Coleta e acompanha os hábitos de nutrição infantil.

Atividade Física (*Physical Activity*): Coleta e monitora as atividades, exercícios e qualidade do sono das crianças.

Perfil (*Profile*): Fornece dados de perfil da criança para estudo.

4.1.1 *Planejamento*

Durante a etapa de planejamento do processo de teste para validação funcional do Dashboard foi necessário a criação do plano de teste, definido abaixo:

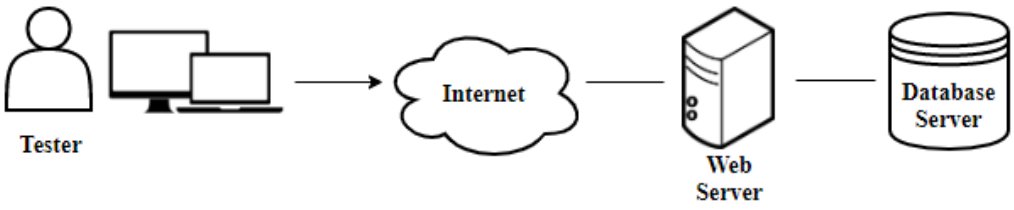
Escopo: Todos os recursos do Dashboard definidos na especificação de requisitos disponíveis no documento D5.2 OCARIoT Deliverable: OCARIoT app and OCARIoT dashboard prototypes design v2.

Estratégia de teste: Testes de integração e de teste de aceitação devem ser realizados durante a validação funcional do painel para garantir a conformidade do sistema com os requisitos.

Objetivos do teste: O objetivo é verificar se todas as funcionalidades estão funcionando conforme o esperado, sem erros ou bugs, antes de serem implementadas em um ambiente de produção.

Ambiente de teste: O ambiente de teste necessário para testar o OCARIoT Dashboard é definido de acordo com a figura 20 abaixo.

Figura 20 - Ambiente de teste do OCARIoT Dashboard



Fonte: OCARIOT (2020).

Planejamento de recursos: São necessários que os recursos do sistema e de equipamentos satisfaçam o ambiente de teste apresentado na Figura 20. Foram escolhidos como ferramentas de teste o Testlink e Bugzilla e dois testadores responsáveis para realização das atividades de teste.

Critérios de Teste: Como critérios de saída para garantir que a fase de teste seja bem-sucedida é necessário que a taxa de execução de testes seja obrigatória em 100%, a menos que seja indicado um motivo claro e a taxa de aprovação seja 90% sem a presença de bugs de alta criticidade.

Cronograma e estimativa: Primeira versão do Dashboard - fevereiro a agosto de 2019. Segunda versão do Dashboard - agosto a dezembro de 2019. Especificação, execução dos testes, relatórios de teste e entrega para ambas as versões.

Funções e Responsabilidades: Um testador será responsável por definir o Plano de Teste e juntamente com um segundo testador será responsável pela criação e execução dos casos de teste, além de relatar bugs.

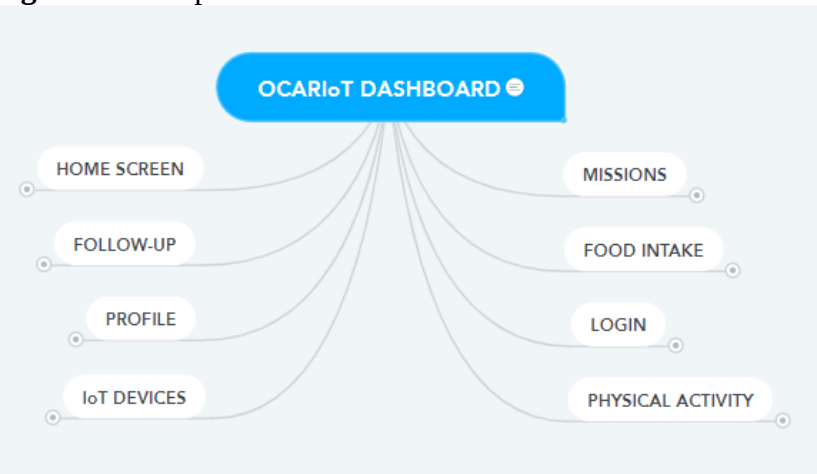
Resultados do teste: Antes dos testes - plano de teste, casos de teste e especificação do projeto de teste. Durante os testes - scripts de teste, dados de teste e log de erros e de execução. Após a fase de teste - Relatórios de teste e *Release*.

4.1.2 *Projeto*

O projeto de teste descreve como as atividades de teste devem ser realizadas, incluindo informações sobre quantos testes precisarão ser executados e condições. Para satisfazer o teste de aceitação é importante escolher a melhor técnica para cobrir os riscos e testar todos os requisitos do Dashboard. O teste funcional considera o comportamento do software, portanto, técnicas de caixa preta podem ser usadas para derivar condições de teste e casos de teste para a funcionalidade do componente ou sistema.

O Dashboard foi dividido em pequenas seções, de acordo com seus requisitos funcionais e organizado em um mapa mental, conforme Figura 21. Cada seção se tornou uma suíte de testes para os casos de teste. A suíte de teste é um contêiner que categoriza um conjunto de testes.

