UNIVERSIDADE DO VALE DO TAQUARI - UNIVATES

CURSO DE SISTEMAS DE INFORMAÇÃO

ANÁLISE COMPARATIVA DE TESTES ESTRUTURADOS E TESTES EXPLORATÓRIOS DE ESPECIALISTAS

Fernanda Elisa Finkler

Lajeado, novembro de 2017

Fernanda Elisa Finkler

ANÁLISE COMPARATIVA DE TESTES ESTRUTURADOS E TESTES EXPLORATÓRIOS DE ESPECIALISTAS

Monografia apresentada ao Centro de Ciências Exatas e Tecnológicas da Universidade do Vale do Taquari - UNIVATES, como parte da exigência para a obtenção do título de bacharel em Sistemas de Informação.

Orientador: Prof. Me. Fabrício Pretto

Lajeado, novembro de 2017

AGRADECIMENTOS

Ao meu professor e orientador Fabrício Pretto, agradeço por todas as horas dedicadas, sejam em orientações ou em conversas informais sobre nosso mundo da Tecnologia da Informação e, principalmente, pelos ensinamentos nas diversas disciplinas ministradas no curso de Sistemas de Informação.

Aos demais professores, responsáveis pela construção do meu conhecimento ao longo dos onze anos de universidade.

Aos meus colegas de aula e de trabalho, pelas experiências trocadas, trabalhos em grupo e pelos auxílios, em especial ao grupo Cadeira de Hoje que tornou os últimos semestres mais leves e descontraídos.

À Interact Solutions pela oportunidade de atuar no mercado de trabalho na área de TI e pelas facilidades com horários de expediente que permitiram a execução e finalização deste trabalho de conclusão de curso.

Ao amigo Artur Tomasi que me auxiliou em suas horas vagas na aprendizagem de programação avançada em momentos cruciais da minha evolução no curso.

Ao meu namorado, Diego Alex Schossler, que pacientemente me acompanhou nesses onze anos de curso e de namoro.

E à minha família que é a base da minha existência e me proporcionou a educação básica e princípios para alcançar este e todos os objetivos da minha vida.

RESUMO

O mercado de software atual está em constante busca por otimização em seus processos e maior qualidade em seus produtos. A Engenharia de Software sugere vários modelos de processos e boas práticas. Entre as fases dos processos está a de Testes de Software, etapa que contribui consideravelmente para a qualidade dos sistemas. Muitas empresas realizam o teste de software de forma manual e não automatizada, causando maior investimento em mão de obra e uma necessidade por busca de aperfeiçoamento para essa forma de teste. Outro fator relacionado aos testes manuais diz respeito à dificuldade de padronização e garantia de completude e abrangência. A dependência do fator humano pode trazer ganhos no que se refere a levantar problemas mais específicos, no entanto, não garante uniformidade. Diante disso, este trabalho apresenta uma análise que compara testes manuais guiados e estruturados utilizando a técnica de *checklist* com os testes não estruturados realizados por testadores especialistas.

Palavras-chave: Testes de Software. Qualidade de Software. *Checklist*. Métricas.

ABSTRACT

The current software market is in constant pursuit for process optimization and greater product quality. Software Engineering proposes several process models and best practices. Between the processes phases is Software Testing, a stage that substantially contributes to the software quality. Many companies perform the software testing manually instead of automatically, causing a bigger investment in workforce and a need to seek improvement for this kind of testing. Another point related to manual testing concerns the difficulty of standardization and assurance of completeness and comprehensiveness. The dependence of human factor can bring gains in finding more specific problems, however, it does not guarantee uniformity. Therefore, this work presents an analysis that compares guided and structured manual tests using the checklist technique with unstructured tests performed by expert testers.

Keywords: Software Testing. Software Quality. Checklist. Metrics.

LISTA DE FIGURAS

LISTA DE GRÁFICOS

LISTA DE QUADROS

LISTA DE ABREVIATURAS E SIGLAS

IC	Integração Contínua
CMMI	Capability Maturity Model Integration
CRC	Classe-Responsabilidade-Colaborador
CVS	Concurrent Version System
EPG	Engineering Process Group
FP	Function point
FS	Fábrica de Software
ITG	Independent test group
KISS	Keep it simple, stupid!
RC	Release Candidate
SVN	Subversion
TDD	Test Driven Development
TS	Testes de Software
XP	Extreme Programming

SUMÁRIO

1 INTRODUÇÃO

Diante da alta competitividade no mercado de software atual, percebe-se cada vez mais movimentos das empresas na busca de melhorias e aperfeiçoamento de suas práticas de criação de software. Essa concorrência faz com que a Qualidade de Software seja apenas um pré-requisito dos sistemas e não mais um diferencial.

Bartié (2002, p.16) citou que “qualidade não é uma fase do ciclo de desenvolvimento de software [...] é parte de todas as fases". Muito ainda se associa Qualidade de Software com testes. Porém, testes são apenas uma das várias etapas pelas quais um sistema passa até sua liberação.

A Qualidade de Software é uma área de conhecimento dentro da Engenharia de Software. Na literatura, muito se encontra sobre o processo de desenvolvimento de software, todas as etapas e atividades ideais. Modelos de processo foram elaborados para guiar a criação de software e, logo, também surgiram os manuais de boas práticas, definição de estratégias, entre outros artefatos que, se adotados com vigor, são um caminho para garantir a qualidade dos sistemas (PRESSMAN; MAXIM, 2016).

Os Testes de Software são realmente uma das principais formas para contribuir na melhoria da qualidade dos sistemas. Eles buscam identificar os defeitos e a conformidade com as exigências específicas de mercado. Conforme Barbosa e Torres (2012), o teste ocupa em torno de 40% do tempo planejado para um projeto, enquanto um erro encontrado na fase de implantação pode provocar um aumento de

60% nos custos de um projeto. Isto confirma a importância dos Testes de Software e a necessidade de dedicar o devido tempo e atenção a eles, buscando também formas de aperfeiçoá-los.

Existem diversos tipos e metodologias que visam facilitar o processo de testes. Sua utilização auxilia a diminuir ao máximo possível os erros, entretanto, nenhum tipo garante a total ausência deles.

A automatização de atividades que integram o processo de desenvolvimento de software é muito defendida. É recomendado automatizar desde rotinas que integram a fase de desenvolvimento dos sistemas, até a execução de testes. No entanto, Papo (2010) afirma que o que efetivamente encontra erros é o teste manual, sendo que 70% dos testes no mercado são manuais. Os testes automatizados servem basicamente para o tipo de teste Regressão, que nada mais é do que a necessidade de reaplicação de um determinado teste diversas vezes.

Atualmente as empresas já empregam mais investimentos para contratar pessoas especificamente para realização de Testes de Software. Algumas chegam a montar equipes mais estruturadas com foco total em qualidade de sistemas; ou, ainda, os *Engineering Process Groups* (EPG), grupos de trabalho focados em melhorias de processo.

Como a grande maioria dos testes ocorre de forma manual, algumas características de perfil dos testadores podem ser importantes, como: ser criativo, detalhista, perfeccionista e explorador. Tendo em vista que o papel do testador é fundamentalmente apontar defeitos, muitas vezes podem ser necessárias características como ser persuasivo e negociador, pois os defeitos acabam sendo causados por pessoas, geralmente os desenvolvedores, desta forma, pode ser necessário entrar em discussão para chegar a um comum acordo.

