ANDRHÉ VICTOR DOS SANTOS SILVA

TESTE DE SOFTWARE: UM ESTUDO DE CASO DA AUTOMATIZAÇÃO DE TESTES

Ji-Paraná

2019

ANDRHÉ VICTOR DOS SANTOS SILVA

TESTE DE SOFTWARE: UM ESTUDO DE CASO DA AUTOMATIZAÇÃO DE TESTES

Monografia apresentada à Banca examinadora do Centro Universitário São Lucas Ji-Paraná, como requisito de aprovação para obtenção do Título de Bacharel em Sistemas de Informação.

Orientador: Prof. Maigon Nacib Pontuschka.

Ji-Paraná

2019

Dados Internacionais de Catalogação na Publicação
Gerada automaticamente mediante informações fornecidas pelo(a) autor(a)

S586t Silva, Andrhé Victor dos Santos

Teste de software: um estudo de caso da automatização de testes / Andrhé Víctor dos Santos Silva-- Ji-Paraná, RO, 2019.

48 p.

Orientador(a): Prof. Me Maigon Nacib Pontuschka

Trabalho de Conclusão de Curso (Graduação em Sistemas de Informação) - Centro Universitário São Lucas

1. Desenvolvimento de produtos. 2. Qualidade da produção. 3. Processo de automação. I. Pontuschka, Maigon Nacib. II. Título.

CDU 004.4:658.562.4

Bibliotecário(a) Alex Almeida CRB 11.8537

SÃO LUCAS
EDUCACIONAL

ATA Nº 05/2019 DE TRABALHO DE CONCLUSÃO DE CURSO EM SISTEMAS DE INFORMAÇÃO

No segundo dia do mês de dezembro de 2019, das 17h as 22h reuniram-se na sala de Inovação Tecnológica 7 o(a) professor(a) orientador(a) Maigon Nacib Pontuschka e os(as) professores(as) José Rodolfo Milazzotto Olivas e Willian Alves de Oliveira Fachetti para comporem Banca Examinadora de Trabalho de Conclusão de Curso em Sistemas de Informação sob presidência do(a) primeiro(a), para analisarem a apresentação do trabalho "Teste de Software: um estudo de caso da automatização de testes e seus benefícios". Após as arguições e apreciação sore o trabalho exposto foi atribuída à menção como nota do Trabalho e Concluso do curso do Acadêmico(a) Andrhe Victor dos Santos Silva.

OBS: Trabalho de Conclusão de Curso (X) Aprovado ou (__) Reprovado com nota total de 9,4, atribuídos o valor de 9,4 (nove vírgula quatro pontos) para o trabalho escrito e o valor de 9,5 (nove vírgula quatro pontos) para a apresentação oral.

Andrhe Victor dos Santos Silva
Andrhe Victor dos Santos Silva

Prof. Esp. José Rodolfo M. Olivas

Prof. Esp. Willian A. de Oliveira Fachetti

Maigon Nacib Pontuschka
Prof. Me. Maigon Nacib Pontuschka
Orientador

Prof. Me. Thyago Bohrer Borges
Coord. Sistemas de Informação

Dedico este trabalho aos meus pais, que me deram total apoio durante toda a sua elaboração. Dedico também à minha namorada, que sempre me motivou para finaliza-lo. Meus amigos que incentivaram e contribuíram de alguma forma durante seu desenvolvimento.

AGRADECIMENTOS

Agradeço aos meus pais que me apoiaram durante toda a minha trajetória acadêmica, me dando apoio e me auxiliando quando foi possível. Sem eles, não seria possível eu ter chegado até aqui.

Agradeço a minha namorada, que sempre esteve ao meu lado nos momentos de dificuldade, sempre me dando forças para que eu seguisse em frente e completasse meus objetivos.

Aos meus colegas de trabalho, que contribuíram de alguma forma no desenvolvimento do projeto.

Aos meus amigos que se mantiveram ao meu lado durante todo o período do curso.

“A persistência é o caminho do êxito.”

(Charles Chaplin)

RESUMO

O presente trabalho trata do processo de automatização de testes de software. A automatização dos testes proporciona diversos benefícios, mas o principal ganho está na qualidade do produto. Softwares que possuem testes automatizados tendem a ter uma qualidade superior aos que não possuem. Outro fator complementar é o de que o ambiente de desenvolvimento se torna menos apreensivo, em que a implementação de uma nova funcionalidade não se torna mais uma grande dor de cabeça, tendo que se pensar se os novos módulos irão se integrar corretamente com os módulos mais antigos. A automação também diminui os riscos que os testes manuais realizados por equipes – mesmo que especializadas – trazem, devido ao fato de que estes testes podem ser realizados erroneamente. O objetivo do trabalho é apresentar as dificuldades no processo de automatização dos testes e apresentar os ganhos que foram obtidos através dessa automação.

Palavras-Chave: Teste de software, testes automatizados, testes unitários.

ABSTRACT

The present work deals with the automation process of software tests. Test automation provides several benefits, but the main gain is product quality. Software that has automated testing tends to have a higher quality than it does not have. Another complementary factor is that the development environment becomes less tense, in which the implementation of a new functionality does not become a major headache anymore, having to think about whether the new modules will integrate correctly with the more modules old ones. Automation also lessens the risks that manual testing by teams - even specialized ones - brings, due to the fact that these tests can be performed erroneously. The objective of this work is to present the difficulties in the automation process of the tests and present the gains that were obtained through this automation.

Keywords: Software testing, automated testing, unit testing.

LISTA DE FIGURAS

LISTA DE SIGLAS

IEEE	-	Institute of Electrical and Electronics Engineers
ABNT	-	Associação Brasileira de Normas Técnicas
ISO	-	International Organization for Standardization
CIP	-	Câmara interbancária de pagamentos
PCR	-	Plataforma de Recebíveis
FEBRABAN	-	Federação Brasileira de Bancos
SPB	-	Sistema de Pagamentos Brasileiro
BACEN	-	Banco Central
XML	-	Extensible Markup Language
HTML	-	Hypertext Markup Language
SLOC	-	Source Lines of Code

SUMÁRIO

1. INTRODUÇÃO

Nos dias atuais, diversas áreas da sociedade estão se tornando cada vez mais dependentes da tecnologia. Essa dependência faz com que os sistemas – ou softwares – tenham que possuir uma qualidade cada vez maior. A qualidade do software deixou de ser um diferencial para a competitividade com outros produtos semelhantes e passou a ser um requisito obrigatório para que ele possa ser adquirido e utilizado.

Um dos grandes desafios do processo de desenvolvimento de software é o de garantir a qualidade do produto. Esse desafio se dá por diversos motivos, sejam eles técnicos, humanos ou econômicos. A falta de qualidade nos sistemas pode causar prejuízos econômicos e, em muitos casos, pode custar até mesmo uma vida humana. Por isso, diversos esforços vêm sendo tomados para que a qualidade de software seja melhorada, tanto no aprimoramento das tecnologias que são empregadas na codificação do software, como nas metodologias de desenvolvimento. O aprimoramento na qualidade do software terá impacto não somente na experiência dos usuários, mas também em todo o processo de desenvolvimento, tornando o software mais flexível e manutenível.

Uma das maneiras de garantir o aumento na qualidade do software, é com a implementação de testes.

Para Bartié (2002), novas características e funcionalidades devem ser acrescentadas ao software para que ele cresça e os usuários possam perceber a sua evolução. Porém, conforme novas funcionalidades vão sendo introduzidas ou funcionalidades existentes vão sendo alteradas, novas baterias de testes precisam ser executadas para assegurar que o software não possui erros.

Bartié (2002) explica que, pelo fato de a cada atualização ser necessário executar testes, é necessário que esses testes sejam automatizados. “A falta de automação dos testes vai se tornando mais crítica à medida que avançamos no desenvolvimento do software e muitas funcionalidades precisam ser testadas”. (BARTIÉ, 2002, p. 45)

Com as premissas mencionadas acima, neste trabalho serão abordadas as dificuldades de se automatizar um teste de software que geralmente é realizado manualmente sempre que uma atualização é disponibilizada.

O software a ser testado é utilizado em uma cooperativa de crédito da região norte do Brasil para realizar a integração entre um sistema emissor de boletos bancários,

com a Câmara Interbancária de pagamentos - CIP. O papel da CIP é de distribuir o boleto emitido pela cooperativa para todos os bancos que fazem parte da Plataforma de Recebíveis – PCR. Esses boletos passam então a ser “boletos registrados”, com as informações como valor, data de vencimento e juros estando disponíveis via consulta, fazendo com que o número de fraudes no pagamento de boletos seja diminuído.

As soluções encontradas durante o processo de implantação e os benefícios de se automatizar serão listados nos resultados, assim como as dificuldades encontradas durante a implementação. Os resultados serão analisados de forma qualitativa, comparando os testes que são realizados manualmente a cada atualização com os testes automatizados.

1.1. Problematização

O objeto de estudo deste trabalho é um sistema de emissão e pagamento de boletos de uma cooperativa de crédito. O sistema em questão, realiza a integração entre seus boletos e a base centralizada da CIP por meio de um software de terceiros. A empresa responsável pelo software disponibiliza frequentemente atualizações para que continuem a estar de acordo com as normas estabelecidas pelo comitê de gestão da CIP. Quando o software é atualizado, é necessário testar a integração entre o software e o sistema da cooperativa.

1.2. Objetivos

1.2.1. Objetivo Geral

Utilizando técnicas de engenharia de software, procura-se realizar a automação dos testes, que até então são realizados de forma manual a cada atualização disponibilizada pela empresa responsável pelo software.

1.2.2. Objetivos Específicos

- Realizar o levantamento de requisitos para automação do sistema.
- Automatizar os testes de integração entre o sistema da cooperativa e o software da empresa terceira.
- Coletar informações referentes ao processo de desenvolvimento dos testes.
- Detalhar as dificuldades encontradas durante o processo.

1.2.3. Delimitação do estudo

A automação dos testes será realizada somente nas funcionalidades que realizam a liquidação e recebimento de boletos de pagamento.