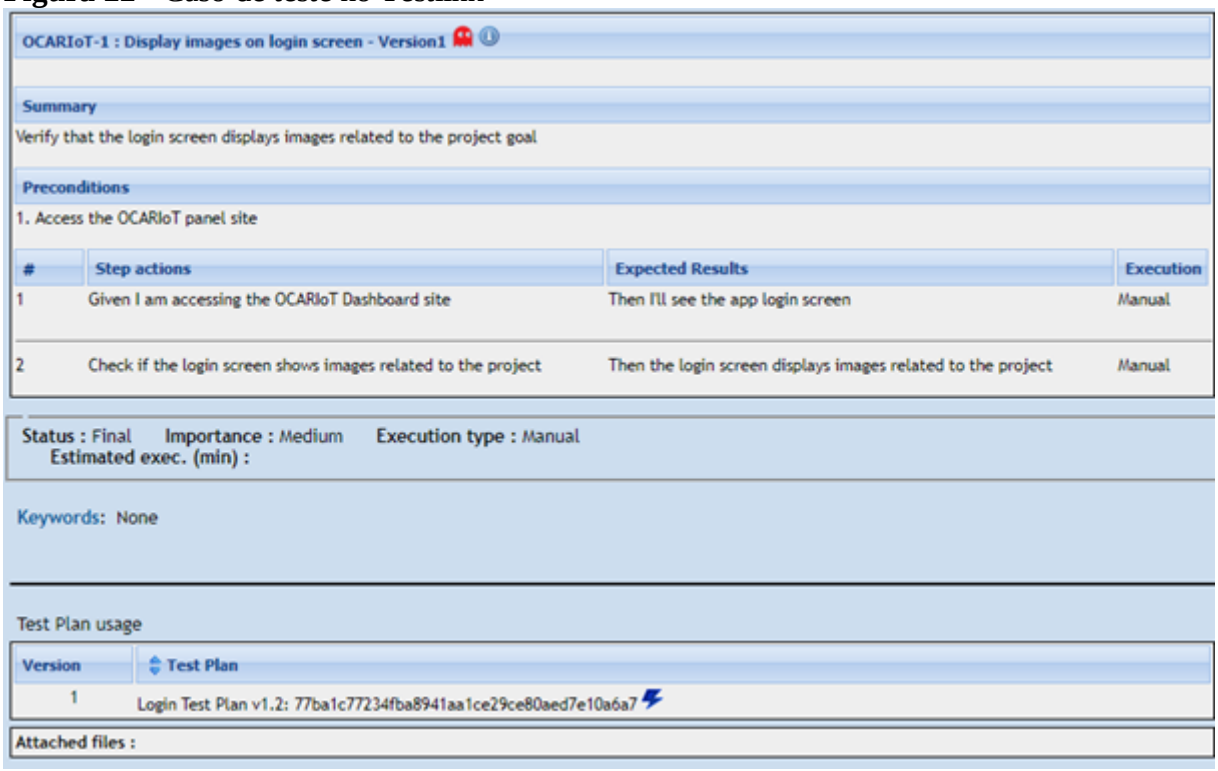
Figura 21 - Mapa mental do OCARIoT Dashboard



Fonte: Adaptado de OCARIOT (2020).

Uma vez especificadas as Suítes de Teste, é necessário definir os casos de teste. O Caso de Teste fornecerá instruções para garantir que o requisito seja válido. Alguns requisitos requerem mais de um caso de teste. Todos os casos de teste foram adicionados à plataforma Testlink, conforme Figura 22:

Figura 22 - Caso de teste no Testlink



Fonte: OCARIOT (2020).

Para a primeira versão do painel, v1.0.0, disponível para a equipe de teste, alguns testes foram automatizados. O Selenium foi estabelecido como framework de automação para os casos de teste, que a equipe identificou como um bom candidato à automação. Para a segunda versão disponível, v2.0.0, foram criados casos de teste para novas funcionalidades e foram observados que alguns dos testes criados para a v1.0.0 estavam desatualizados e precisavam ser atualizados ou removidos para a realização das novas execuções.

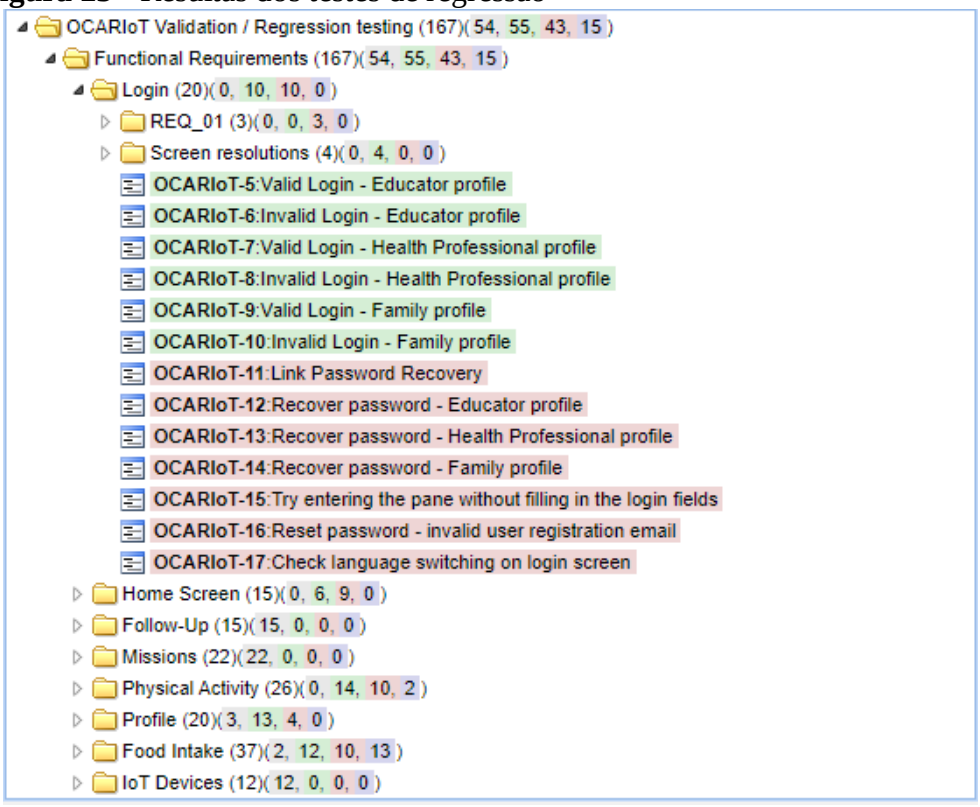
4.1.3 *Execução*

Para organizar as execuções das diferentes versões dos testes escritos, foi necessário a criação de diferentes planos de teste no Testlink. Em cada plano de teste foram adicionados os

casos de teste que farão parte da nova execução, desta forma é possível ter um histórico de execuções.

Para a v1.0.0, os casos de teste associados à Tela inicial (Home Screen) e ao Login foram realizados executados manualmente e através de testes automatizados. Uma vez disponível o v2.0.0, um novo plano de teste foi gerado com todos os casos de teste, que haviam sido criados para a versão anterior para realização de testes de regressão, como mostra a figura 23.

Figura 23 - Resultas dos testes de regressão



Fonte: OCARIOT (2020).

A partir dos resultados obtidos da execução do teste de regressão, a equipe de teste observou que alguns casos de teste estavam desatualizados e precisavam ser atualizados ou removidos. Desta forma, foi criado um novo plano de teste, o Dashboard v2, com os novos testes e atualizações para a v2.0.0.

4.1.4 *Resultados*

Após as execuções dos testes das funcionalidades de Login e Tela Inicial (Home Screen) da v1.0.0 foram obtidos os seguintes resultados apresentados na Tabela 1, a seguir:

Tabela 1 - Resultados da execução dos testes do Dashboard v1.0.0

Suíte de Teste	Testes Especificados	Passando	Falhando	Bloqueado
Login	20	55.00%	20.00%	25.00%
Home Screen	15	73.33%	6.67%	20.00%
Resultados	**35**	**66.00%**	**12.00%**	**22.00%**

Fonte: Adaptado de OCARIOT (2020).