Com a intenção de analisar os Testes de Software manuais, este trabalho se propõe a desenvolver um estudo comparativo sobre testes realizados de forma estruturada, organizados via *checklist*, com os testes de especialistas realizados de forma exploratória de acordo com a experiência destes.

1.1 Definição do problema

O presente estudo vai ao encontro das dificuldades das empresas em organizar a fase de Testes de Software com pessoas dedicadas e equipe específica para empenhar mais tempo na busca por maior qualidade nos produtos. Mesmo realizando este investimento, ainda existe uma dificuldade em medir a qualidade dos seus produtos e ter clareza se os métodos empregados são adequados.

Este trabalho pretendeu buscar uma forma de ter um resultado quantitativo para a Qualidade de Software no que tange a medição de qualidade de produto por meio de testes funcionais e avaliar os ganhos com testes mais formais. A falta de um roteiro dificulta a replicação de testes, visto que não há uma referência a ser seguida e depende principalmente da leitura que o testador faz da demanda. Sendo assim, fatores externos podem afetar a forma e completude dos testes, em razão de que o estado emocional e preocupações particulares e/ou profissionais afetam diretamente a concentração e produtividade.

Ainda, neste formato exploratório, não há como realizar a medição do nível de qualidade do software, pois cada teste tende a ser diferente, não passando por todos os mesmos passos de testes anteriores, consequentemente, não possibilitando coletar resultados e realizar comparações.

A realização deste trabalho não contempla nenhum tipo de automatização de testes de software.

1.2 Objetivos

O objetivo geral deste trabalho consiste em realizar um estudo de caso em uma empresa de software para avaliar um formato estruturado de Teste de Software, por meio da elaboração e aplicação de *checklist* e comparar com testes exploratórios realizados por especialistas.

Os objetivos específicos se dividem em:

- Realizar uma pesquisa bibliográfica na área do objeto de estudo;
- Aplicar o estudo em uma empresa já identificada;
- Elaborar um plano que seja o guia para os testes formais;
- Definir métricas para obter resultados dos testes;
- Aplicar, medir e acompanhar a realização dos testes (estruturados e não estruturados);
- Avaliar os resultados e o processo formal;
- Comparar registros de erros antigos com os registros após aplicação do processo formal.

1.3 Justificativa

Uma das motivações para a proposta deste trabalho é a forma abstrata com que se lida com o nível de Qualidade de Software, no que tange a medição da qualidade funcional de um produto. Uma grande lacuna no processo de software diz respeito a como informar a um gestor, cliente ou *sponsor* (patrocinador de projeto) qual o nível de qualidade de um sistema de forma que essa resposta retrate o estado atual do software em questão.

Esta dificuldade leva à necessidade de definir uma estrutura que abranja todos as funções de um software e sirva de guia para garantir a completude dos testes. Por outro lado, testes exploratórios que contam com testadores de largo conhecimento sobre o sistema, podem levantar problemas em caminhos difíceis de serem planejados.

Inicialmente a estruturação dos testes através de *checklists* pode parecer vantajosa. Entretanto, exige tempo para sua estruturação e, posteriormente, precisa ser constantemente revisada e atualizada. Através do estudo de caso foi possível avaliar a eficácia de cada formato, de acordo com o tempo despendido e os erros que foram encontrados.

1.4 Estrutura do Trabalho

Este trabalho está dividido em capítulos e sua ordem é definida da seguinte forma: o primeiro capítulo é de Introdução, constituído de uma preliminar do assunto abordado ao longo do trabalho, dos objetivos e desta explicação acerca da forma com que se estrutura; é sucedido pelo segundo capítulo que se trata de um levantamento bibliográfico com a revisão de vários conceitos que compõem o processo de desenvolvimento de software; por conseguinte, a metodologia de pesquisa resume o conteúdo do terceiro capítulo; o quarto apresenta os resultados encontrados no estudo de caso e uma análise sobre eles; o quinto e último capítulo finaliza o presente trabalho com as conclusões obtidas.

2 REFERENCIAL TEÓRICO

Neste capítulo serão apresentados os conceitos que fundamentam este trabalho, a disciplina em que o assunto se situa, os modelos de desenvolvimento de software e os tipos de testes de software existentes.

Com este embasamento, será possível aprofundar em testes de software e realizar uma análise dos resultados que serão encontrados em cada método de teste.

2.1 Engenharia de Software

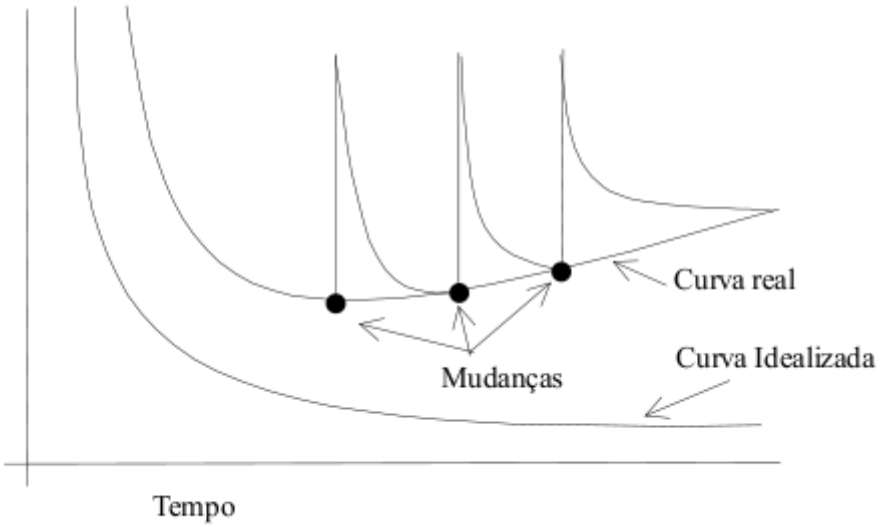
De acordo com Pressman e Maxim (2016), a Engenharia de Software é um conjunto de práticas que auxiliam os profissionais na criação de softwares, sendo uma disciplina muito importante para que o processo não se torne conturbado. No entanto, é flexível a ponto de se adaptar às necessidades de cada empresa ou profissional. O grande objetivo é sempre entregar o produto dentro do prazo e com qualidade, para satisfazer os patrocinadores e usuários do sistema.

A Engenharia de Software também é uma tecnologia em camadas onde a pedra fundamental é o foco na qualidade. As demais camadas consistem em Processo, Métodos e Ferramentas. Rezende (2005) explica que Métodos tratam de como fazer, são um norte para a construção do programa; Ferramentas são

um apoio ao como fazer, proporcionam detalhes e técnicas para sustentar os métodos; o Processo ou Procedimento, consiste no elo que forma um roteiro de atividades, definindo uma sequência lógica para o desenvolvimento do software.

O software, diferentemente do hardware, não sofre desgaste de peças e componentes, que ocorre devido a fatores externos. Logo no início da vida de um programa, sua taxa de defeitos ainda pode ser mais alta devido a erros que são encontrados e não foram vistos durante sua fase de criação. Mas estes são corrigidos e logo sua taxa de defeitos entra em declive e faz uma curva que deveria ficar achatada, conforme pode ser visto na Figura 1, na indicação de curva idealizada.

Figura 1 - Curva de defeitos para software



Fonte: Adaptado pela autora com base em Pressman e Maxim (2016, p.6)

Porém, ao longo da vida do programa, surgem demandas de alterações, e junto com a introdução dessas alterações, podem ser implementados novos erros. Estes também são corrigidos, entretanto, antes que a curva volte ao ideal, novas alterações são requisitadas. Isto indica que o software, apesar de não sofrer desgastes do tempo, se deteriora em função de sua manutenção (PRESSMAN; MAXIM, 2016).