2. REFERENCIAL TEÓRICO

2.1. Visão geral de teste de software

Bartié (2002) explica que nos primórdios do desenvolvimento de software, o processo de teste não passava de analisar o código identificando possíveis problemas e corrigi-los. Eram os próprios desenvolvedores que realizavam essa tarefa e, por isso, geralmente só era realizado quando o software já estava quase pronto para ser entregue.

Para Vicente (2010) o termo “Teste de software” abrange diversas atividades, como o teste unitário realizado pelo programador até o teste de aceitação realizado pelo consumidor final. Segundo ele, os casos de testes devem ser planejados em cada estágio de acordo com seus objetivos. Os testes podem ter objetivos diferentes de acordo com a abordagem estabelecida, como expor a conformidade estabelecida nos requisitos, testar a performance do software ou ainda expor os erros existentes.

Segundo Bernardo (2011) os testes de software objetivam aumentar a qualidade do software por meio de verificações e validações. Os testes ainda podem gerar dados importantes para os desenvolvedores, como a porcentagem de aceitação dos testes.

Já para Sommerville (2011) o teste deve ser aplicado para demonstrar se o software atende aos propostos e também serve para encontrar possíveis erros no software antes de seu uso. Os testes devem ser executados utilizando dados fictícios e os resultados dos testes devem ser analisados a fim de se encontrar erros ou informações incorretas.

Portanto, podemos afirmar que o principal objetivo dos testes é encontrar erros nas funcionalidades que o software se propõe a executar. Outro objetivo dos testes é o de encontrar erros no código-fonte produzido pelos programadores durante o processo de desenvolvimento, estes mais fáceis de serem encontrados. Os testes também têm o papel de encontrar inconsistências entre as especificações do produto, e o produto que realmente está sendo entregue.

2.1.1. Validação e Verificação

A validação e a verificação são processos utilizados para assegurar a qualidade de software, confirmando se este cumpre as especificações para as quais foi construído para atender.

Moreno (2012) explica que a verificação é uma atividade responsável por certificar que o software desenvolvido atende aos requisitos funcionais e não-funcionais. Validação é o processo de certificação de que o software atende aos requisitos para os quais foi projetado. Os dois processos – validação e verificação – não são processos independentes.

> O teste é parte de um amplo processo de verificação e validação (V&V). Verificação e validação não são a mesma coisa, embora sejam frequentemente confundidas. (SOMMERVILLE, 2011, p. 145)

Segundo a ABNT (2003), validação é a confirmação de que os requisitos que foram estipulados para um determinado software foram atingidos e verificação é a confirmação de que os requisitos sejam alcançados.

Podemos diferenciar verificação e validação como:

- Verificação: não verifica se os resultados estão corretos, apenas verifica a funcionalidade
- Validação: verifica se os resultados estão de acordo com o esperado e se seguem as regras estabelecidas pelos *stakeholders*.

Para Sommerville (2011) a diferença entre validação e verificação pode ser simplificada em duas perguntas:

- Validação: estamos construindo o produto correto?
- Verificação: estamos construindo o produto da maneira certa?

2.2. A importância dos testes

Estamos cada vez mais dependentes de softwares, direta ou indiretamente. É muito difícil realizarmos alguma atividade na qual não exista um software que se propõe a fazer ou auxiliar na realização da mesma. Os softwares se tornaram produtos que proporcionam serviços que oferecem mais praticidade na execução de diversas tarefas de nosso cotidiano.

Para Pressman (2011), os testes precisam ser aplicados seguindo uma estratégia pré-definida, ou então o tempo gasto nos testes terá sido em vão. Isso porque erros podem passar despercebidos sem terem uma definição clara.

Com o aumento exponencial no consumo e produção de softwares, a qualidade que estes possuem é um ponto decisivo na escolha dos consumidores, portanto, é primordial para um software que este tenha qualidade.

O papel do teste de software é garantir que ele tenha qualidade e garantir que os desenvolvedores possam trabalhar em novas funcionalidades sem medo de causar incompatibilidades e erros com módulos mais antigos. O teste também permite que alterações sejam feitas sem o receio de que a funcionalidade não funcionará do mesmo jeito que antes.

2.2.1. Qualidade de software

Sommerville (2011) afirma que a qualidade de software começou a ser importante já na década de 60, quando notou-se que os softwares desenvolvidos apresentavam muitos problemas, eram lentos, e de difícil manutenibilidade. Para diminuir esses problemas, técnicas de gerenciamento de qualidade (inspiradas nas técnicas utilizadas na indústria de manufatura) foram aplicadas no desenvolvimento do software, que apresentou melhoras significativas na qualidade, diminuindo os problemas de lentidão e manutenibilidade.

Para Bartié (2002), entre os principais motivos de falha na implantação de processos de qualidade de software, estão:

1. A gerência de qualidade não tem independência dos outros setores.
2. Ausência de testes automatizados
3. A qualidade tenta ser aplicada somente depois do produto ser desenvolvido.
4. Ausência de profissionais qualificados na área de qualidade de software.
5. Os testes cobrem apenas as novas funcionalidades desenvolvidas.
6. Planejamento de testes ineficaz.
7. Testes são sacrificados pela falta de tempo ou orçamento.
8. Testes são realizados pelos próprios desenvolvedores ou analistas.

2.2.2. Por que não existe fase de teste?

Verificar se um software atende aos requisitos especificados é uma parte muito importante no processo de desenvolvimento. Sendo assim, é normal nos questionarmos do porquê de não existir uma fase no processo de desenvolvimento dedicada ao teste do sistema.

Schach (2007) explica que verificar o software depois que o processo de codificação tiver terminado, já é tarde demais. Pior ainda se o software já foi entregue

ao cliente. Um dos motivos que podem fazer esse tipo de situação acontecer é a falha na documentação. Se uma falha está presente na documentação, e esta foi levada adiante durante todo o processo de desenvolvimento e durante a implantação, o problema tomará muito mais tempo e será muito mais custoso.

Há momentos no processo de desenvolvimento em que o teste é realizado quase que exclusivamente. Isso deve ocorrer no final de cada etapa (verificação) e é extremamente necessário antes que o produto seja entregue ao cliente (validação). Embora existam momentos em que o teste esteja sendo feito mais do que qualquer outra tarefa, nunca deve haver momentos em que nenhum teste esteja sendo realizado. Se o teste for tratado como uma fase de testes separada, existe um perigo muito grande de que o teste não seja realizado constantemente ao longo de todas as fases do processo de desenvolvimento e manutenção do produto.

O ponto a ser atingido é a verificação contínua do software, em todas as etapas de desenvolvimento e manutenção de software. Por isso uma fase de testes separada é incompatível com o objetivo de garantir que um produto de software seja o mais livre de falhas possível em todos os momentos. (SCHACH, 2007)

2.3. Modelos de teste

Existem diversas maneiras de se testar um software, que fornecem critérios e técnicas diferentes aos envolvidos no projeto de teste de software. Essas técnicas oferecem também uma forma de melhorar a qualidade do software, fazendo com que seja mais provável encontrar erros antes do produto ser entregue ao cliente final. As duas principais técnicas serão apresentadas abaixo.

2.3.1. Modelo Caixa-Preta

O teste de caixa-preta, ou teste funcional, é uma técnica de teste que consiste em considerar o software como uma caixa-preta, na qual não se sabe o que existe em seu interior. Neste tipo de teste, as validações e verificações são aferidas a partir da inserção de dados de entrada, e analisando se os dados de saída estão de acordo com a especificação realizada no projeto do software. Neste tipo de procedimento, o software é avaliado do ponto de vista do usuário. (DELAMARO, MALDONADO e JINO, 2007)

Segundo Pan (1999), os testes baseados no modelo caixa-preta, derivam-se somente dos requisitos funcionais do software, e não se baseiam no código-fonte.

Eles também são chamados de testes orientados a dados, ou testes baseados em entrada e saída.

Neste modelo, somente a funcionalidade do módulo a ser testado é que importa, e por isso ele também se refere ao teste funcional – um modelo que tem o foco na execução das funções do software e na análise nas entradas e saídas.

O encarregado de testar enxerga o software como uma caixa-preta, e o que está no seu interior não lhe interessa. O importante para ele é o resultado que o software está retornando de acordo com a entrada que foi feita. O dever dele é inserir entradas variadas, e validar se as saídas estão de acordo com as especificações.

A figura 1 representa o modo que o software é visto durante a realização dos testes.

Figura 1 - Teste caixa-preta

Fonte: BARTIÉ, 2002

De acordo com Myers (2004), a única maneira de encontrar todos os erros do software utilizando o método de caixa-preta, é inserindo todas as entradas possíveis, até mesmo as que são inválidas para o sistema. Esse procedimento é chamado de “teste exaustivo de entrada”. Executar testes utilizando o método de entrada exaustiva é praticamente impossível, visto que o número de entradas possíveis para um software complexo pode ser infinito.

Delamaro, Maldonado e Jino (2007) afirmam que a limitação proporcionada pelo “teste exaustivo de entrada” fez com que as atividades de testes fossem diversificadas, e os critérios utilizados para serem realizadas as avaliações fossem definidos de forma diferente para cada tipo de teste.

Myers (2004) conclui que o método de caixa-preta não é eficiente para encontrar erros no software. O método seria eficiente para testar saídas controladas, onde as entradas são previsíveis. Outra afirmação de Myers, é que para testar com eficiência

usando esse método, você pode fazer algumas suposições. Por exemplo, um software que detecta um tipo de triângulo, onde são inseridos os valores de cada um de seus lados. Se inserirmos 2, 2, 2, e o software reconhecer o tipo do triângulo como equilátero, podemos levar em consideração que ele fará o mesmo se inserirmos 3, 3, 3. Com esse tipo de suposição, a quantidade de casos de teste pode ser diminuída.

Uma das grandes vantagens do teste caixa-preta está no fato do modelo ser facilmente implementado, pois os únicos critérios que são requeridos são as especificações do software. Desta forma, os testes podem ser aplicados em softwares que utilizam diversas metodologias distintas, como, por exemplo, a orientada a objetos, a procedural e a funcional.