Na v2.0.0 foram executados os testes referentes a novas funcionalidades e dos testes atualizados que haviam sido criados durante a fase de projeto da v1.0.0, os resultados obtidos são detalhados na Tabela 2 abaixo:

Tabela 2 - Resultados da execução dos testes de regressão do Dashboard v2.0.0

Suíte de Teste	Testes Especificados	Passando	Falhando	Bugs Reportados	Corrigido	Bloqueado
Login	20	10	10	10	0	0
Home Screen	15	6	9	9	0	0
Follow-up	0	0	0	0	0	0
Missions	0	0	0	0	0	0
Physical Activity	26	14	10	10	0	2
Profile	17	13	4	4	0	0
Food Intake	35	12	10	10	0	13
Iot Devices	0	0	0	0	0	0
Total	**113**	**55**	**43**	**43**	**0**	**15**
Resultados		**48.6%**	**38.00%**	**100.00%**	**0.00%**	**13.2%**

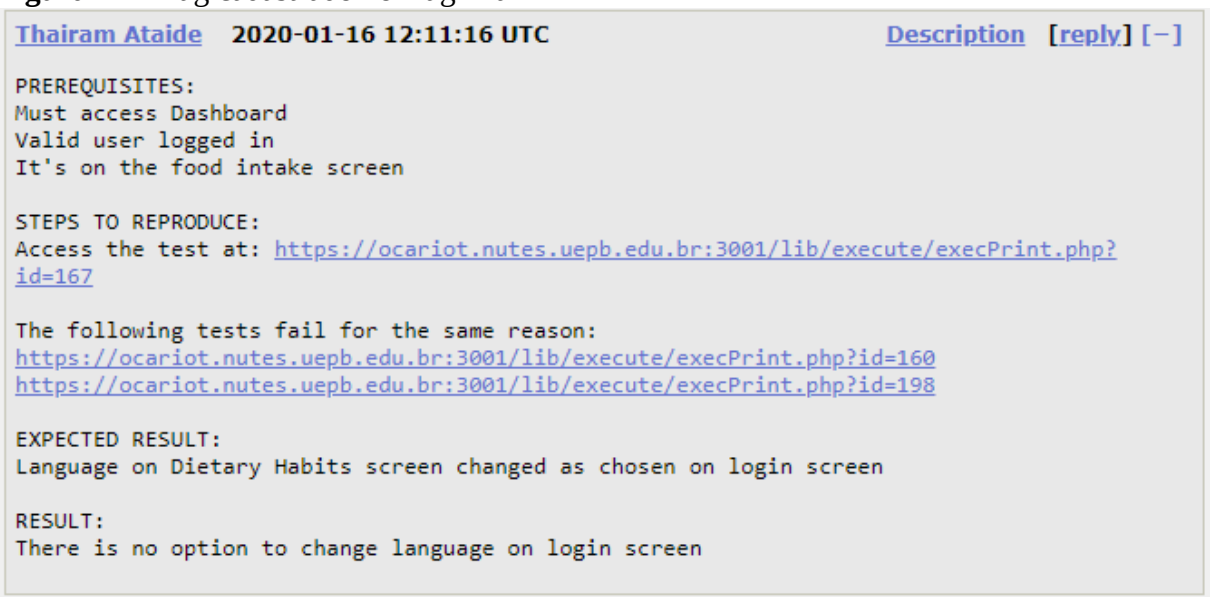
Fonte: Adaptado de OCARIOT (2020).

É importante destacar que 13,2% dos testes estão bloqueados porque algumas funcionalidades não foram totalmente implementadas até o momento. A suíte de teste será executada novamente para as próximas versões do Dashboard.

4.1.5 *Gerenciamento de Bugs*

Todos os bugs identificados após a execuções dos testes de regressão da v2.0.0 foram relatados no Bugzilla. Na Tabela 2 é possível identificar o número de bugs encontrados durante a execução dos testes e o número de bugs que foram corrigidos. No cadastro do bug são identificados caso de teste que falhou, pré-requisitos, passa a passo para reprodução, evidências, resultado esperado e os resultados obtidos, como apresentado na Figura 24.

Figura 24 - Bug cadastrado no Bugzilla



Fonte: OCARIOT (2020).

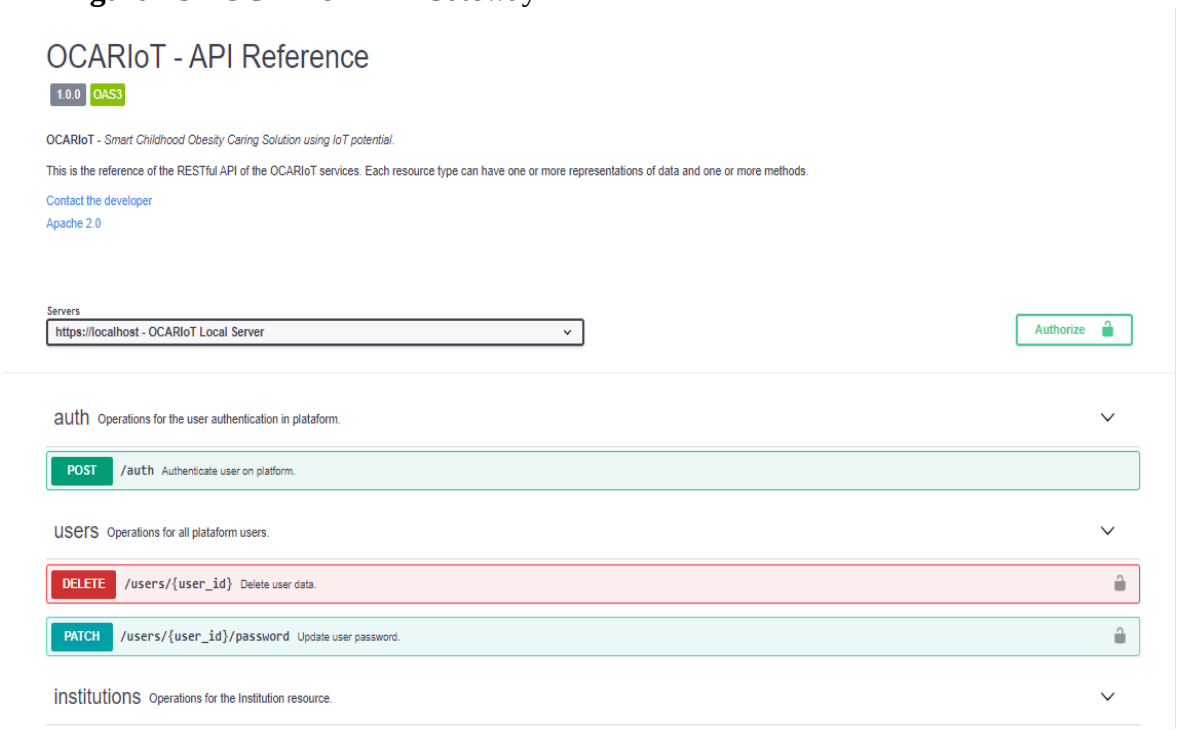
4.2 API Gateway

O OCARIoT API Gateway é uma API RESTful, que permite a integração de plataformas IoT por meio de microsserviços e é o ponto de entrada único de comunicação entre clientes OCARIoT, entre estes o OCARIoT Dashboard e o Data Acquisition App, e os serviços OCARIoT. O API Gateway reduz o acoplamento e a complexidade de clientes e microsserviço, aumentando a segurança e reduzindo a superfície de ataque.