Os requisitos de tecnologia da informação demandados por pessoas e por organizações, estão cada vez mais complexos. O que antigamente era desenvolvido por uma pessoa só, atualmente é feito por grandes equipes. Sendo assim, segundo Pressman e Maxim (2016), projetar se torna uma atividade primordial.

Além disso, o software é cada vez mais fundamental na vida das pessoas, negócios e governos, para consulta de informações, análises e controles. Caso um software falhe, podem haver desde pequenos contratempos até problemas catastróficos. Desta forma, entende-se que além de ser importante projetar a construção dos programas, primar pela sua qualidade passa a ser tão primordial quanto.

Rezende (2005), define Engenharia de Software como metodologia de desenvolvimento e manutenção de sistemas modulares, sendo que processo é uma de suas características. Uma metodologia genérica de processo de engenharia de software engloba basicamente cinco grupos de atividades (etapas):

- Comunicação: etapa onde é necessário interagir com o cliente, entender as necessidades e o objetivo do programa, para levantar os requisitos e refinar suas funções;
- Planejamento: cria um plano de projeto, que será um plano de trabalho, definindo os recursos necessários, levantado possíveis riscos, descrevendo atividades técnicas, o produto resultante e elaborando um cronograma de execução;
- Modelagem: elabora modelos que servirão de base para a etapa de construção do software; define aspectos de arquitetura e como as partes serão interligadas;
- Construção: esta etapa consiste de geração de código e testes que auxiliarão a identificar possíveis erros na codificação;
- Entrega: entrega parte ou todo software ao cliente, que, por sua vez, avalia se o produto final ficou de acordo com o solicitado e fornece *feedback*.

Para Pfleeger (2007), processo é um conjunto de tarefas ordenadas, que possuem etapas envolvendo atividades, restrições e recursos definidos para alcançar ou concluir uma determinada proposta. Além disso, são características de

processo: critérios de entrada e saída para cada atividade do processo; a possibilidade de existência de subprocessos; e, uma ordem de execução das atividades em relação às outras. No contexto de software, Pfleeger (2007, p. 38) determina “Todo processo de desenvolvimento de software tem como entrada os requisitos do sistema e como saída um produto fornecido”.

A seguir, serão revisados os processos existentes para desenvolvimento de software, em uma abordagem tradicional e outra ágil, expondo alguns exemplos de cada. Esta última surgiu no ano de 2001, conforme Pressman (2011), quando dezessete renomados desenvolvedores se reuniram para assinar o Manifesto para o Desenvolvimento Ágil de Software.

2.1.1 Modelos tradicionais

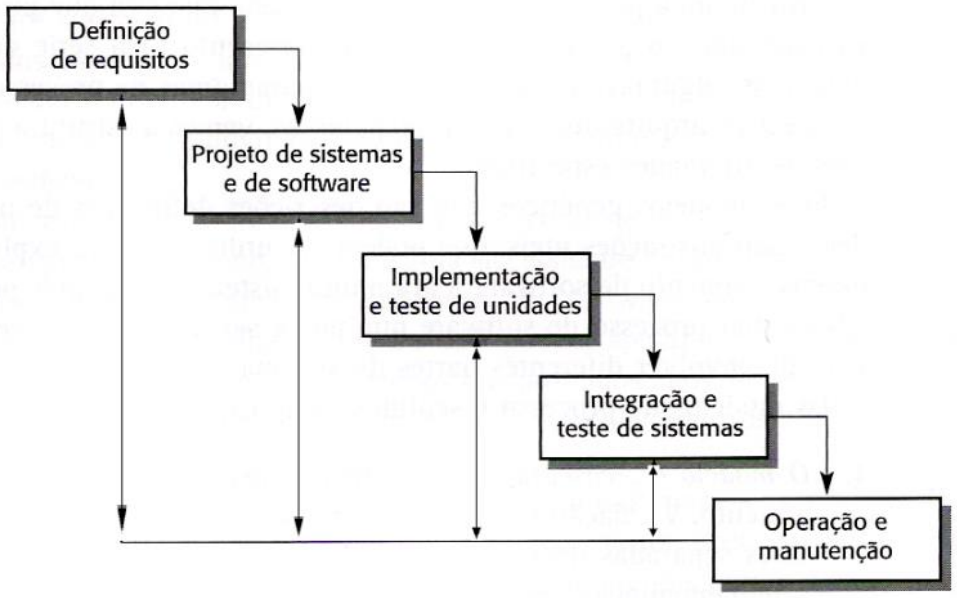
Para Sommerville (2007), cada modelo de processo representa um processo sob uma determinada perspectiva e são abstrações que podem ser usadas para explicar diferentes abordagens para o desenvolvimento de software. Já Pressman (2007), lembra que a existência de um processo de software não dá garantias de que o produto será entregue dentro do prazo esperado e que atenderá as necessidades do cliente, mas sim, deve ser combinado com práticas consistentes de engenharia de software.

São exemplos de modelos de processo de software tradicionais:

- **Modelo cascata**: compreende as atividades fundamentais de especificação, desenvolvimento, validação e evolução e as representa em fases de processo separadas: especificação de requisitos, projeto de software, implementação e teste. O nome do modelo se dá devido ao seu ciclo de vida clássico, que encadeia uma fase com a outra, onde uma fase seguinte não deve começar sem a fase anterior ter terminado. Suas fases ou estágios, se determinam conforme a Figura 2.

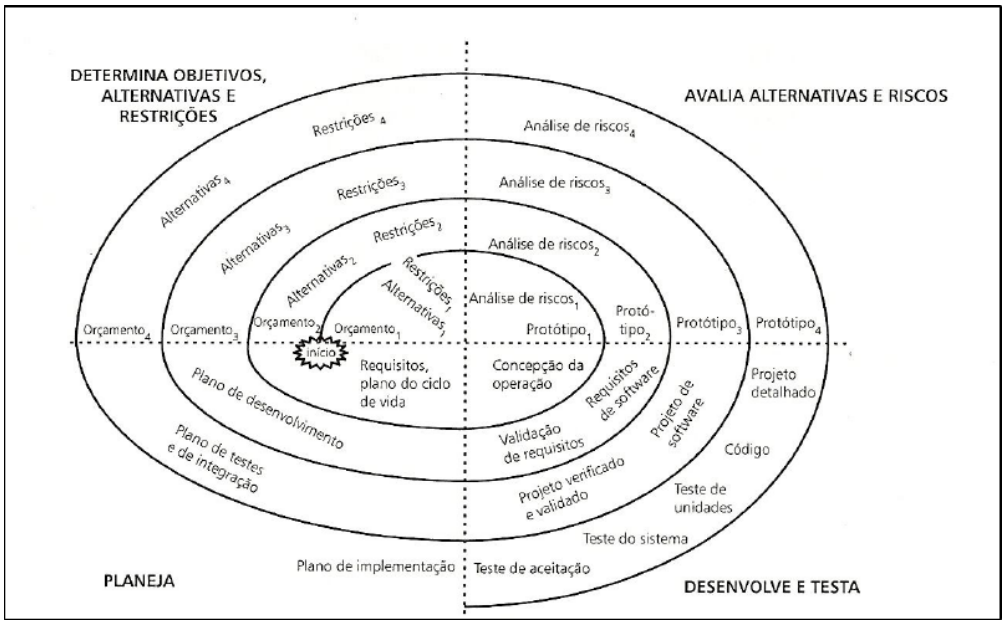
- **Modelo em espiral**: segundo Pressman (2007, p. 65), “fornece potencial para o rápido desenvolvimento de versões cada vez mais completas do software”. O software evolui à medida que o processo avança (cada volta na espiral). Utiliza prototipação para redução de riscos e a aplica em qualquer estágio do processo. Inicia em um espiral, começando no centro e seguindo no sentido horário, passando por um conjunto de atividades metodológicas definidas pela equipe de engenheiros de software. Os riscos, definidos no início do processo, são revisados e analisados a cada evolução da espiral. A evolução do modelo pode ser observada mais detalhadamente na Figura 3.
- **Modelo evolucionário**: baseia-se em uma implementação de um protótipo inicial que é exposto para comentários do cliente/usuário, podendo sofrer diversos ajustes até que se chegue a um sistema adequado. Uma vantagem é que a especificação ocorre de forma incremental. Também é recomendado quando nem cliente nem equipe de desenvolvimento conhecem bem os requisitos do sistema, pois a implementação prévia facilita a comunicação e diminui as dúvidas. Na Figura 4, um exemplo possível de fluxo do processo evolucionário.