Portanto, o modelo de caixa-preta se mostra útil para validações onde apenas o resultado final do programa é importante. Pode ser usado principalmente pela equipe encarregada das regras de negócio e que detém as especificações do software. É importante ter em mente, que nesse tipo de teste é impossível cobrir todos os cenários.

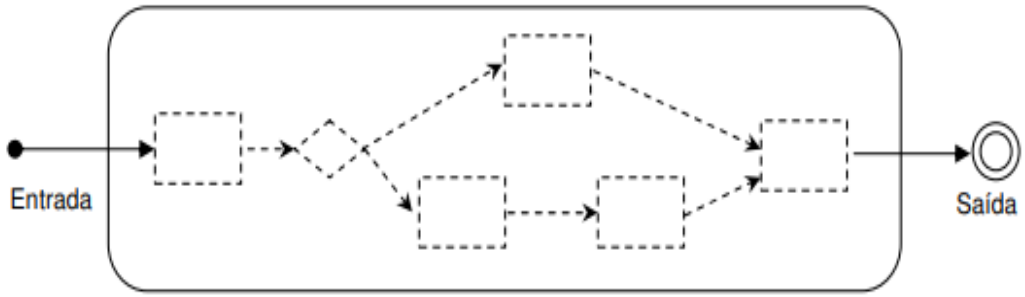
2.3.2. Modelo Caixa-Branca

Ao contrário do modelo caixa-preta, o modelo de caixa-branca leva em consideração as estruturas internas do programa, principalmente o seu código-fonte. “Essa estratégia obtém os dados do teste a partir da observação da lógica do programa (e frequentemente, infelizmente, negligenciando a especificação).” (MYERS, 2004)

O modelo caixa-branca, também chamado de técnica estrutural, tem seus requisitos estabelecidos a partir de uma implementação pré-determinada, e a aferição é realizada a partir da execução de partes ou componentes do software. Nessa execução, diversas etapas são analisadas, como os caminhos lógicos pelos quais foram passados, o uso e sobrescritas das variáveis e as condições dos laços de repetição. (DELAMARO, MALDONADO e JINO, 2007)

Segundo Bartié (2002), o encarregado de implementar os testes de caixa-branca deve ter pleno conhecimento das tecnologias empregadas no software: desde o código-fonte à estrutura do banco de dados. Esse modelo de teste tem uma grande capacidade de encontrar erros, porém são os mais trabalhosos de serem implementados. Existe também o fato de que é necessário um profissional qualificado para escrevê-los, o que encarece sua implementação.

Figura 2 - Modelo caixa-branca



Fonte: FURLAN, 2009, p. 24

Segundo Delamaro, Maldonado e Jino (2007), o modelo caixa-branca apresenta algumas limitações e desvantagens em relação aos demais. Essas limitações fazem com que a automatização dos testes possa ser comprometida. Entre as limitações, encontra-se uma falha nas validações de controle de fluxo: se o software não implementa alguma função, não é possível afirmar que existe um erro visto que nenhum teste irá passar por ela.

Outra desvantagem, para Delamaro, Maldonado e Jino (2007), está no fato de que o software, coincidentemente pode apresentar um dado correto a partir do dado que foi passado como entrada, fazendo com que o teste seja completado corretamente. Porém, se outro dado de entrada for passado, o resultado apresentado poderia estar incorreto, fazendo com que o teste não esteja completamente coberto de inconsistências.

2.4. Fases de teste

Pressman (2011) explica que o processo de desenvolvimento de software, é formado por um conjunto de etapas, e para cada etapa existe uma fase de teste. Além disso, muitos tipos de erros requerem tipos de testes específicos para serem encontrados e solucionados. Por isso, os casos de testes são separados por tipos. Segundo Bernardo (2011) essa separação ocorre por 4 motivos:

1. Facilita a manutenção dos testes;
2. Cada tipo de teste pode utilizar uma ferramenta e metodologia distinta;
3. As execuções dos testes mais lentos não irão interferir na velocidade dos testes que são realizados mais rapidamente.

4. Facilita a coleta dos dados realizados por cada tipo de teste;

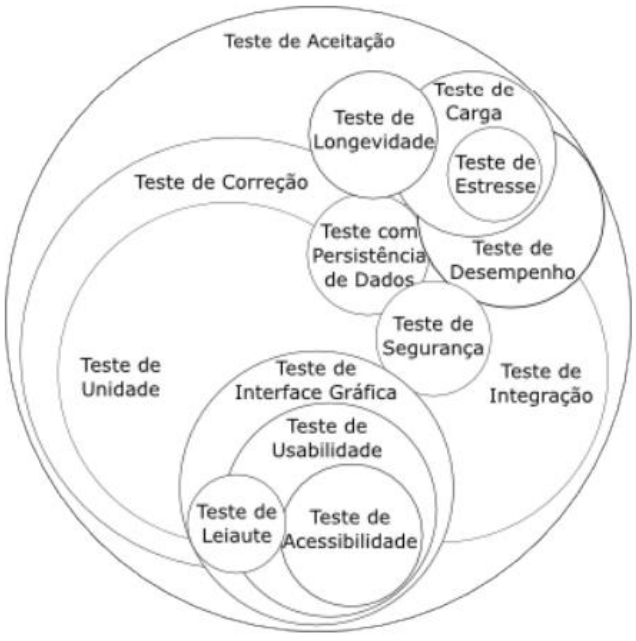
Para Bernardo (2011) um dos principais motivos que levam à criação de testes de baixa qualidade, é a utilização de ferramentas que não pertencem ao tipo de teste ao qual estão sendo empregadas. Para fazer isso, o código será menos enxuto e mais difícil de manter, o que pode acabar fazendo com que o teste perca seu propósito ou sua eficácia.

Segundo Sommerville (2011), no processo de teste todo software deve passar pelas seguintes fases:

1. Teste de desenvolvimento – Nessa etapa, enquanto o sistema está sendo desenvolvido, simultaneamente ele é testado para descobrir *bugs* ainda durante a codificação.
2. Teste de versão – Nessa etapa, é recomendável que os testes sejam executados por uma equipe independente da equipe de desenvolvimento. Essa etapa será realizada sempre que uma nova versão estiver pronta para ser liberada para os usuários.
3. Teste de usuário – Etapa em que os próprios usuários irão realizar os testes, preferencialmente no mesmo ambiente que irão utilizar o software normalmente.

A figura 3 ilustra o relacionamento entre os diversos tipos de teste.

Figura 3 - Tipos de teste



Fonte: BERNARDO, 2011, p. 39

2.4.1. Teste unitário

Segundo Myers (2004), teste unitário – ou teste de módulo – é o processo de testar processos, funções ou rotinas individuais de um software. Nesse tipo de teste, o software não é testado como um todo, mas sim como um conjunto de pequenas partes que trabalham juntas. A finalidade é comparar o resultado das funções do programa, com as especificações referentes ao módulo.

O teste unitário é a primeira fase do processo de testes e tem por objetivo “[...] exercitar adequadamente toda a estrutura interna de um componente, como os desvios condicionais, os laços de processamento e todos os possíveis caminhos alternativos de execução.” (BARTIÉ, 2002)

Sommerville (2011, p. 148) afirma que “[...] teste unitário é o processo de testar os componentes de programa, como métodos ou classes de objeto.”. Outra afirmação de Sommerville (2011, p. 20) é que “O teste unitário envolve a verificação de que cada unidade atenda a sua especificação”.

Conforme Pressman (2011) afirma, no teste de unidade são testados fluxos de controle para descobrir se o módulo se comporta da maneira esperada. Testes unitários tem o foco na lógica interna e nas estruturas de dados internas. A complexidade desse tipo de teste depende do tamanho do componente que está sendo testado.

Para Myers (2004), o teste unitário traz três benefícios principais: o primeiro é que ele oferece uma maneira de gerenciar os diferentes módulos do sistema, já que a atenção é focada unidades específicas do programa. O segundo, é que ele facilita a tarefa de depuração (processo de identificar um erro), pois, quando um erro é encontrado, é possível descobrir de qual módulo ele pertence. Por último, o teste unitário oferece paralelismo no processo de teste, visto que é possível testar de testar vários módulos ao mesmo tempo.

Após a realização dos testes unitários, é esperado que todos os módulos estejam de acordo com as especificações do projeto. Após a execução dos testes unitários, espera-se que a próxima etapa seja o teste de integração entre os componentes. Desta forma, é possível manter um desenvolvimento contínuo dos testes.

2.4.2. Teste de integração

Pressman (2011) descreve teste de integração como uma maneira de se modelar um software de modo que os testes sejam associados e dependentes entre si. O principal objetivo dos testes de integração é o de relacionar os testes unitários – construídos anteriormente – para realizar a validação de dados que são passados de um módulo a outro.

Segundo Myers (2004), testes de integração precisam de um plano de integração, pois o plano de integração irá ter um papel importante na definição de papéis. O plano de integração define a ordem da integração, a capacidade funcional de cada módulo e as responsabilidades de cada um.

Conforme Bartié (2002) explica, os testes de integração são uma continuação natural dos testes unitários, e consistem basicamente em manter a compatibilidade com os módulos existentes no software. Os erros que devem ser identificados neste estágio, são principalmente as dependências de funções e atributos compartilhados entre os componentes. Ainda segundo ele, as integrações não ocorrem somente em níveis básicos, como entre classes. As dependências podem ser entre simples classes, até subsistemas inteiros.

Muitas pessoas podem se questionar, se todos os módulos do sistema estão sendo testados individualmente (nos testes unitários), por que devemos testá-los juntos? Pressman (2011) explica que durante a comunicação de um componente com outro dados podem ser perdidos, a combinação de resultados entre funções pode se comportar diferente, uma determinada imprecisão que individualmente pode ser aceitável pode se amplificar de modo que torne-se inaceitável, entre outros problemas ocasionados pela interface de comunicação entre os componentes/módulos.

2.4.3. Teste de sistema

Para Sommerville (2011) os testes de sistema envolvem a integração dos módulos do sistema para que uma versão do sistema seja criada. O teste de sistema realiza a verificação de que todos os módulos do sistema são compatíveis e se realizam as interações entre si de forma correta.