O OCARIoT API Gateway foi projetado e documentado usando o Swagger[4], como apresentado na Figura 25, por ser uma solução de código aberto.

[4] https://swagger.io/about/

Figura 25 - OCARIoT API Gateway



Fonte: OCARIOT (2020).

4.2.1 *Planejamento*

Durante o planejamento do processo de teste para validação da API Gateway foi criado o plano de teste, definido abaixo:

Escopo: todas as funções da API disponíveis em https://api.ocariot.lst.tfo.upm.es/v1/reference/

Estratégia de teste: o teste de integração e o teste do sistema devem ser realizados durante a validação funcional do API Gateway para garantir a conformidade com os parâmetros, respostas e corpo de solicitação da API.

Objetivos do teste: O objetivo é verificar se todas as funções documentadas na API estão funcionando conforme o esperado, sem erros ou bugs, antes de serem implantadas em um ambiente de produção.

Ambiente de teste: Todos os scripts foram criados utilizando Node.js, o framework de teste Mocha e a biblioteca Chai. Os scripts foram implementados em um Ambiente de Desenvolvimento Integrado (IDE) sendo executados nas máquinas dos testadores envolvidos.

Planejamento de recursos: São necessários que os recursos do sistema e de equipamentos satisfaçam as informações especificadas para o ambiente de teste. Como

ferramenta de gerenciamento de bugs será utilizado o Bugzilla. Dois testadores serão responsáveis pela realização das atividades de teste.

Critérios de teste: 100% de cobertura dos *endpoints* da API e a taxa de aprovação de 90%, sem a presença de bugs críticos.

Cronograma e estimativa: Primeira versão - fevereiro a agosto de 2019. Segunda versão - agosto a dezembro de 2019. Especificação, execução dos testes, relatórios de teste e entrega para ambas as versões.

Funções e responsabilidades: Um testador será responsável por definir o plano de teste e, ao lado de um segundo testador, registrar e executar casos de teste e relatar bugs.

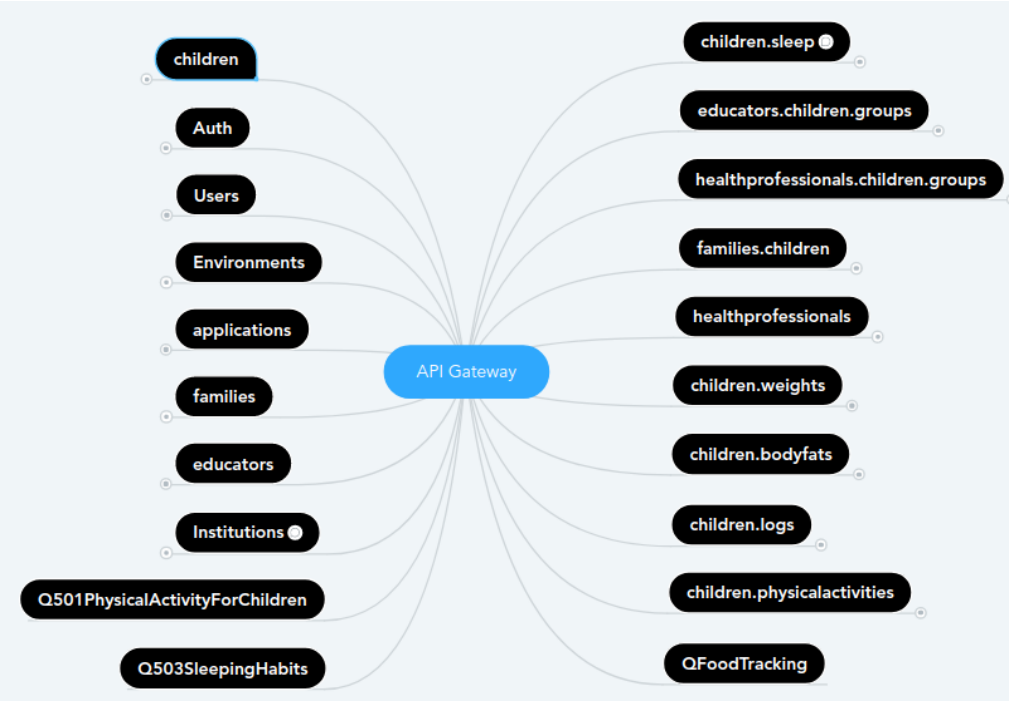
Resultados do teste: antes do teste - Plano de teste, casos de teste e especificação de design de teste. Durante o teste - scripts de teste, dados de teste e log de erros e logs de execução. Após a fase de teste - Relatórios de teste e *Release*.

4.2.2 *Projeto*

Conforme executado nos testes do Dashboard, o teste da caixa preta também será usado para testar se os métodos da API estão corretos. Todos os métodos disponíveis na API funcionarão como casos de teste, uma vez que se estabeleça as entradas e saídas esperadas do teste. Portanto, para cada *endpoint* fornecido pela API, casos de teste personalizados são especificados para verificar a funcionalidade de cada recurso.

A Figura 26 mostra a visão geral do mapeamento de alguns recursos da API. Cada recurso fornece os *endpoints* de acordo com as regras de negócio documentadas. A partir do *endpoint* são gerados os casos de teste necessários para a validação do mesmo.

Figura 26 - Mapa mental do OCARIoT API Gateway



Fonte: OCARIOT (2020).

A automação dos testes é feita de forma personalizada, com a utilização de ambientes e bibliotecas que auxiliam no processo de geração dos scripts, como Node.js, Mocha e Chai. Primeiro foram especificados os casos de teste no MindMeister (farreamenta de criação de mapa mental) e, posteriormente, a criação dos scripts para que a partir deste ponto, os testes possam ser executados automaticamente.