Figura 2 - Representação do modelo cascata



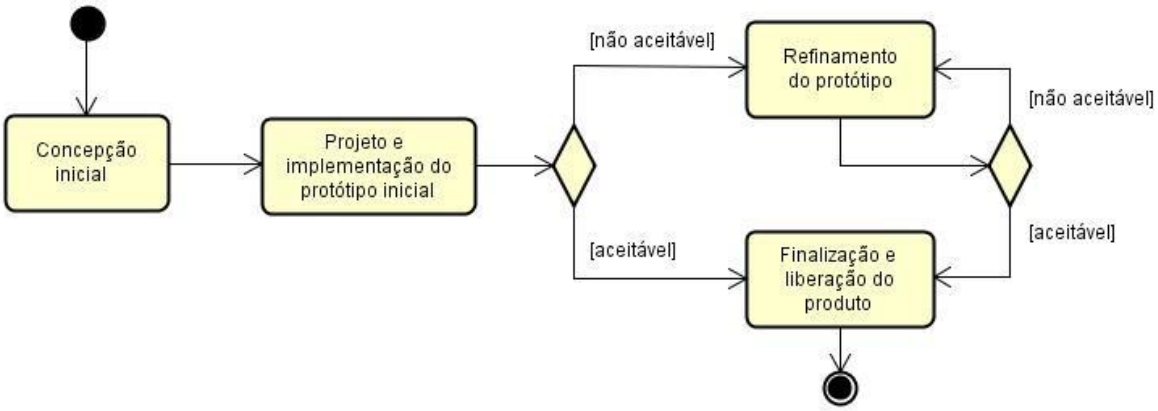
Fonte: SOMMERVILLE, 2007, p. 44

Figura 3 - Representação do modelo espiral



Fonte: PFLEEGER, 2004, p. 47

Figura 4 - Representação do modelo evolucionário



Fonte: Adaptado pela autora com base em Wazlawick (2013, p. 38)

2.1.2 Modelos ágeis

Na atualidade, quando se fala em processo de software, logo se pensa em modelos ágeis. Agilidade no contexto de software pode ser relacionado a intervenções, à capacidade de corresponder rapidamente às mudanças de planos, de cenários e de regras.

Os métodos ágeis foram criados devido aos modelos prescritivos (tradicionais) serem mais demorados, onerando tempo com uma documentação mais completa em todas as suas fases.

De acordo com o Manifesto Ágil, que ocorreu em 2001 reunindo dezessete renomados profissionais da área para criação dos métodos ágeis para desenvolvimento de software, são quatro valores que o fundamentam (AGILE MANIFESTO, 2013):

- Os indivíduos e suas interações acima de procedimentos e ferramentas;
- O funcionamento do software acima de documentação abrangente;
- A colaboração com o cliente acima da negociação e contrato;
- A capacidade de resposta a mudanças acima de um plano pré-estabelecido;

Pressman e Maxim (2016), expõe que Desenvolvimento Ágil poderia ser melhor denominado como engenharia de software flexível, pois as atividades metodológicas básicas permanecem. No entanto, tornam-se apenas um conjunto de atividades mínimas que disparam a equipe para o desenvolvimento e entrega. Uma metodologia de desenvolvimento ágil prevê que se possa eliminar tudo, exceto os artefatos essenciais, conservando os enxutos, e dê ênfase a entregas incrementais, liberando para o cliente um produto operacional o mais rapidamente possível.

Como exemplos de métodos ágeis de desenvolvimento de software, pode-se citar Extreme Programming (XP), Scrum, Feature Driven Development (FDD), Lean Software Development (LSD), Agile Unified Process (AUP) e Dynamic System Development Method (DSDM).

A seguir, serão detalhados dois processos ágeis de desenvolvimento de software relacionados com essa pesquisa.

2.1.2.1 Extreme Programming

O processo ágil XP (programação extrema) entende que existe uma volatilidade de requisitos e, ao invés de tentar eliminá-los, trata-os com naturalidade e adapta o processo de desenvolvimento de software para uma abordagem flexível e

colaborativa, onde cliente e equipe do projeto trabalham do mesmo lado procurando gerar um produto com alto valor agregado, conforme Prikladnicki, Willi e Milani (2014). O XP combina uma abordagem colaborativa com boas práticas de engenharia de software que contribuem para o aumento da qualidade e garantem que o produto final atenda às necessidades do negócio. Revisão do código, testes automatizados, *design* simples, integração e *feedback* do cliente são alguns exemplos dessas boas práticas.

O extremismo desta metodologia, vem de utilizar as boas práticas ao extremo, como: a revisão de código é feita constantemente através da programação em pares; testes, que ajudam a aumentar a qualidade do produto, em XP é feito Desenvolvimento Orientado a Testes (TDD); *feedback* do cliente é feito com acompanhamento constante para validar a funcionalidade. Desta forma, Prikladnicki, Willi e Milani (2014) explanam que a força de XP está no conjunto das boas práticas, que se apoiam e criam sinergia.

A proposta de XP se fundamenta em cinco valores:

- Comunicação: transparência entre equipe, cliente e usuários, com estimativas realistas e exposição de dificuldades sendo parte para comunicação eficaz. Também é importante para disseminar o conhecimento e identificar soluções.
- Simplicidade: modelagem, arquitetura e codificação mais simples do que complexas. Isto não significa baixa qualidade; encontrar uma solução simples é mais difícil do que encontrar uma solução. A sofisticação precoce atrasa a entrega de funcionalidades básicas e dificulta o crescimento e manutenção da arquitetura.
- Coragem: o sucesso de um projeto de desenvolvimento de software depende das pessoas envolvidas, por isso, XP adota uma postura de equipes encorajadas e comprometidas com os objetivos.
- *Feedback*: XP prima por *feedback* constante, adotando ciclos curtos de desenvolvimento e colocando à prova suas implementações rapidamente. Qualquer defeito a ser corrigido é mais fácil e rápido de fazê-lo do que se identificado seis meses após implementado.

- Respeito: como os indivíduos são o elemento mais importante para o sucesso de qualquer projeto, é necessário respeito para que a comunicação e *feedback* sejam eficazes. Criar oportunidades de aprendizado e colaboração, mais do que encontrar culpados, valoriza os indivíduos e fortalece a equipe.