Pressman (2011) resume teste de sistema como uma série de testes que tem o objetivo de exercitar o software. Ele cita 4 pontos importantes para serem seguidos durante o teste de sistema:

1. Prever e manipular erros vindos de diferentes elementos do software.

Executar e simular testes com dados incorretos.

2. Registrar os resultados, de forma que se tenha evidências das causas dos erros.
3. Planejar e projetar o teste de forma que o software seja testado adequadamente.

Segundo Bartié (2002), a fase do teste de sistema é a fase mais complexa de ser implementada e é a mais mal compreendida pelos desenvolvedores, pois nessa fase os testes irão avaliar o sistema como um todo. Nessa fase, a maior parte dos erros já foram detectados pelos testes unitários e os testes de integração.

Nessa etapa dos testes, as especificações do projeto não são mais tão importantes, e o essencial é dar o foco em outros quesitos como performance, segurança, disponibilidade, instalação e recuperação. É importante que, ao chegar nessa etapa, o hardware utilizado na execução do software seja o mais parecido possível com o que será usado em produção, pois dessa forma é possível obter métricas mais precisas. (BARTIÉ, 2002)

2.4.4. Teste de regressão

Cada vez que se faz alterações no software, é preciso executar novamente os testes que já foram executados e que passaram, para ter certeza de que as mudanças realizadas não interferiram nas funcionalidades que já existiam. A execução pode ser manual ou automatizada através de algumas ferramentas. (FURLAN, 2009)

Sommerville (2011, p. 156) resume o teste de regressão como:

> Conjuntos de testes que tenham sido executados com sucesso, após as alterações serem feitas em um sistema. O teste de regressão verifica se essas mudanças não introduziram novos bugs no sistema e se o novo código interage com o código existente conforme o esperado.

Pressman (2011) afirma que conforme o teste de integração avança, o número de testes de regressão irá crescer exponencialmente. Por isso, os casos de testes que serão reexecutados devem conter apenas os testes que cobrem funções importantes no software. Caso contrário, o teste de regressão se tornará ineficiente e não será nada prático.

Os testes de regressão não são realizados durante o processo de desenvolvimento inicial do software, mas sim durante as manutenções que são realizadas ao longo de sua vida útil. São executados nesse momento para verificar se

novos erros foram introduzidos no decorrer do processo de codificação. (DELAMARO, MALDONADO e JINO, 2007)

Para Bernardo (2011), o termo “teste de regressão” deve ser utilizado apenas no caso de execução dos testes de forma manual, visto que ao realizar os testes de forma automatizada, todos os testes passam a realizar o papel de um teste de regressão, já que eles ajudam a minimizar os erros encontrados nas atualizações do software.

2.4.5. Teste de aceitação

Sommerville (2011) define que os testes de aceitação são o último estágio de testes. Esse estágio é realizado antes que o software seja entregue ao cliente. Por ser o último processo de teste, os dados utilizados nessa fase são fornecidos pelo próprio cliente, e isto pode fazer com que novos erros sejam encontrados ou sejam descobertos inconsistências nas definições dos requisitos do software. Isto porque dados reais fazem com que o software se comporte de maneiras diferentes. Outro fator que pode ser analisado neste estágio é a performance, que pode não satisfazer o cliente.

Pressman (2011) defende que os testes de validação – como também são conhecidos – podem ser realizados assim que o teste de integração seja finalizado. Segundo ele, o foco do teste de validação está nas ações que o usuário irá desempenhar no sistema e nas saídas que são apresentadas a ele. “[...] a validação tem sucesso quando o software funciona de uma maneira que pode ser razoavelmente esperada pelo cliente”. (PRESSMAN, 2011, p. 417)

Bartié (2002) aponta que geralmente os testes de aceitação, estão restritos às regras de negócio, e podem deixar outros requisitos de fora – propositalmente. Espera-se que os requisitos que não serão tratados nessa fase, já tenham sido automatizados. Bartié ainda aponta que esse tipo de teste é utilizado em projetos que serão utilizados por mais de uma empresa em que geralmente, elas realizam o teste de aceitação em conjunto, como um comitê. Esse comitê pode aprovar ou não, as novas funcionalidades que foram entregues.

2.5. Documentos de testes

No desenvolvimento de um sistema, uma etapa muito importante é a de testes, pois é nela em que nos certificamos que não há erros críticos no software e é quando se verifica se todos os requisitos que foram especificados foram atendidos. Embora

essa etapa seja de plena importância para a garantia de qualidade do software, poucas são as equipes que a executam da maneira correta. Geralmente são feitos somente debugs simples, durante o processo de codificação. (DINIZ, 2008)

Segundo (Bandeira, et al. 2016) um analista de teste gasta em média 50% a 60% do tempo elaborando documentações relacionadas a teste. A norma IEEE 829-1998 define oito documentos que cobrem as tarefas de planejamento, especificação e resumo dos testes.

É importante escolher quais documentos irão compor cada etapa do teste, visto que eles têm que dar suporte independentemente do tamanho e complexidade do sistema. Os documentos estão listados abaixo.

2.5.1. Plano de teste

Para Diniz (2008), o documento de plano de teste deve ser realizado antes da solução ser codificada, pois é nessa fase em que será definido o que o software se propõe a fazer e de que maneira ele será construído. Fazendo dessa forma, é possível planejar a forma que irá se certificar que o software realmente atende aos requisitos.

2.5.2. Especificação de projeto de teste

A especificação de projeto de teste delimita o plano de testes, identificando quais funcionalidades do software serão testadas, e quais características serão observadas durante os testes. O documento também deve identificar os procedimentos que serão executados nos testes. (NOBIATO, DA SILVA, *et al.*, 2004)

2.5.3. Especificação de caso de teste

Deve definir os casos de testes. Esse documento deve conter os tipos de dados de entrada que serão utilizados, quais são os resultados esperados, quais ações são necessárias para a realização do caso de teste, as condições especiais para realização (caso existam) e as dependências entre os casos de teste. (NOBIATO, DA SILVA, *et al.*, 2004)

2.5.4. Especificação de procedimento de teste

“Especifica os passos para executar um conjunto de casos de teste.” (NOBIATO, DA SILVA, *et al.*, 2004, p. 7). Para Carline (2012, apud RIOS, 2007), o documento deve conter as seguintes seções: identificador, propósito, requisitos especiais e passos do procedimento.

2.5.5. Diário de Teste

O diário de teste apresenta os registros realizados durante o processo de teste. Os registros devem possuir a data em que foram realizados, com observações relevantes acerca dos testes. Quanto mais detalhados forem esses registros, mais relevantes os diários de testes se tornam. (NOBIATO, DA SILVA, *et al.*, 2004)

2.5.6. Relatório de Incidente de Teste

Assim como o diário de teste, o relatório de incidente deve apresentar registros relevantes que foram realizados durante o processo de teste. Diferente do diário, esse relatório irá conter apenas eventos que irão precisar de uma análise ou supervisão posterior. (NOBIATO, DA SILVA, *et al.*, 2004)

2.5.7. Relatório-Resumo de Teste

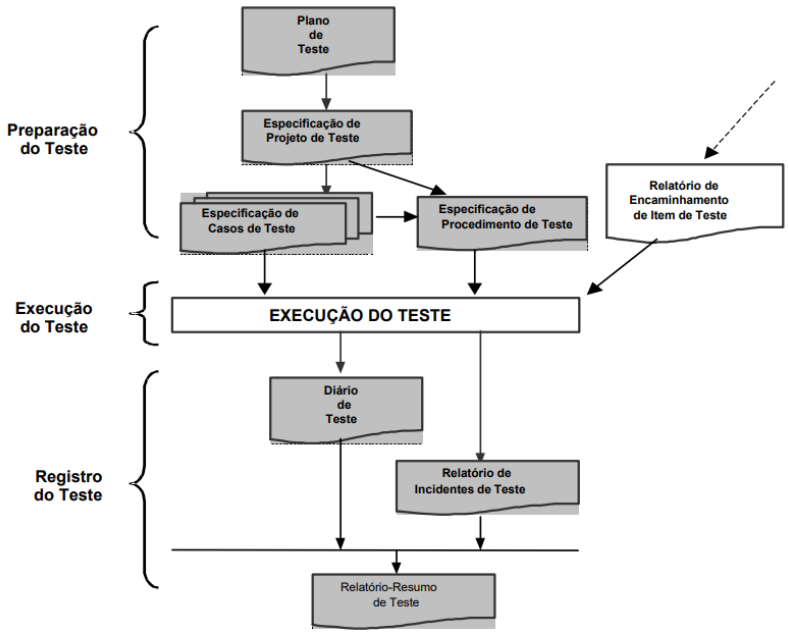
Deve apresentar resumidamente os resultados provenientes dos testes que são associados a determinadas especificações do projeto do teste. Ele também deve mostrar de forma clara, avaliações baseadas em seus resultados. (NOBIATO, DA SILVA, *et al.*, 2004)

2.5.8. Relatório de Encaminhamento de Itens de Teste

Deve identificar quantos e quais itens foram encaminhados para teste, caso estes sejam executados por uma equipe diferente da equipe de desenvolvimento. Carline (2012, apud RIOS, 2007), afirma que o relatório deve possuir os campos: identificador, itens passados, localização, estado e aprovações.

Segundo Nobiato *et al.* (2004), a norma IEEE 829 separa as atividades de teste em três etapas: preparação, execução e registro do teste. A figura abaixo separa e relaciona as etapas e documentos de testes:

Figura 4 - Relacionamento entre os Documentos de Teste



Fonte: NOBIATO *et al.*, 2004, p. 7

Além de apresentar e especificar uma série de documentos que podem ser utilizados no processo de planejamento de testes, a norma também fornece informações importantes para a realização dos testes. Se usado de maneira correta, a norma pode ajudar a gerenciar as fases de planejamento, projeto e a própria realização da atividade de teste. A norma pode evitar principalmente, a realização de testes somente após a codificação do software estar concluída. (NOBIATO *et al.*, 2004)

2.6. Testes automatizados

Carline (2012) afirma que a maioria das empresas de pequeno a médio porte, executam a bateria de teste de seus softwares manualmente. Esse tipo de abordagem não é recomendada, pois ela é suscetível a falhas por parte de quem está executando os procedimentos de teste. Outro fator limitante deste tipo de teste, é o tempo que se leva para completar o teste de uma unidade ou módulo, e consequentemente aumenta o custo de operação.