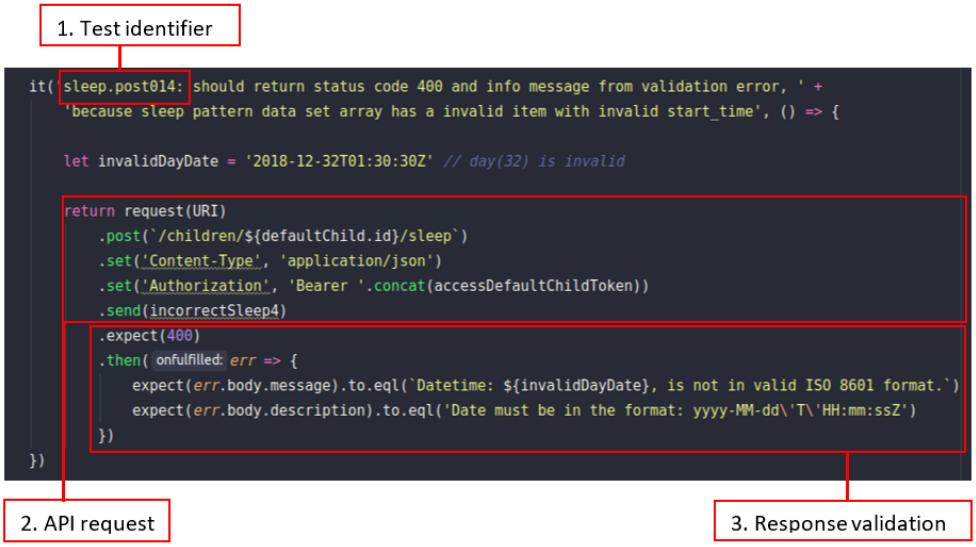
4.2.3 *Execução*

Os testes são realizados utilizando alguns recursos do Mocha e da biblioteca Chai no ambiente Node.js, permitindo ao testador, por meio de scripts, simular solicitações diretamente na API, bem como as validações necessárias para as respostas obtidas por aquelas solicitações.

Cada teste possui um identificador composto por <resource.verb_http + ID de três dígitos>, o qual é criado manualmente visando controlar a execução dos testes, permitindo que, através de scripts, os testes possam ser realizados individualmente ou em grupos

personalizados. A Figura 27 mostra o corpo de um caso de teste específico para o recurso *sleep*, onde é possível identificar a solicitação à API e a validação da resposta obtida.

Figura 27 - Script de teste do recurso *sleep*



Fonte: OCARIOT (2020).

4.2.4 *Resultados*

A versão inicial V1.0.0 do API Gateway serviu como ponto de partida para os testes. Por se tratar de uma versão inicial foi decidido pelas equipes envolvidas no projeto, que os bugs encontrados não seriam reportados nesta versão, mas na próxima, pois muitos dos *endpoints* disponíveis seriam modificados e outros novos inseridos.

A Tabela 3 mostra os resultados da execução de testes automatizados para a versão inicial da API, onde cada linha da tabela corresponde aos testes realizados para um determinado microsserviço.

Tabela 3 - Resultados da execução dos testes da API Gateway v1.0.0

Suíte de Teste	Testes Especificados	Passando	Falhando	Bugs Reportados	Corrigido
Account	611	474	137	0	0
IoT Tracking	265	199	66	0	0
Total	**876**	**673**	**203**	**0**	**0**
Resultados		**77%**	**23%**	**0%**	**0%**

Fonte: Adaptado de OCARIOT (2020).

A Tabela 4 mostra os resultados da execução dos testes para a nova versão da API, V1.2.1.

Tabela 4 - Resultados da execução dos testes da API Gateway v1.2.1

Suíte de Teste	Testes Especificados	Passando	Falhando	Bugs Reportados	Corrigido
Account	649	575	74	74	22
IoT Tracking	453	375	78	78	69
Questionary	192	124	68	0	0
Food	0	0	0	0	0
Total	**1294**	**1074**	**220**	**152**	**91**
Resultados		**83%**	**17%**	**69%**	**41%**

Fonte: Adaptado de OCARIOT (2020).

Embora o número de casos de teste tenha aumentado consideravelmente da v1.0.0 para a V1.2.1, a proporção do número de testes que foram aprovados também aumentou, enquanto que o número de testes que falharam diminuiu, o que indica que a API está evoluindo em qualidade e atendendo às especificações, mostrando a importância do relatório de erros e do processo de correção para esses resultados. Cerca de 65% dos recursos fornecidos atualmente pela API foram testados.

4.2.5 *Gerenciamento de Bugs*

Ao reportar os bugs identificados algumas informações relevantes devem ser relatadas, como uma lista de membros responsáveis por gerenciar e corrigir os erros de forma que acompanhem todo o ciclo de vida do bug, sendo automaticamente notificados pelo Bugzilla através de um e-mail previamente cadastrado para os membros.

Imediatamente após o cadastro do bug, o Bugzilla envia um e-mail para todos os envolvidos no processo de correção que irão realizar a análise e correção do erro. A Figura 28 mostra a descrição de uma correção de bug feita por um dos membros responsáveis.

Figura 28 - Resposta de correção do bug

Jefferson Medeiros **2019-10-30 18:05:27 UTC** Comment 1

Bug fixed in commit: https://github.com/ocariot/iot-tracking/commit/af84dfa29668b773bffe653087fb0e9307b13dc9

Reason for bug existence:

- The scenario in which the date day is invalid is not validated in the default date validator.

Actions required for correction:

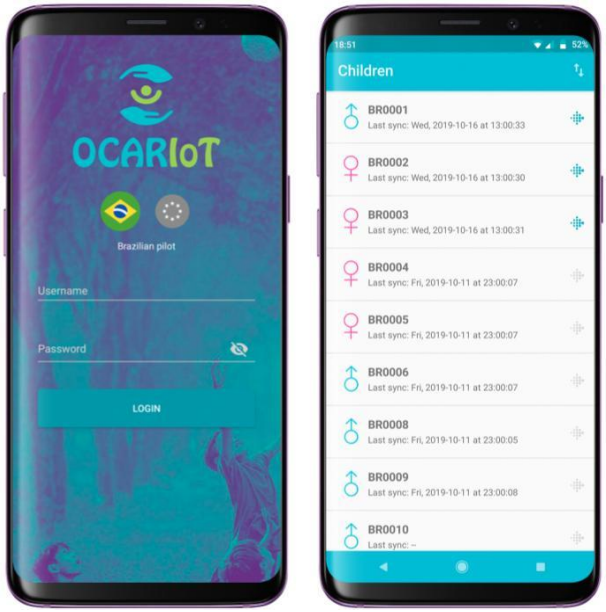
- After calling the default date validator, was added to check if a 'new Date()' call in the received value generates an invalid date, and if so, an exception is thrown with the same message pattern as already defined.