Documentação enxuta é uma característica de XP, pois muitos documentos tornam-se onerosos à medida que mudanças são necessárias e ficam sujeitos à falta de informação por falhas na elaboração ou atualização dos mesmos. Por isso, concentra seus esforços em criar software com código de alta qualidade e fácil compreensão, para que não sejam necessários outros materiais para seu entendimento, presumem Prikladnicki, Willi e Milani (2014).

A Figura 5 representa o processo Extreme Programming e suas atividades-chave (PRESSMAN; MAXIM, 2016).

Figura 5 - O processo Extreme Programming

Fonte: PRESSMAN; MAXIM, 2016, p. 72.

Conforme Pressmann e Maxim (2016), as atividades chave do processo Extremme Programming são:

- Planejamento: atividade de ouvir e criar um conjunto de histórias de usuários que descrevem características e funcionalidades esperadas. Cada história recebe um valor de negócio atribuído pelo cliente e um custo (semanas) atribuído pela equipe XP.
- Projeto: segue o princípio KISS (*keep it simple, stupid!*), ou seja, não complicar. É desestimulado qualquer suposição de funcionalidade extra que possa vir a ser necessária. Nesta fase são criados os cartões CRC (classe-responsabilidade-colaborador), único artefato de projeto produzido. Neles são identificadas as classes orientadas a objetos relevantes para o incremento do software.
- Codificação: inicia com o desenvolvimento de uma série de testes que exercitam as histórias de usuário. Tendo isto resolvido, é focado no que precisa ser desenvolvido para passar no teste. Nada estranho pode ser adicionado (KISS), estando completo o código já pode passar para os testes de unidade e receber *feedback* imediato.
- Testes: a premissa é que os testes de unidade sejam criados visando sua automatização, para serem facilmente repetidos sempre que o código sofrer alterações. Os analistas de testes exercem um papel proativo, ajudando clientes e desenvolvedores a escrever testes antes de serem implementados. E, ao longo das interações, podem ajudar os desenvolvedores na automatização dos testes. Quando não é possível automatizar, eles os executam manualmente.

2.1.2.2 Scrum

Scrum é um método ágil similar ao XP, porém com maior foco em gerência. É fundamentado na teoria empírica de controle de processos e, segundo Salazar (2016), é suportado pelo tripé:

- Transparência: garante que todos os elementos do processo sejam visíveis e compreendidos por todos os envolvidos.
- Inspeção: defende que todos os elementos do processo devem ser monitorados com frequência para garantir que mudanças inesperadas sejam identificadas e aplicadas as devidas medidas.
- Adaptação: constitui-se de realizar ajustes no processo quando ocorrem variações imprevistas. Os ajustes devem ser rápidos para evitar impactos negativos na qualidade do produto.

A Figura 6 representa o processo de acordo com o método Scrum.

Figura 6 - Fluxo de trabalho conforme Scrum

Fonte: Adaptado pelo autor com base em Rubin (2013, p.17)

O desenvolvimento através do método Scrum ocorre de forma iterativa e incremental, com ciclos de desenvolvimento completos e duração fixa, que resultam em incrementos de software potencialmente entregáveis (PRIKLADNICKI; WILLI; MILANI, 2014). Essas iterações são chamadas de *Sprints*, sua duração pode ser de duas ou quatro semanas, o que permite *feedback* constante do Dono do Produto.

O Scrum possui três tipos de papéis:

- *Product Owner* ou Dono do Produto: responsável pelo valor de negócio do produto;

- *Scrum Master*: organiza o *sprint* de um time e recebe as inferências externas;
- *Team* ou Equipe de Desenvolvimento: multidisciplinar, formada por indivíduos que realizam diversas atividades de construção do software, responsável por entregar valor em um incremento de software.

Além disso, também fazem parte do Scrum os artefatos *Backlog do Produto* e *Backlog do Sprint*. O primeiro engloba o conjunto de requisitos e funcionalidades desejadas para o produto; o segundo é um conjunto de requisitos retirado a partir do *Backlog do Produto* e que forma o escopo de trabalho para um *Sprint*.

Quatro cerimônias também compõe os elementos do Scrum:

- Reunião de Planejamento do Sprint, onde a equipe se compromete com um conjunto de requisitos e estima o esforço para resolução;
- Reuniões Diárias, que têm o intuito de promover maior organização e comunicação entre a equipe, é onde os membros relembram o que foi feito no dia anterior e traçam a meta do que será feito no dia corrente, geralmente ocorre no início do dia de trabalho;
- Revisão de Sprint, momento em que a Equipe se reúne com o Dono do Produto para inspecionar o resultado;
- e a Retrospectiva de Sprint, que ocorre apenas entre os integrantes do time, oportunidade onde ocorre revisão e adaptação dos processos empíricos.

Quanto aos testes, de forma semelhante ao XP, no Scrum o testador também planeja e executa testes. Ele já inicia um Sprint planejando atividades de elaboração de cenários de teste, de acordo com os requisitos que foram escolhidos para constituírem o *Backlog* do *Sprint*. Também, visto que os ciclos de Scrum são relativamente curtos e a cada *Sprint* resulta um produto ou uma parte entregável, é necessário agilidade para garantir a qualidade de tudo que já estava implementado e tudo que foi agregado. Para isso, uma boa solução, é executar testes de regressão automatizados. Portanto, passa a ser mais uma tarefa para o testador da equipe, criar e atualizar a automatização de testes quando possível e viável (CRUZ, 2015).

2.2 Qualidade de Software

Embora pareça simples e intuitiva, qualidade de software quando estudada a fundo é complexa. A qualidade tem importância fundamental para concorrência entre as empresas da área, deixou de ser um diferencial e já é tratada como um pré-requisito. Além disso, um software sem qualidade pode causar desde pequenos danos até problemas catastróficos.

A qualidade de software está relacionada aos requisitos solicitados pelo cliente e, também, às regras de desenvolvimento. No entanto, existem diversas definições para Qualidade.

Koscianski e Soares (2006) enfatizam que a noção de qualidade pode ser relativa, mas tem um objetivo: satisfazer o cliente. Para isso, é preciso considerar diversos fatores, desde os que atingem a construção do programa até os que influenciam no julgamento dos usuários.

Mecenas e Oliveira (2005) simplificam que um sistema de qualidade “é a seleção de um conjunto de características e de como elas se relacionam, de tal forma que forneçam os elementos básicos para a especificação dos requisitos e sua correta implementação”.

A busca pela qualidade de software se motiva por alguns benefícios, como: facilidade de uso do sistema, funcionamento correto, manutenibilidade e, principalmente, integridade dos dados. Por outro lado, a qualidade no processo de software também é importante pois pode reduzir custos. Pressman (2011) diz que há duas opções na criação de um sistema: pode-se fazer certo na primeira vez ou pode-se fazer tudo de novo. Um software produzido com baixa qualidade, ao longo do tempo demandará retrabalhos, e retrabalho é gasto.

Desta forma, percebe-se que a qualidade de software se dá tanto no produto quanto no processo de desenvolvimento. No processo, pode-se quantificar sua qualidade através de métodos de garantia da qualidade no formato de auditorias e

reportes para a alta gerência, além de avaliações constantes do processo e análise estatística de controle do processo. No produto, os métodos de garantia da qualidade são revisões, inspeção formal e teste de software, além de revisão dos resultados do teste de software realizada por especialistas, testes realizados pelo cliente e avaliações ISO.