O ideal é utilizar testes automatizados, pois com eles é possível realizar um maior número de testes em menor tempo. Os testes automatizados realizam a verificação

de que o software atende aos requisitos que foram especificados. Os testes automatizados geralmente fornecem um relatório, onde é possível identificar se foram realizados com sucesso ou não.

Para Bernardo (2011) automatizar testes, é o ato de tornar um ou mais testes independentes da ação humana. Assim como os testes manuais, o objetivo dos testes automatizados é garantir e melhorar a qualidade dos softwares validando e verificando seus requisitos. Ainda segundo ele, a automação dos testes expande a área de conhecimento relacionada a testes, fazendo com que a implantação, manutenção e execução dos testes sejam diferentes.

É importante enfatizar que nem sempre os testes manuais podem ser substituídos completamente pelos testes automáticos. Muitos sistemas contam com regras de negócio complexas, que fazem com que a automatização de uma tarefa seja de extrema dificuldade.

Na opinião de Sommerville (2011) devemos automatizar os testes sempre que for possível. Na automação de testes unitários, geralmente utilizam-se *frameworks* de automação de testes, que fornecem recursos prontos para serem utilizados. Esses *frameworks* permitem que todos os testes automatizados que foram escritos, sejam executados e no final, apresentar um relatório apresentando os erros (se ocorreram) e onde aconteceram. Geralmente essa execução automática demora poucos segundos, e assim pode ser realizada cada vez que ocorre uma alteração no código.

Sommerville (2011) aponta que os testes automatizados são implementados em três etapas:

1. Uma parte de configuração, em que o sistema é preparado com as entradas e as saídas que são esperadas.
2. Uma parte de chamada, quando se executa o método ou função a ser testado.
3. E uma parte de afirmação, onde o resultado da “chamada” é comparado com a saída que era esperada. Caso seja igual, o teste foi realizado com sucesso. Caso seja diferente, o teste falhou.

Conforme Bartié (2002) explica, uma das grandes vantagens da automatização dos testes está na reutilização deles nas várias iterações que o software pode ter. Por exemplo, se em 3 iterações a mesma funcionalidade foi alterada, o mesmo teste pode ser executado em cada uma dessas iterações. O fato de exatamente o mesmo teste ser executado em cada iteração se mostra como mais uma vantagem em relação aos

testes manuais. Nos testes manuais, a pessoa que está executando o teste pode inadvertidamente fazer algo ligeiramente diferente em cada uma dessas iterações, e dessa forma, cada uma pode apresentar resultados diferentes das demais.

Para Bernardo (2011), além da verificação que os testes automatizados realizam, outro benefício é o de encontrar os efeitos colaterais de ferramentas e *frameworks*. Os testes de interface e os de aceitação podem realizar a demonstração do sistema e ainda entregar um manual pronto para o usuário. Além desses pontos, os relatórios que são obtidos após a execução dos testes podem ser utilizados na documentação do sistema e como parte da especificação dos requisitos. O ponto forte desse tipo de documento é que ele não se torna ultrapassado, já que ele é gerado dinamicamente conforme os testes são executados. Eles também dificilmente irão apresentar erros, visto que se algum requisito não for mais atendido, será possível visualizar no documento.

2.7. Elixir

Elixir é uma linguagem de programação que utiliza o paradigma funcional. O Elixir utiliza a máquina virtual Erlang para que possa usufruir dos benefícios desta, entre eles: fornecer aplicações distribuídas, distribuir aplicativos que rodam em tempo real, fornecer sistemas altamente tolerante a falhas. Todas essas características fazem com que a linguagem forneça aplicações altamente escaláveis. (ELIXIR, 2019)

2.7.1. ExUnit

ExUnit é o framework de testes padrão para a linguagem Elixir. O framework inclui todas as funcionalidades necessárias para a escrita de testes unitários, como asserção de resultados, pattern matching, chamadas assíncronas, entre outros. Além disso, podemos realizar diversas configurações no ambiente de testes, graças ao ExUnit. (ELIXIR SCHOOL, 2019)

2.8. Sistema de pagamentos Brasileiro

O sistema de pagamentos Brasileiro (SPB) é composto por um conjunto de entidades, sistemas e procedimentos que realizam o processamento e liquidação de operações bancárias de transferência de fundos, operações com moedas estrangeiras ou com ativos financeiros. (BANCO CENTRAL DO BRASIL, 2018)

A principal finalidade do SPB é o de permitir que as instituições financeiras possam intermediar as operações financeiras entre pessoas físicas, pessoas jurídicas,

órgãos governamentais e o Banco Central (BACEN). Por exemplo, as operações que envolvem cheques, cartões de crédito e transferência interbancária utilizam o sistema SPB. (BANCO CENTRAL DO BRASIL, 2018)

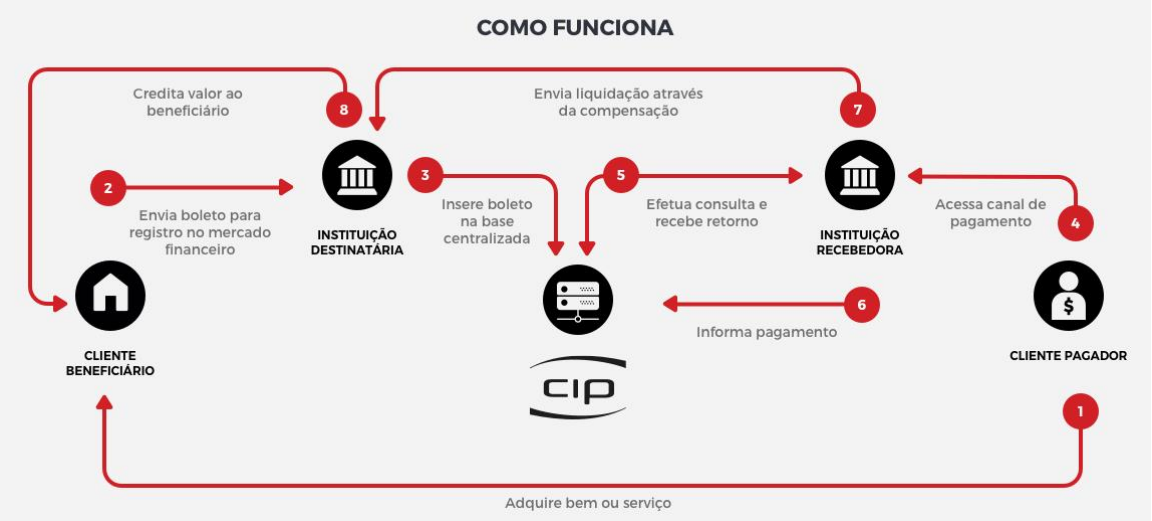
2.8.1. Boleto de pagamento

Boleto bancário – ou boleto de pagamento – é um documento utilizado para realizar a cobrança de dívidas provenientes de qualquer natureza. O documento possui uma padronização para que possa ser aceito em todas as instituições financeiras regulamentadas pelo Banco Central. As operações de liquidações dos boletos são transacionadas através do SPB. (BANCO CENTRAL DO BRASIL, 2018)

2.8.2. Plataforma Centralizada de Recebíveis

A Plataforma Centralizada de Recebíveis (PCR) é uma implementação criada pela Câmara Interbancária de Pagamentos (CIP) e funciona como uma base centralizada dos dados dos boletos de pagamento. As instituições financeiras que são credenciadas pelo Banco Central (BACEN) e pela Federação Brasileira de Bancos (FEBRABAN) podem incluir os dados de seus boletos nessa base, assim como podem altera-los. As instituições também devem realizar a consulta das informações inseridas pelas outras instituições financeiras. (CIP, 2017)

Figura 5 - Funcionamento da Plataforma centralizada de recebíveis



Fonte: CIP, 2017

O principal objetivo dessa plataforma é a de gerar mais segurança para quem está emitindo os títulos, para quem está pagando e para a instituição que está recebendo o pagamento. (FEBRABAN, 2018)

Outro objetivo da nova plataforma é a de modernizar a forma em que o pagamento de boletos é processado. Para isso, é realizada a troca de informações entre as instituições financeiras e a base centralizadora, o que permite que os dados dos boletos sejam validados antes da efetivação do pagamento, evitando que fraudes aconteçam. (CAIXA ECONÔMICA FEDERAL, 2018)

As trocas de informações entre as instituições financeiras e a PCR são realizados através da troca de arquivos XML criptografados que devem obedecer a uma estrutura estabelecida pela CIP. Essas estruturas estão descritas nos manuais disponibilizados por ela.

3. METODOLOGIA

O método de pesquisa, segundo Wazlawick (2009, p. 40), “consiste na sequência de passos necessários para demonstrar que o objetivo proposto foi atingido [...]”. Ainda segundo ele, a metodologia é uma etapa fundamental da pesquisa científica, que deve ser definida logo após que o objetivo foi definido.

O levantamento de conteúdo para formulação do referencial teórico e para a apresentação das ferramentas que serão utilizadas na aplicação da pesquisa, foi feito utilizando o procedimento de pesquisa bibliográfica. A pesquisa bibliográfica é realizada a partir da extração de informação em materiais que já foram publicados. Entre os materiais que podem ser pesquisados, estão documentos, sites da internet, livros, relatórios, periódicos e revistas. (FONTELLES, SIMÕES, *et al.*, 2009)

A metodologia utilizada foi a qualitativa, que segundo Fontelles, Simões, et al. (2009) é o tipo de pesquisa que mais se encaixa com o estudo de caso. Ainda segundo os autores, a metodologia qualitativa não deve considerar resultados numéricos ou estatísticos que os resultados possam vir a ter.

A instituição na qual o estudo de caso será aplicado, é uma cooperativa de crédito com presença muito forte na região norte do Brasil, e que desde 2013 oferece aos seus cooperados a opção de pagamento de boletos por meio de seus próprios canais de atendimento.