Fonte: OCARIOT (2020).

4.3 Data Acquisition App

O OCARIoT Data Acquisition App proporciona ao usuário os dados fornecidos por meio do OCARIoT Dashboard após a sincronização de dados. O aplicativo permite aos usuários acessar uma lista de crianças, como apresentado na Figura 29, com seus dados referentes a atividade física, sono, peso e frequência cardíaca. Quando uma criança tem seu Fitbit vinculado ao app, é possível enviar solicitações onde todos os dados serão exibidos.

Figura 29 - OCARIoT Data Acquisition App



Fonte: OCARIOT (2020).

Somente educadores e profissionais de saúde podem disponibilizar o acesso direto no aplicativo. A principal funcionalidade do aplicativo é a sincronização com dispositivos que utilizam a tecnologia IoT, permitindo que informações sobre peso e frequência cardíaca sejam sincronizados diretamente com o aplicativo. Além disso, o app fornece informações sobre:

Atividade física:
- A duração de uma atividade;
- Número de passos e distância percorrida;
- Despesa calórica e a quantidade por minuto;
- E os níveis de atividade.

Sono:
- Detalhes do sono;
- Monitoramento da eficiência do sono;
- O período de sono e tempo;
- Monitorar os detalhes do sono, como o período em que dorme profundamente, tem um sono leve, adormece, acorda durante o sono, momento agitado e sono REM.

A tela de atividade física fornece uma lista de todas as atividades que foram realizadas e fornecidas ao painel. Após sincronizar os dados, é possível visualizar os detalhes de cada um, o mesmo acontece com o sono, peso e frequência cardíaca. Os dados de peso estão sempre visíveis para o usuário e os de frequência cardíaca somente quando sincronizados com o dispositivo.

4.3.1 *Planejamento*

O plano de teste definido na etapa de planejamento para validação Data Acquisition App é apresentado abaixo:

Escopo: Todos os recursos do aplicativo disponíveis no manual.

Estratégia de teste: Teste de aceitação e de integração através de testes caixa preta.

Objetivos do teste: Verificar se todos os recursos documentados estão funcionando conforme o esperado, sem bugs, antes de serem implantados em produção;

Ambiente de teste: Todos os scripts foram criados utilizando a ferramenta Appium foram implementados em um ambiente de desenvolvimento integrado (IDE) rodando em um dispositivo móvel e nas máquinas dos testadores envolvidos;

Planejamento de recursos: A ferramenta Appium foi escolhida para automação dos testes e o Android Studio, como ambiente de desenvolvimento integrado para os testes garantindo a conformidade com os requisitos. Um testador foi responsável pelos testes.

Critérios de teste: O teste será considerado concluído quando 100% forem executados com uma taxa de aprovação de 90% sem a presença de bugs críticos.

Cronograma e estimativa: Fevereiro a agosto de 2019. Especificação, execução dos testes, relatórios de teste e entrega.

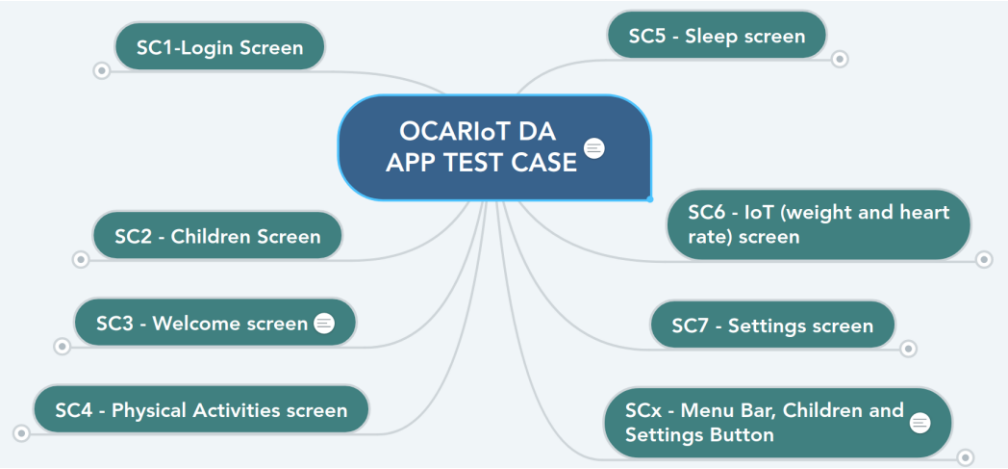
Funções e responsabilidades: Um testador foi responsável pela documentação do plano de teste, criação e execução dos casos de teste;

Resultados do teste: Antes do teste - plano de teste, casos de teste e especificação de design de teste. Durante o teste - scripts de teste, dados de teste e logs de erro e de execução. Após a fase de teste - teste e *release*.

4.3.2 *Projeto*

O Data Acquisition App foi dividido em seções de acordo com as telas do aplicativo, cada seção passou a ser uma suíte de testes, onde serão especificados os casos de teste, conforme se observa na Figura 30 a seguir:

Figura 30 - Mapa mental do OCARIoT Data Acquisition App



Fonte: OCARIOT (2020).

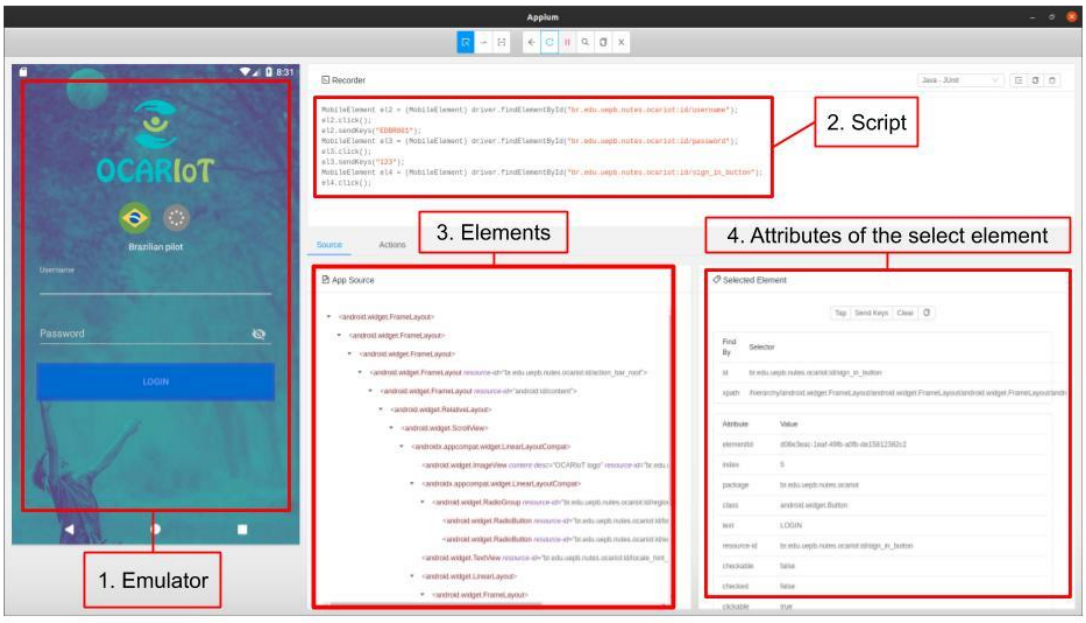
Após a especificação, é necessário modelar os casos de teste de forma que forneçam as instruções necessárias para garantir que os requisitos sejam validados. Para cada teste é criado um caso de teste que mostra o pré-requisito, o passo a passo de como deve ser executado e os detalhes de sua execução, incluindo o resultado do teste.