Para os dois casos existem modelos de boas práticas e normas que regulamentam como se dá a qualidade em cada um. A seguir, será explanada uma norma que rege a qualidade do produto e um modelo que guia a qualidade de processo.

2.2.1 NBR ISO/IEC 25010

A NBR ISO/IEC 25010 Engenharia de Software - Qualidade de Produto de Software estabelece um modelo de qualidade interna e externa do produto, categorizando atributos de qualidade de software em seis características e, estas, subdivididas em subcaracterísticas. O objetivo é que o produto tenha o efeito requerido num contexto de uso específico. O Quadro 1 contém detalhadamente todos os atributos avaliados por esta norma.

Quadro 1 - Características e Subcaracterísticas da NBR ISO/IEC 25010

Características	**Subcaracterísticas**
Funcionalidade	Adequação; Acurácia; Completude funcional.
Eficiência	Comportamento em relação ao tempo; Utilização de recursos; Capacidade.
Compatibilidade	Coexistência; Interoperabilidade.
Usabilidade	Inteligibilidade; Apreensibilidade; Operacionabilidade; Atratividade.
Confiabilidade	Maturidade; Disponibilidade; Tolerância a falhas; Recuperabilidade.

Segurança	Confidencialidade; Integridade; Não repúdio; Autenticidade; Prestação de contas.
Manutenibilidade	Analisabilidade; Modificabilidade; Estabilidade; Testabilidade; Reusabilidade.
Portabilidade	Adaptabilidade; Capacidade para ser instalado; Capacidade para substituir.

Fonte: Da autora, adaptado de NBR ISO/IEC 25010 (2011)

2.2.2 CMMI - *Capability Maturity Model Integration*

É um metamodelo de processo desenvolvido pelo SEI (Software Engineering Institute - Carnegie Mellon University - EUA) na década de 1990. É um modelo abrangente e qualificado em capacidades de sistema e engenharia de software que devem existir ao passo que as empresas atingem diferentes níveis de capacidade e maturidade de processo. Possui dois tipos de representações: contínuo e por estágios. O caminho pela representação contínua permite melhorar de forma incremental os processos correspondentes a uma ou mais áreas de processo escolhidas pela empresa; para esta representação, utiliza-se a expressão Nível de Capacidade.

Já o caminho pela representação por estágios possibilita melhorar um conjunto de processos inter-relacionados e tratar sucessivos conjuntos de áreas de processo de forma incremental; emprega-se a expressão Nível de Maturidade. O Quadro 2 apresenta uma comparação de estágios entre os níveis de Capacidade e Maturidade.

Quadro 2 - Comparação entre Níveis de Capacidade e Maturidade

Nível	**Níveis de Capacidade**	**Níveis de Maturidade**
Nível 0	Incompleto	Não se aplica
Nível 1	Executado	Inicial
Nível 2	Gerenciado	Gerenciado
Nível 3	Definido	Definido
Nível 4	Gerenciado Quantitativamente	Gerenciado Quantitativamente
Nível 5	Em Otimização	Em Otimização

Fonte: Da autora, adaptado de CMMI-DEV (2006)

2.2.3 Métricas

Medidas podem ser usadas para avaliar a qualidade dos sistemas construídos. Porém, no mundo do software, medidas são muitas vezes indiretas, diferentemente de outras áreas que possuem medidas diretas como velocidade, temperatura, tensão, entre outras. No entanto, podem proporcionar uma maneira sistemática de avaliar a qualidade baseando-se em um conjunto de regras previamente definidas (PRESSMAN, 2011).

Para Koscianski e Soares (2007), a utilização de medidas também ajuda a resolver o problema da influência de fatores subjetivos no julgamento da qualidade dos sistemas. Um critério quantitativo ajuda a equipe a trabalhar sob menos pressão de opiniões.

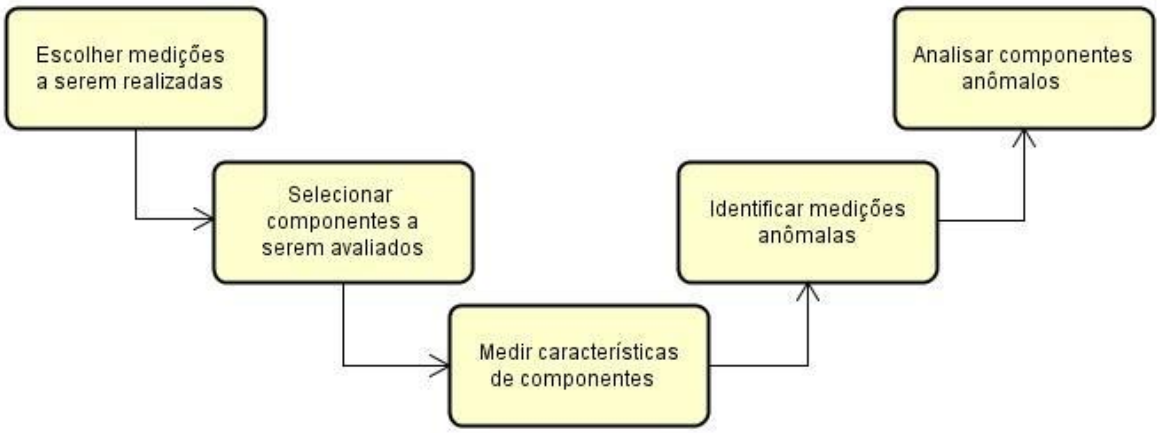
Pressman (2011) explica a diferença para os seguintes termos:

- Medida: indica a extensão, quantidade, capacidade ou tamanho de algum atributo de um produto de processo.
- Medição: é o ato de determinar uma medida.

- Métrica: medida quantitativa do grau que um sistema ou processo possui em determinado atributo.

Na Figura 7, Sommerville (2007) ilustra o processo de medição de software. Ele considera que componentes do sistema devem ser analisados separadamente e os valores das métricas comparados entre eles. Os dados coletados devem ser mantidos para posteriormente possuir uma base histórica para comparação entre projetos. Uma vez que existir essa base histórica, é possível refinar as métricas para juntar especificamente as informações relevantes que auxiliam a empresa no aprimoramento da qualidade do seu produto.

Figura 7 - Processo de medição do produto



Fonte: Adaptado pela autora com base em Sommerville (2007, p. 434)

Ainda, Sommerville (2007) declara que, infelizmente, as características de software que podem ser facilmente medidas não têm relação direta com a qualidade do produto.

Pressman (2011) sintetiza diversos tipos de métricas:

- Métricas para modelo de requisitos: focam em função, dados e comportamento. *Function point* ou pontos de função, são um exemplo de métrica utilizada, que estabelece valores de domínio, como: Número de entradas externas, Número de saídas externas, Número de consultas externas, Número de arquivos lógicos internos e Número de arquivos de interface externos.

- Métricas para projeto: consideram projeto de interface, projeto em nível de componente e aspectos de arquitetura.
- Métricas para WebApps: também consideram aspectos de interface de usuário, conteúdo, navegação e estética da aplicação.
- Métricas para código-fonte: utiliza operadores e operandos presentes no código com medidas primitivas que são utilizadas em expressões para medir atributos como o tamanho global do sistema, volume mínimo potencial para um algoritmo, nível de linguagem, esforço de desenvolvimento.
- Métricas para teste: as métricas para teste são deficientes e normalmente testadores se baseiam em métricas de análise, projeto e código-fonte para execução dos casos de teste. Binder apud Pressman (2011) sugere que algumas métricas de projeto têm impacto direto na testabilidade de um software orientado a objeto. Consideram aspectos de encapsulamento e herança: Falta de coesão em métodos, Porcentagem pública e protegida, Acesso público a membros de dados, Número de classes-raiz, *Fan-in* (medida do número de funções que chamam alguma outra função ou método) e Número de filhos e altura da árvore de heranças.
- Métricas para manutenção: indica a estabilidade de um programa. Exemplo o SMI (*software Maturity Index*), ou índice de maturidade do software, utilizado para planejar atividades de manutenção de software. Está relacionado com o tempo médio necessário para produzir uma versão.