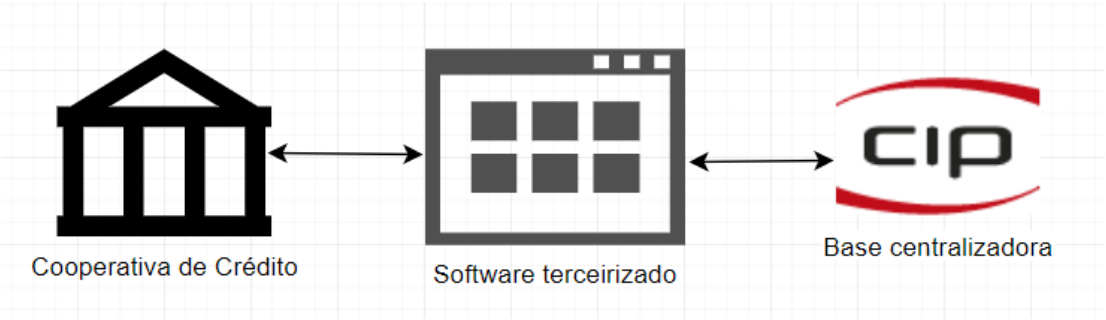
Dentre os canais de atendimento, destacam-se o Internet Banking que está disponível através de aplicativo para sistemas Android e iOS, e também disponível através do site. Outro canal em destaque são as plataformas de caixa, onde são disponibilizados os atendimentos presenciais através das agências espalhadas pelo estado de Rondônia, Acre e Pará.

A cooperativa de crédito utiliza o software de uma empresa parceira que realiza a intermediação entre a CIP e a cooperativa. O sistema da cooperativa se comunica com o software terceiro através de comunicação SOAP, informando os dados de cada pagamento realizado em seus canais e o software intermediador se encarrega de realizar o envio dos dados para a PCR.

O uso do software dessa empresa faz com que a comunicação entre a cooperativa e a PCR seja mais simples, pois a cooperativa não precisa gerar os XML’s e enviá-los diretamente à CIP. O sistema da cooperativa precisa apenas enviar os dados dos pagamentos para o software, e então o software realiza a montagem do XML e se encarrega de enviá-los.

O uso do software intermediador simplifica também o processo de emissão dos boletos de pagamento, pois ele também realiza a integração dos dados dos boletos da cooperativa com a CIP.

Figura 6 - Fluxo de integração com a CIP



Fonte: Próprio autor

A CIP frequentemente realiza adequações em suas funcionalidades e, por isso, é recorrente que a empresa que fornece o software disponibilize atualizações que corrijam ou implementem novas funcionalidades. Depois que uma atualização é disponibilizada para a cooperativa, ela primeiro aplica a atualização em um ambiente homologatório, para que testes sejam realizados tanto na funcionalidade de emissão de boletos para pagamento, quanto na funcionalidade de recebimento de boletos. Esses testes são realizados manualmente, necessitando de muita atenção aos procedimentos que são executados.

Os testes consistem em:

1. Realizar consulta de boleto na CIP;
2. Realizar o pagamento do boleto;
3. Verificar se o boleto foi devidamente baixado na CIP;
4. Realizar a consulta da baixa operacional;
5. Realizar o cancelamento do pagamento;
6. Verificar se o boleto está aberto novamente na CIP;

Os testes serão automatizados utilizando a ferramenta ExUnit, ferramenta open-source que auxilia na elaboração e execução de testes unitários para a linguagem Elixir.

Os resultados serão reunidos a partir das observações realizadas durante todo o processo de automatização, desde o planejamento até a execução dos procedimentos de teste.

4. RESULTADOS

Durante o processo de elaboração dos casos de testes unitários, foram realizadas as seguintes observações:

- **Erros no software terceirizado**

Durante a elaboração dos testes unitários, houveram erros na configuração do software que realiza a comunicação com a CIP. Esse erro ocorreu devido uma configuração incorreta no ambiente de homologação, o que fez com que fosse necessário entrar em contato com o suporte do software para que eles regularizassem os problemas e ele voltasse a funcionar. Isso mostra o quão dependente a aplicação se torna desse software, e o quanto se torna dependente da equipe de suporte da empresa que fornece o software para resolver quaisquer problemas relacionados à ele.

- **Testar funções de data**

Como implementar o teste de uma função que retorna a data atual do sistema? Por exemplo, como verificar que uma função que retorna a data atual com precisão de milissegundos está correta? A única solução encontrada foi a de praticamente reimplementar a função nos testes unitários, e realizar a verificação de que as duas datas estão iguais. Durante a implementação desse teste em específico, paramos para pesquisar o que os outros desenvolvedores estavam utilizando para realizar os testes com esse tipo de função, e não foram encontradas soluções práticas.

Uma das soluções encontradas foi a de alterar a data do computador em que o teste está sendo executado, o que não é muito eficaz, pois não temos controle de em quais ambientes os testes estão sendo executados – se é um servidor Linux, ou dentro de um container Docker, ou na própria máquina do desenvolvedor.

- **Banco de dados de teste**

Na cooperativa de crédito onde os testes foram implementados, existem três ambientes de banco de dados: desenvolvimento, homologação e produção. O ambiente de produção, por conter dados sigilosos e conter informações atualizadas dos cooperados, não pode ser utilizado para executar os testes.

O ambiente de homologação é atualizado semanalmente, e por isso os dados constantemente estão sendo atualizados, o que faz com que os testes passassem a

ser executados com erro, ora por que os registros já não existiam mais, ora por que eles haviam sido alterados por outro sistema.

O ambiente de desenvolvimento é semelhante ao de homologação, onde todos os desenvolvedores têm acesso para modificação dos dados, o que faz com que os dados utilizados para testar o software fossem constantemente alterados, e os testes passassem a falhar.

A solução encontrada foi de criar um banco de dados dinamicamente conforme os testes eram executados. No momento de execução dos testes, scripts SQL eram executados e os registros inseridos em uma base local, e logo após a execução esse banco de dados era apagado.

- **Demora para informações serem persistidas na CIP**

Pelo fato de os testes estarem sendo executados em ambiente de homologação da CIP, foi sentido um certo atraso na recepção das informações que foram enviadas por nós. Isso fez com que o processo de execução dos testes fosse aumentado consideravelmente, pois em determinados testes unitários foi necessário criar um mecanismo de espera, até que essas informações fossem persistidas na CIP.

- **Queries específicas precisam de registros específicos**

Para completar o teste das queries, em alguns casos foi necessário que fosse inserido um novo registro apenas para que uma determinada query pudesse ser executada corretamente. Isso fez com que os registros de testes que foram inseridos crescessem significativamente.

- **Não é possível criar mocks[1] para registros fora do domínio do software**

Nos módulos onde os registros de outros sistemas são utilizados, não foi possível criar mocks desses registros. Isso porque as chamadas dos módulos externos estão encapsuladas dentro das funções do sistema, não permitindo com que esses registros pudessem ser manipulados pelos scripts de testes.

[1] Mocks são registros criados apenas para fins de testes. Esses registros simulam dados reais.

- **Reverter registros do banco de dados ou configurar para que o teste possa ser rodado várias vezes**

Determinadas funções do sistema alteram e/ou inserem registros no banco de dados, e isso faz com que os testes de outras funcionalidades fossem impactados por essas alterações. As soluções encontradas foram:

- **Configurar banco de dados como sandbox[2]:**
  Com essa configuração, ao final dos testes as informações não seriam persistidas na base, e os registros seriam revertidos automaticamente. Essa abordagem era a mais recomendada, porém a funcionalidade não funcionou como o esperado, fazendo com que os registros fossem persistidos mesmo assim.
- **Reverter as alterações do banco de dados:**
  Outra solução foi reverter manualmente os registros no banco de dados, voltando-os ao estado original nos casos onde foram alterados, e excluindo os registros nos casos onde eles foram inseridos. Embora menos recomendada e mais demorada, foi a solução aplicada no cenário estudado.

- **Implementar 100% de cobertura de testes**

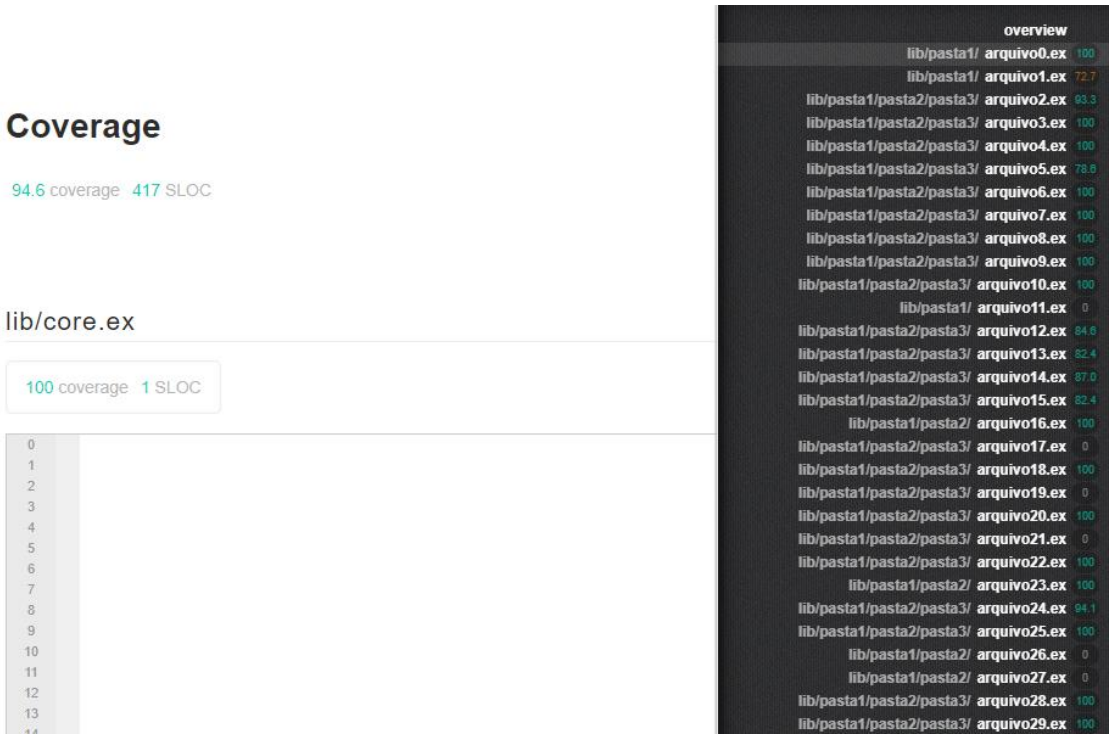
Embora fosse o objetivo original, não foi possível cobrir 100% de testes o software devido aos obstáculos apresentados acima, e também pelo curto tempo de implementação dos casos de teste. Devido ao tempo fornecido para realização dos testes, não foi possível estudar a fundo a ferramenta ExUnit para entender melhor seu funcionamento, nem explorar todas suas configurações possíveis para extrair o potencial completo da ferramenta.