O Appium é responsável por executar o aplicativo fornecendo um script de teste. A execução é feita diretamente no dispositivo de forma automatizada que recebe os comandos e envia a resposta da execução, podendo ser acompanhada no log de execução.

4.3.3 *Execução*

Os testes são realizados diretamente em um dispositivo móvel usando o APK instalado. O APK pode ser baixado do GitHub do OCARIoT e a ferramenta do site oficial Appium. O emulador permite que o testador simule iterações diretamente no aplicativo e cada iteração pode ser gravada gerando um script, que pode ser usado no ambiente de desenvolvimento de teste e gerando as validações necessárias. É possível ter acesso a cada elemento da interface do usuário, onde cada um possui atributos que o testador pode utilizar na criação dos testes, como observado na Figura 31 abaixo:

Figura 31 - Painel de emulação do Appium



Fonte: OCARIOT (2020).

Com o script finalizado no ambiente de desenvolvimento e executado, os resultados esperados indicando se “Passou” para os casos de teste aprovados e “Falhou” para aqueles com algum tipo de erro na aplicação.

4.3.4 *Resultados*

Após a execução dos testes implementados foi obtido um resultado satisfatório na construção da aplicação visto que dos testes executados que passaram o resultado foi superior a 90%, como observado na Tabela 5 a seguir:

Tabela 5 - Resultados da execução dos testes do OCARIoT Data Acquisition App

Suíte de Teste	Testes Especificados	Implementado	Passando	Falhando
Login	25	23	23	0
Children	14	2	2	0
Settings	25	19	19	0
Welcome	4	4	3	1
Physical activity	7	6	6	0
Sleep	11	11	11	0
IoT - Weight and heart rate	8	8	7	1
Menu	3	3	2	1
Total	**97**	**76**	**73**	**3**
Resultados			**96.06%**	**3.94%**

Fonte: Adaptado de OCARIOT (2020).

4.3.5 *Gerenciamento de Bugs*

Os testes especificados que foram bloqueados ou não implementados serão implementados na próxima versão do aplicativo. Os erros encontrados dos casos de teste que falharam foram passados para equipe de desenvolvimento para correção, como mostrado na Tabela 6 a seguir:

Tabela 6 - Resultados dos testes que falharam do OCARIoT Data Acquisition App

Suíte de Teste	Falharam	Bugs Reportados	Corrigido	Bloqueado	Não implementado
Login	0	0	0	0	2
Children	0	0	0	12	12
Settings	0	0	0	6	6
Welcome	1	1	1	0	0
Physical activity	0	0	0	0	1
Sleep	0	0	0	0	0
IoT - Weight and heart rate	1	1	1	0	0
Menu	1	1	1	0	0
Resultados	**3**	**3**	**3**	**18**	**21**

Fonte: Adaptado de OCARIOT (2020).

4.4 Considerações finais

Apesar dos benefícios oferecidos pelo DDS, os desafios relacionados a pessoas, tecnologias, comunicação, gestão e processos impactam diretamente o processo de teste. Com a aplicação do processo de teste e uso de ferramentas propostas foi possível que todas informações das etapas e os artefatos gerados estivessem disponíveis para toda a equipe, mesmo que dispersas.

A partir da utilização do processo foi possível obter como resultado a criação de artefatos de teste e validações de funcionalidades do OCARIoT Dashboard, API Gateway e Data Acquisition App, cuja cobertura de teste indica valores como: Dashboard OCARIoT, com 62% de cobertura de telas, com índice de aprovação de 48%. Os valores são considerados baixos por ainda não ter sido concluído o ciclo completo do bug e algumas funcionalidades não terem sido totalmente implementadas. O API Gateway, com 65% de cobertura de recursos, com índice de aprovação de 83%. O App Data Acquisition, com 78% de cobertura, com índice de aprovação de 96%. Sendo esses índices considerados satisfatórios para a validação do API Gateway e Data Acquisition App.

5 CONCLUSÃO

As atividades de teste no processo de desenvolvimento de software proporcionam ao mercado a garantia de um produto de qualidade. Para realização de testes é necessário que um processo de teste seja definido, de forma que identifiquem quais serão as atividades realizadas, artefatos gerados e as responsabilidades da equipe. Os testes permitem que erros sejam encontrados e corrigidos antes da colocação do produto no mercado.

No contexto atual existe um aumento de projetos no modelo de Desenvolvimento Distribuído de Software (DDS), já que são apontados benefícios relacionados a diminuição de custo, *time-to-market* e o acesso ao mercado global. O DDS é caracterizado pela distribuição geográfica dos membros envolvidos no desenvolvimento do software.

Contudo, são apresentados desafios associados a este modelo, principalmente relacionados a comunicação, cultura, problemas estratégicos, gestão e processo. Desta forma, faz-se necessário um processo de teste único para os envolvidos, sendo importante o uso de normas, padrões de documentações e ferramentas.

Este trabalho teve como foco a adaptação do processo de teste para validação de projetos com DDS. Para isso foram estudados os desafios apresentados por este modelo de desenvolvimento, de forma a identificar quais necessidades suprir com o processo de teste proposto e obter como resultado a mitigação de erros encontrados no teste do software.

O processo de teste foi aplicado no OCARIoT por ser um projeto em colaboração com instituições distribuídas geograficamente, em países da Europa e Brasil. Os resultados obtidos na utilização do processo no Dashboard, API Gateway e Data Acquisition App foram artefatos de teste como plano de teste, casos de teste, scripts de automação, relatórios de execuções e reporte de bugs. Como resultado final, o processo contribuiu na identificação dos bugs nas funcionalidades permitindo sua correção. A partir destas informações foi possível identificar a cobertura de testes das funcionalidades implementadas e o índice de aprovação dos testes.