2.2.4 Testes

Rios e Moreira (2013) definem que teste de software “é o processo que visa a sua execução de forma controlada, com o objetivo de avaliar o seu comportamento baseado no que foi especificado”. Ou seja, é verificar se o software está fazendo o que deveria fazer, e se não está fazendo o que não deveria fazer.

Segundo Martin&McClure apud Rios e Moreira (2013), 67% dos custos totais de software são contabilizados para manutenção. Portanto, quanto mais tarde forem identificados os defeitos, maior o custo para corrigi-los.

Importante diferenciar o conceito de alguns termos que muitas vezes podem ser confundidos (WAZLAWICK, 2013):

- um **defeito** está relacionado ao código, bloco ou conjunto de dados; é uma não conformidade no produto ou no processo; é algo que está implementado de maneira errada e pode provar um erro.
- uma **falha** é um não funcionamento do sistema que pode ser provocado por um defeito ou até mesmo uma questão ligada ao hardware.
- um **erro** é a diferença encontrada entre um resultado obtido e um resultado esperado; é um resultado que, por causa de um defeito ou falha, é diferente do esperado.
- um **engano** é uma ação humana que produziu um defeito no software.

Conforme Wazlawick (2013), durante muitos anos a atividade de teste de software era de responsabilidade dos programadores. Entretanto, isto era uma tarefa ingrata, visto que destacava a incapacidade de produzir software sem defeitos.

Atualmente, muitas empresas despendem investimentos para um grupo independente de testes, o ITG (*independent test group*). Isto evita um possível conflito de interesses que pode haver entre o grupo de programadores que criou o software. Ainda assim, os desenvolvedores não deixam de participar dos testes, são responsáveis pelos testes de unidades individuais (componentes) do programa, garantindo que cada um apresente o comportamento para o qual foi projetado (PRESSMAN, 2011).

Segundo Sommerville (2008), existem duas grandes fases de teste de sistemas: **Testes de integração**, identificam problemas de interação entre os componentes, geralmente acessando o código-fonte do sistema, a equipe procura por defeitos e identifica os componentes que devem ser depurados; e, **Testes de *release***, quando é testada uma versão do sistema que já é candidata à entrega para os clientes, tem por objetivo validar se o sistema atende os requisitos e assegurar que é confiável.

Além disso, há duas categorias de atividades de teste, que são a **Verificação** e a **Validação**. Conforme Koscianski e Soares (2007), a verificação consiste em um processo de averiguar se o sistema está de acordo com os requisitos especificados,

enquanto que a validação é o processo de confirmar que as especificações preestabelecidas realmente atendem às necessidades dos *stakeholders*.

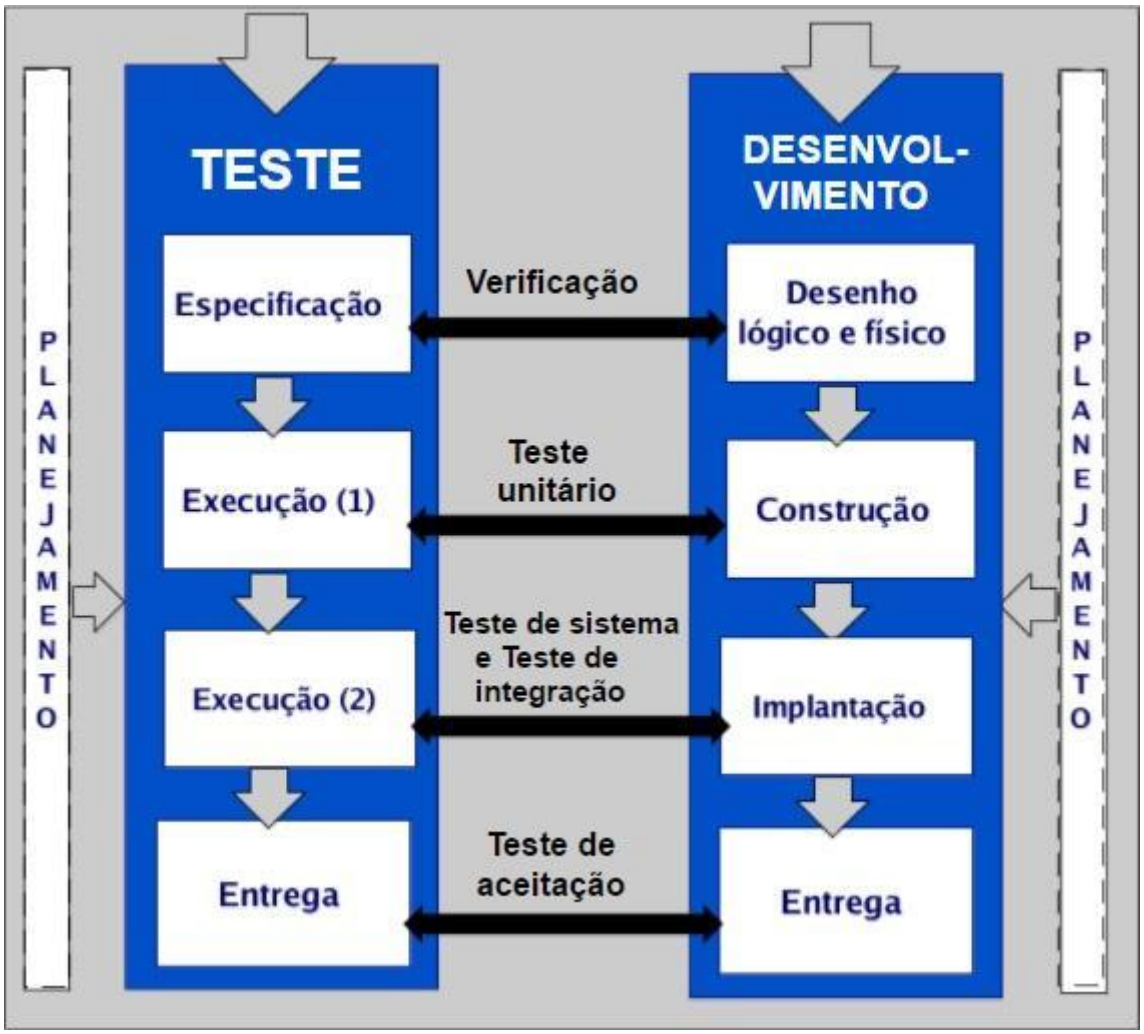
A seguir são descritos alguns dos muitos tipos de teste de software existentes, definições de acordo com Rios e Moreira (2013):