4.1. Relatório Coveralls

A ferramenta de testes ExUnit acompanha um módulo exclusivo para a geração de um relatório de cobertura de testes. Dentre as opções de configuração deste módulo, está disponível configurar o relatório em formato HTML, formato esse que foi escolhido para a geração neste projeto.

[2] Sandbox é um modo de manipulação do banco de dados, onde os registros não serão persistidos ao final dos testes.

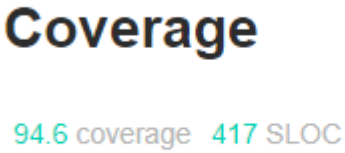
Figura 7 - Relatório de cobertura de testes



Fonte: Próprio autor

O relatório apresenta diversas informações, como porcentagem de cobertura de testes geral do projeto, e também a cobertura individual de cada arquivo que pertence a ele. No relatório é possível visualizar cada arquivo individualmente, observando quais linhas do código precisam ser testadas, quais foram testadas e quais não foram.

Figura 8 - Cobertura total do projeto

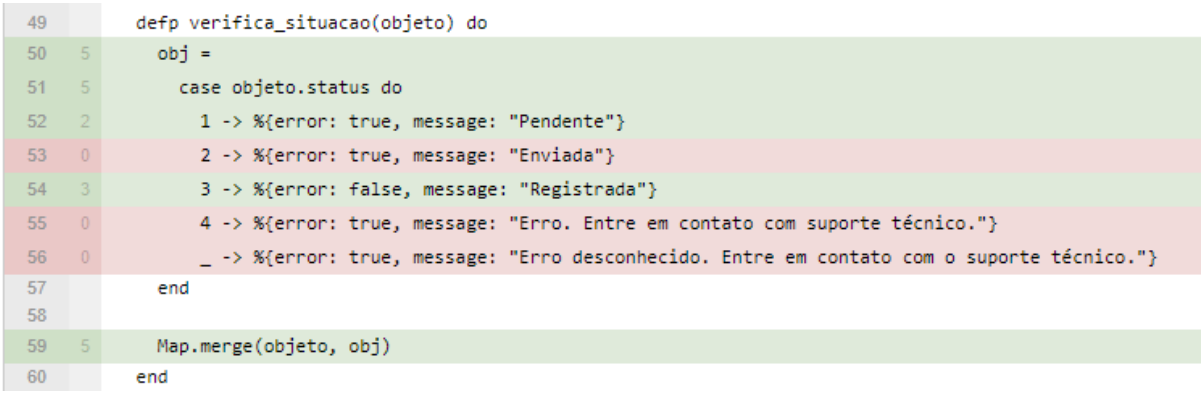


Fonte: Próprio autor

Na figura 8, podemos identificar 2 dados. A porcentagem de cobertura total do projeto (94,6%) e a quantidade de linhas de código que precisam ser testadas (417

SLOC[3]). É importante salientar que, SLOC nesse caso não é a quantidade de linhas de código totais do projeto, mas sim a quantidade de linhas que precisam ser testadas.

Figura 9 - Linhas testadas e não testadas



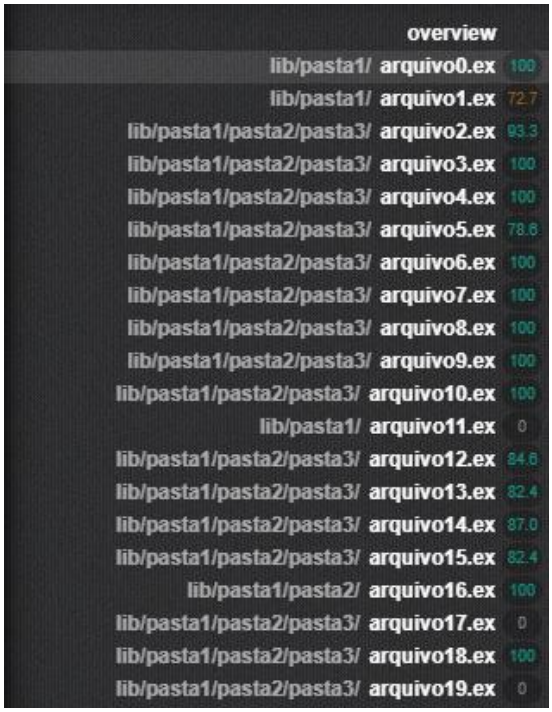
Fonte: Próprio autor

A figura 9 mostra um trecho de código de um determinado arquivo do projeto. Esse trecho foi disponibilizado pelo relatório para que possa ser identificado quais linhas precisam ser testadas. No exemplo da figura, as linhas em branco são linhas que não precisam ser testadas. As linhas na cor verde são linhas que foram testadas, e as linhas na cor vermelha identificam trechos do código em que os testes não foram implementados.

A figura 9 ilustra também uma das dificuldades encontradas ao cobrir com testes todos os arquivos do projeto. Determinadas clausulas não poderiam ser cumpridas com dados disponibilizados no ambiente de homologação, pois os dados neste ambiente não contemplam todas as funcionalidades propostas no projeto.

[3] Source Line Of Code: Em tradução literal, significa linhas de código fonte.

Figura 10 - Cobertura de teste por arquivo



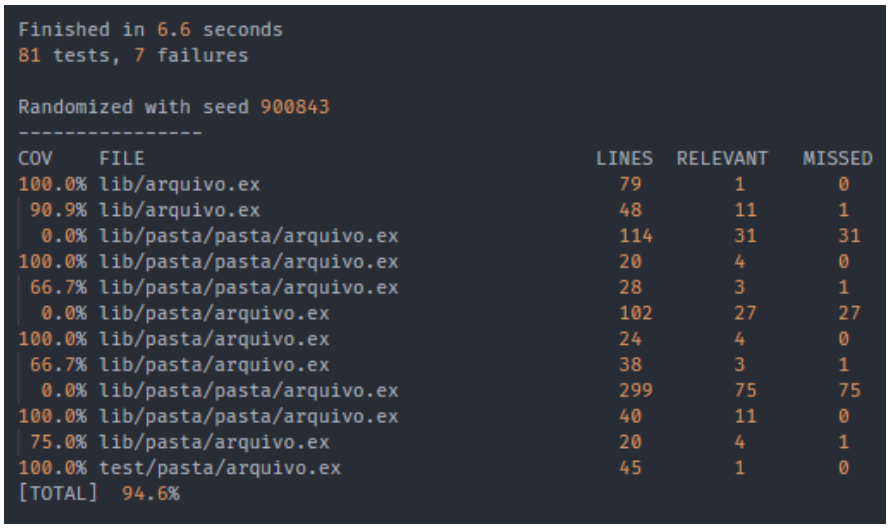
Fonte: Próprio autor

Podemos observar na figura 10 a apresentação de todos os arquivos do projeto, com sua respectiva estrutura de pastas na qual está contido, e também a porcentagem de teste alcançada neste arquivo.

Apesar da configuração padrão escolhida ser o formato HTML, ainda é possível obter o relatório em formato de texto. Esse formato de saída do relatório é útil, pois com ele é possível realizar a integração com outras ferramentas, como por exemplo uma ferramenta de deploy[4] automatizado, que só realiza o deploy caso os testes sejam executados com sucesso. No formato HTML, capturar esse dado seria mais difícil.

[4] Deploy: disponibilizar um sistema para uso, seja num ambiente de desenvolvimento, para testes ou em produção

Figura 11 - Relatório em formato de texto



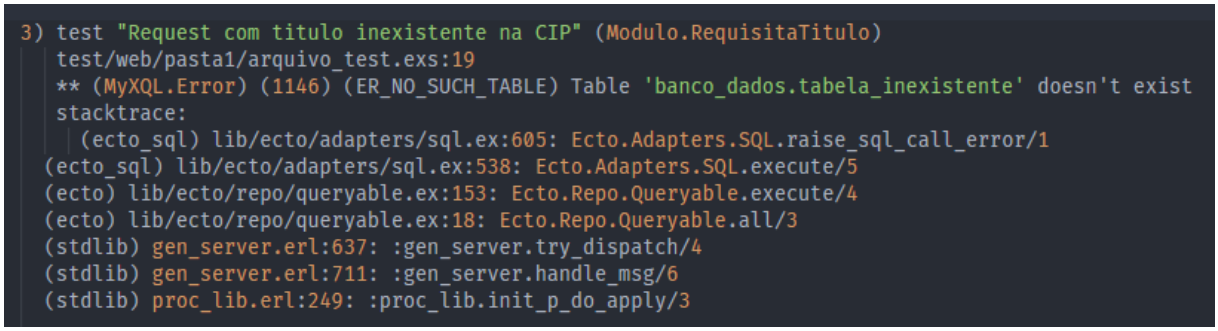
```
Finished in 6.6 seconds
81 tests, 7 failures

Randomized with seed 900843
----------------
COV    FILE                                        LINES  RELEVANT   MISSED
100.0% lib/arquivo.ex                                79         1        0
 90.9% lib/arquivo.ex                                48        11        1
  0.0% lib/pasta/pasta/arquivo.ex                   114        31       31
100.0% lib/pasta/pasta/arquivo.ex                    20         4        0
 66.7% lib/pasta/pasta/arquivo.ex                    28         3        1
  0.0% lib/pasta/arquivo.ex                         102        27       27
100.0% lib/pasta/arquivo.ex                          24         4        0
 66.7% lib/pasta/arquivo.ex                          38         3        1
  0.0% lib/pasta/pasta/arquivo.ex                   299        75       75
100.0% lib/pasta/pasta/arquivo.ex                    40        11        0
 75.0% lib/pasta/arquivo.ex                          20         4        1
100.0% test/pasta/arquivo.ex                         45         1        0
[TOTAL]  94.6%
```

Fonte: Próprio autor

O relatório apresenta as mesmas informações do relatório em formato HTML, como porcentagem total de cobertura de testes e a porcentagem de cobertura de cada arquivo. Além dessas informações, apresenta a quantidade de testes que foram executados, quantos erros ocorreram na execução e o tempo que o teste levou para ser executado.