A abordagem apresentada neste trabalho ainda está em uso no processo de teste das novas funcionalidades do OCARIoT, podendo ainda ser obtidos resultados de sua aplicação. O uso do processo de teste apresentado pode ser aplicado também em outros projetos cujo modelo de desenvolvimento é o DDS, de forma a contribuir com equipes de testes.

REFERÊNCIAS

ANICHE, Mauricio. **Testes automatizados de software:** Um guia prático. São Paulo: Editora Casa do Código, 2015.

AUDY, J. L. N.; PRIKLADNICKI, R. **Desenvolvimento distribuído de software**. Rio de Janeiro: Elsevier, 2008.

BARTIÉ, Alexandre. **Garantia da qualidade de software**. Rio de Janeiro: Elsevier, 2002.

BLANCO, M. Z. **Documentação de Teste Baseado na Norma IEEE 829** - Estudo de Caso: "Sistema de Apoio a Tomada de Decisão". **Revista T.I.S.**, São Carlos, v. 1, n. 1, p. 91-97 jul. 2012.

BSTQB, Brazilian Software Testing Qualifications Board. **Certified Tester Foundation Level Syllabus**. Tradução realizada pelo WG-Traduções do BSTQB do syllabus do ISTQB: Certified Tester Foundation Syllabus. Versão 2018br, 2018.

COLLINS, Eliane et al. An industrial experience on the application of distributed testing in an agile software development environment. In: **2012 IEEE Seventh International Conference on Global Software Engineering**. IEEE, p. 190-194, 2012.

DE AMORIM, Deivison Gomes et al. Gerenciamento de teste de software: Um comparativo entre ferramentas open source. **GESTÃO. Org**, v. 14, n. 5, p. 296-302, 2016.

DE, Brajesh. **API Management**: An Architect's Guide to Developing and Managing APIs for your Organization. 1. ed. Bangalore: Apress, 2017.

DEVMEDIA. **Processo de Teste de Software**. 2012. Disponível em: https://www.devmedia.com.br/processo-de-teste-de-software/23795. Acesso em: 22 de Mai. de 2020.

DOCKER Docs. **Docker Overview**. 2020. Disponível em: https://docs.docker.com/get-started/overview/. Acesso em: 01 de Dez. de 2020.

EDUCBA. **What is Test Environment?**. 2020. Disponível em: https://www.educba.com/what-is-test-environment/. Acesso em: 01 de Dez. de 2020.

HERBSLEB, J. D., & MOITRA, D. Global software development. **IEEE software**, v. 18, n. 2, p. 16-20, 2001. Disponível em: sci-hub.se/10.1109/52.914732. Acesso em: 03 de Jun. de 2020.

IEEE Standards Coordinating Committee. Standard Glossary of Software Engineering Terminology. (IEEE Std 610.12-1990). Los Alamitos. **CA: IEEE Computer Society**, v. 169, 1990.

ISLAM, Rashedul; ISLAM, Rofiqul; MAZUMDER, Tohidul. Mobile application and its global impact. **International Journal of Engineering & Technology (IJEST)**, v. 10, n. 6, p. 72-78, 2010.

JAIN, Ritu; SUMAN, Ugrasen. A systematic literature review on global software development life cycle. **ACM SIGSOFT Software Engineering Notes**, v. 40, n. 2, p. 1-14, 2015.

JIMÉNEZ, M; PIATTINI, M; VIZCAÍNO, A. Challenges and improvements in distributed software development: A systematic review. **Advances in Software Engineering**, v. 2009, p. 1-15, jun. 2009. Disponível em: http://www.hindawi.com/journals/ase/2009/710971.html. Acesso em: 01 de Jun. de 2020.

MOCKUS, Audris; HERBSLEB, James. Challenges of global software development. In: **Proceedings seventh international software metrics symposium**. IEEE, p. 182-184, 2001.

MYERS, G. J. **The Art of Software Testing**. New York: John Wiley and Sons, 1979.

_____, G. J., SANDLER, C., BADGETT, T. **The Art of Software Testing**. 3. ed. Hoboken: John Wiley and Sons, 2011. 20 p.

OCARIOT. **Solução inteligente para cuidar da obesidade infantil usando o potencial da IoT.** 2018. Disponível em: http://www.ocariot.com.br/. Acesso em: 15 de Nov. de 2020.

_____ Consortium, **D5.2 OCARIoT Deliverable: OCARIoT app and OCARIoT dashboard prototypes design v2**, European Union and RNP, 2018.

_____ Consortium, **D5.12 OCARIoT Deliverable: OCARIoT Functional Validation v1**, European Union and RNP, 2020.

OTADUY, Itziar; DIAZ, Oscar. User acceptance testing for Agile-developed web-based applications: Empowering customers through wikis and mind maps. **Journal of Systems and Software**. v. 133. p. 212-229, 2017.

PRESSMAN, Roger S. **Engenharia de Software**: Uma abordagem profissional. 7. ed. Porto Alegre: AMGH. 2011.

PRIKLADNICKI, Rafael et al. Desenvolvimento distribuído de software: um modelo de classificação dos níveis de dispersão dos stakeholders. **Anais do I Simpósio Brasileiro de Sistemas de Informação**. SBC, 2004. p. 153-161. Disponível em: https://sol.sbc.org.br/index.php/sbsi/article/download/6373/6269. Acesso em: 02 de Jun. de 2020.

RIOS, Emerson; MOREIRA, Trayahú. **Teste de Software**. 3. ed. Rio de Janeiro: Alta Books Editora, 2013.

SELENIUM. **Getting started with WebDriver**. 2020. Disponível em: https://www.selenium.dev/documentation/en/getting_started_with_webdriver/. Acesso em: 06 de Set. de 2020.

SERRANO, Nicolas; CIORDIA, Ismael. Bugzilla, ITracker, and other bug trackers. **IEEE software**, v. 22, n. 2, p. 11-13, 2005.

SINGH, Shiwangi; GADGIL, Rucha; CHUDGOR, Ayushi. Automated Testing of mobile applications using scripting Technique: A study on Appium. **International Journal of Current Engineering and Technology (IJCET)**, v. 4, n. 5, p. 3627-3630, 2014.

SIOCHOS, Vasilis; PAPATHEODOROU, Christos. **Developing a formal model for mind maps**, p. 39-44, 2011. Disponível em: http://eprints.rclis.org/15842/1/04.Siochos.pdf. Acesso em: 28 de Nov. de 2020.

WHITE, L. J. Software Testing and Verification. **Advances in Computers**, Orlando, v. 26, p. 335-391, Jul. 1987.