Quando um erro ocorre durante a execução, é exibido no log do console, e também é disponibilizado no relatório em texto no formato abaixo.

Figura 12 - Erro ocorrido durante teste



```
3) test "Request com titulo inexistente na CIP" (Modulo.RequisitaTitulo)
   test/web/pasta1/arquivo_test.exs:19
   ** (MyXQL.Error) (1146) (ER_NO_SUCH_TABLE) Table 'banco_dados.tabela_inexistente' doesn't exist
   stacktrace:
     (ecto_sql) lib/ecto/adapters/sql.ex:605: Ecto.Adapters.SQL.raise_sql_call_error/1
   (ecto_sql) lib/ecto/adapters/sql.ex:538: Ecto.Adapters.SQL.execute/5
   (ecto) lib/ecto/repo/queryable.ex:153: Ecto.Repo.Queryable.execute/4
   (ecto) lib/ecto/repo/queryable.ex:18: Ecto.Repo.Queryable.all/3
   (stdlib) gen_server.erl:637: :gen_server.try_dispatch/4
   (stdlib) gen_server.erl:711: :gen_server.handle_msg/6
   (stdlib) proc_lib.erl:249: :proc_lib.init_p_do_apply/3
```

Fonte: Próprio autor

O detalhamento do erro apresenta qual o nome do teste em que o erro ocorreu, o módulo em que o arquivo está contido e o stacktrace[5] do erro.

[5] Relatório de execução do sistema em determinado período. Geralmente é exibido quando ocorre um erro.

5. CONCLUSÃO

O processo de desenvolvimento de testes tem como principal objetivo melhorar a qualidade do software. Essa melhoria se dá por três motivos principais: garantindo que os requisitos funcionais e não-funcionais especificados em seu projeto estejam sendo cumpridos, testando a performance do software e expondo os erros existentes nele. Essa melhoria se dá principalmente através dos processos de verificação e validação.

Existem diversos tipos de testes, sendo o teste unitário o mais utilizado pelo fato de que é o mais simples de ser planejado e executado. Todo o processo de desenvolvimento de testes deve ser planejado antes de começar a ser implementado, garantindo uma cobertura mais eficaz dos módulos do software.

O processo de automatização de testes é um fator primordial na garantia da qualidade, visto que ações manuais passam a ser menos frequentes nos processos de verificação de requisitos ou de verificação de performance. A equipe de desenvolvimento ganha confiança para realizar melhorias no código, sem se preocupar com quebras de contrato entre funções e módulos do sistema.

Diante do levantamento dessas informações, foi realizado o planejamento dos testes de um sistema de pagamento de boletos bancários. Foi definido que o projeto teria 100% de cobertura de testes e que os testes seriam automatizados utilizando a ferramenta ExUnit. Porém durante o seu desenvolvimento, foram identificadas algumas dificuldades que impediram o cumprimento desta meta. Dentre essas dificuldades, destacam-se os erros no software que realiza a comunicação com a CIP, que impediu a execução dos testes até que os erros fossem resolvidos, além de não ter sido possível desenvolver outros testes que dependiam dessa funcionalidade. Outro ponto observado durante o desenvolvimento de testes, foi a lentidão na persistência dos dados na CIP. Isso fez com que fosse necessário implementar mecanismos de espera até que essas informações estivessem persistidas e disponíveis para consulta.

Foi observado que as ferramentas ExUnit e ExCoveralls funcionam muito bem em conjunto, complementando as funcionalidades uma da outra. O ExUnit oferece funções para realizar aferições no código, enquanto o ExCoveralls oferece relatórios com informações muito precisas, que podem ajudar a identificar onde o software não está sendo testado. Além dessas informações, também oferece dados como

quantidade de erros ocorridos durante a execução de testes e o tempo levado para execução.

Com base nas informações contidas neste documento, a Cooperativa pode terminar a implementação dos testes sem muitas dificuldades. Levando em conta as dificuldades percebidas durante todo o processo, é possível adotar uma estratégia diferente para a implementação. Portanto, os objetivos principais propostos para este trabalho foram cumpridos.

REFERÊNCIAS

ASSOCIAÇÃO BRASILEIRA DE NORMAS TÉCNICAS. **Engenharia de software - Qualidade de produto**. Rio de Janeiro, p. 21. 2003.

BANCO CENTRAL DO BRASIL. Relatório Cidadania Financeira, 2018. Disponivel em: <https://www.bcb.gov.br/content/cidadaniafinanceira/Documents/RIF/Relatorio%20Cidadania%20Financeira_BCB_16jan_2019.pdf>. Acesso em: 1 jun. 2019.

BANCO CENTRAL DO BRASIL. Sistema de Pagamentos Brasileiro (SPB). **Banco Central do Brasil**, 2018. Disponivel em: <https://www.bcb.gov.br/estabilidadefinanceira/spb>. Acesso em: 1 jun. 2019.

BANDEIRA, G. et al. O PROCESSO DE TESTE DE SOFTWARE. **Tecnologias em Projeção**, Brasilia, 7, 2016. 10. Disponivel em: <http://revista.faculdadeprojecao.edu.br/index.php/Projecao4/article/viewFile/704/623>. Acesso em: 21 maio 2019.

BARTIÉ, A. **Garantia da qualidade de software:** As melhores práticas de engenharia de software aplicadas à sua empresa. Rio de Janeiro: Editora Campus, 2002.

BERNARDO, P. C. **Padrões de testes automatizados**. Universidade de São Paulo. São Paulo, p. 197. 2011.

CAIXA ECONÔMICA FEDERAL. Caixa. **Nova Plataforma de Cobrança Bancária**, 2018. Disponivel em: <http://www.caixa.gov.br/empresa/pagamentos-recebimentos/recebimentos/nova-cobranca-bancaria/Paginas/default.aspx>. Acesso em: 30 maio 2019.

CARLINE, C. **TESTE DE SOFTWARE: UMA NECESSIDADE DAS EMPRESAS**. UNIJUÍ. Santa Rosa, p. 91. 2012.

CAUZZI, E. J. **Proposta de plano de garantia da qualidade de software para o laboratório de criação e aplicação de software**. UNIVERSIDADE DE CAXIAS DO SUL. Caxias do sul, p. 141. 2015.

CIP. CIP - Soluções e Serviços. **Plataforma Centralizada de Recebíveis**, 2017. Disponivel em: <https://www.cip-bancos.org.br/Paginas/PCR.aspx>. Acesso em: 29 maio 2019.

DELAMARO, M.; MALDONADO, J. C.; JINO, M. **Introdução ao teste de software**. Rio de janeiro: Campus, 2007.

DINIZ, D. Uma abordagem para documentação de Testes de Software baseado no IEEE 829-1998. **Linha de Código**, 2008. Disponivel em: <http://www.linhadecodigo.com.br/artigo/1807/uma-abordagem-para-documentacao-de-testes-de-software-baseado-no-ieee-829-1998.aspx>. Acesso em: 20 maio 2019.

ELIXIR. **Elixir**. Disponivel em: <https://elixir-lang.org/>. Acesso em: 22 Outubro 2019.

ELIXIR SCHOOL. **Elixir School**. Disponivel em: <https://elixirschool.com/pt/lessons/basics/testing/>. Acesso em: 22 Outubro 2019.

FEBRABAN. FEBRABAN. **Nova Plataforma de Boletos de Pagamento-Cobrança Registrada**, 2018. Disponivel em: <https://portal.febraban.org.br/pagina/3150/1094/pt-br/servicos-novo-plataforma-boletos>. Acesso em: 30 maio 2019.

FONTELLES, M. J. et al. **METODOLOGIA DA PESQUISA CIENTÍFICA: DIRETRIZES PARA A ELABORAÇÃO DE UM PROTOCOLO DE PESQUISA**. Belém, p. 8. 2009.

FURLAN, F. P. **VISUALIZAÇÃO DE INFORMAÇÃO COMO APOIO AO PLANEJAMENTO DO TESTE DE SOFTWARE**. UNIVERSIDADE METODISTA DE PIRACICABA. Piracicaba, p. 140. 2009.

MORENO, L. A Importância da validação e da verificação. **Dev Media**, 2012. Disponivel em: <https://www.devmedia.com.br/a-importancia-da-validacao-e-da-verificacao/24559>. Acesso em: 22 maio 2019.

MYERS, G. J. **The Art of Software Testing**. 2ª. ed. Hoboken: John Wiley & Sons, 2004.

NOBIATO, A. et al. **Uma Metodologia para Teste de Software no Contexto da Melhoria de Processo**. Campinas, p. 15. 2004.

PAN, J. **Software Testing**. Carnegie Mellon University. Pittsburgh, p. 12. 1999.
PRESSMAN, R. S. **Engenharia de Software:** Uma abordagem profissional. 7ª. ed. São Paulo: Pearson Makron Books, 2011.

SCHACH, S. R. **Object-Oriented Software Engineering**. 7ª. ed. [S.l.]: McGraw-Hill Education, 2007.

SOMMERVILLE, I. **Engenharia de Software**. 9ª. ed. [S.l.]: PEARSON & ARTMED, 2011.

VICENTE, A. A. **Definição e gerenciamento de métricas de teste no contexto de métodos ágeis**. Universidade de São Paulo. São Carlos, p. 136. 2010.

WAZLAWICK, R. S. **Metodologia de Pesquisa para Ciência da Computação**. Rio de Janeiro: Elsevier, 2009. 163 p.