- Testes Caixa Preta: objetiva verificar a aderência das funcionalidades em uma perspectiva externa, sem conhecer a lógica interna e código do sistema.
- Testes Caixa Branca: avaliam o código, a lógica entre os componentes, configurações e outros elementos técnicos.
- Testes unitários: pertencem ao nível mais baixo da escala de testes, verificam o funcionamento de um pedaço de sistema e geralmente são aplicados pelos próprios desenvolvedores.
- Testes de integração: são executados em uma combinação de componentes (pedaços de código, módulos, aplicações distintas, etc) e visam assegurar que as partes funcionem juntas corretamente e que os dados são processados de acordo com as especificações.
- Testes de regressão: é a reaplicação de uma sequência de testes que visa garantir que o software permaneça íntegro após a introdução de mudanças que podem danificar códigos já considerados bons e aceitos. Para isso deve ser mantida uma *baseline* de dados e *scripts* para que possam ser comparados os resultados após as mudanças.
- Testes de carga: visa avaliar a resposta do sistema para uma carga pesada de dados, grandes valores numéricos, repetições de entrada de dados, consultas complexas e, por outro lado, acesso simultâneo com grande número de usuários para verificar o nível de escalabilidade. Também são chamados de testes de estresse.
- Testes de configuração: verificam se o software funciona em diferentes versões e configurações de ambientes (software e hardware), como em navegadores diferentes e versões diferentes de navegadores, por exemplo.
- Testes de usabilidade: verificam o grau com o que o sistema facilita a vida do usuário, são observados alguns pontos como interatividade, *layout*, clareza, personalização, acessibilidade, telas de ajuda.
- Testes de instalação: verificam o processo de instalação e atualização do software.

- Testes de segurança: avaliam integridade, autenticação, permissões, confidencialidade, disponibilidade e não repúdio (garantir que um emissor ou receptor não possam alegar que não tenham enviado ou recebido uma mensagem, por exemplo).
- Testes de recuperação: verificam a capacidade do software de se recuperar após falhas e voltar a seu estado operacional.
- Testes de compatibilidade: avaliam a capacidade do software para rodar em ambientes específicos (sistema operacional, redes, etc).
- Testes de desempenho: também chamados de testes de performance, verificam o tempo de resposta.
- Testes de qualidade de código: verificam o código-fonte em consonância com padrões, boas práticas, instruções não executadas e outros.
- Testes de recuperação de versão: verificam a capacidade de voltar a uma versão anterior do sistema.
- Testes funcionais: avaliam se as funcionalidades especificadas estão implementadas e funcionando de acordo com as regras de negócio definidas.
- Testes de aceitação: testes finais executados pelos usuários antes da utilização em produção, objetivam validar se o sistema atende aos objetivos do negócio e seus requisitos, no que tange funcionalidade e usabilidade.

Sendo que os testes de software têm ligação direta com a qualidade do produto, a literatura sugere que seja realizado um planejamento de testes, que inicia antes mesmo da construção do software (RIOS; MOREIRA, 2013). De modo geral, um plano de testes engloba as funcionalidades a serem testadas, casos de teste, saídas esperadas, critérios de aceitação. A Figura 8 apresenta uma representação de como os processos de teste e desenvolvimento se relacionam.

Figura 8 - Modelo de integração entre os processos de desenvolvimento e teste



Fonte: Adaptado pela autora com base em Rios e Moreira (2013, p. 14)

Em consonância com a seção anterior, o Quadro 3 elenca alguns exemplos de medições realizadas durante a fase de testes de software, conforme visto em Wazlawick (2013)

Quadro 3 - Métricas para testes de software

Métricas durante a preparação e execução dos testes	**Métricas relativas ao produto**
Número de testes previstos Números de testes planejados Número de testes executados a) teste que falharam b) testes que foram aprovados	Número de defeitos descobertos Número de defeitos corrigidos Distribuição dos defeitos por grau de gravidade Distribuição dos defeitos por módulo

Fonte: Da autora, adaptado de WAZLAWICK (2013).

2.3 *Releases* de Software

Release é um termo utilizado na Engenharia de Software pela área de Gerência de Configuração que define *release* como uma versão do próprio software; e na prática de IC (integração contínua) que possibilita a criação de vários *branches*. Wazlawick (2013) define *release* como sendo a distribuição da versão de um software para fora do ambiente de desenvolvimento. Conforme Andrade Junior (2015), há dois tipos de *releases*: as principais, são versões mais significativas do software, geralmente oferecem novas funcionalidades; e as menores, comumente corrigem *bugs* e reparam defeitos relatados.

Bartié (2002), resolve que uma *release* deve ser produzida e liberada com regularidade para forçar resultados concretos. Ou seja, para que seja liberada uma versão entregável, evitando que o sistema fique em um processo de desenvolvimento infinito, recebendo correções e novas ideias constantemente sem ter um fim.

Conforme Pressmann (2008), as principais empresas de software criam programas com erros conhecidos e mesmo assim entregam uma *release* para os usuários. Reconhecem que o tempo para colocação do produto no mercado é mais importante que um sistema perfeito. Para isso, existe o apoio da área de Gerência da Configuração, que versiona e cria *baselines* dos produtos de trabalho. Isto garante que novos esforços possam ser empenhados no programa, inserindo novas funcionalidades, testando e corrigindo sempre que necessário sem prejudicar a utilização do usuário final.

Também, com a liberação de *releases*, é favorecido o *feedback* do cliente, conforme visto anteriormente como características dos métodos ágeis. O cliente pode utilizar um software acabado, porém não completo, e já fornecer sugestões para a próxima versão. Isto ajuda a quebrar as barreiras entre os desenvolvedores e os clientes, e os torna mais colaborativos (FOWLER, 2006).

As *releases* fazem parte do processo de entrega de software, bem como, da prática de integração contínua. Fowler (2006) explica integração contínua como sendo uma prática do desenvolvimento de software onde um grupo de programadores trabalha simultaneamente no mesmo código, criando primeiramente cópias locais (*branches*), e depois fazendo integração com o código principal (*trunk*). Após a integração, se dá a compilação do código e, então, passa para a fase testes.

Para que se dê a integração contínua, é necessário fundamentalmente que exista um controle de versão do código-fonte. Alguns exemplos de ferramentas *free-source* são: CVS (*Concurrent Version System*), Mercurial, Git e SVN. Comparação de código, possibilidade de restaurar versões anteriores e gestão de mudanças, são alguns benefícios quando se realiza o gerenciamento das versões do código-fonte.

Além disso, para que ocorra uma boa integração, é necessário a automatização dos *builds.* O processo de *build* consiste em compilar e preparar o executável do programa, e devem compor o *build* todo o material necessário para ter um programa executando (scripts de teste, arquivos de configuração, esquema de banco de dados, scripts de instalação, bibliotecas de terceiros). Sua automatização é importante para evitar problemas e algumas ferramentas podem ser utilizadas como apoio: ant, Maven, Nant, MSBuild, entre outras. Segundo Fowler (2006), também é importante que os builds sejam auto testáveis, para melhorar e agilizar a detecção de erros.

O maior benefício da integração contínua pode ser considerado a redução de riscos, pois a IC promove a comunicação entre a equipe sobre as alterações que são realizadas (FOWLER, 2006), permitindo resolver rapidamente qualquer falha na construção de uma RC (*release candidate*). Isto contribui para entregas mais frequentes, que são valorosas pois permitem aos usuários terem acesso às novas funcionalidades e dar *feedback* rapidamente conforme já mencionado anteriormente.

A integração contínua e a liberação de *releases* são etapas importantes e estão interligadas no processo de desenvolvimento de software com mais qualidade, auxiliando no crescimento e entregas dos sistemas. Consequentemente, haverá mais feedbacks dos usuários, o que contribui na melhoria e qualidade dos produtos.

2.4 *Checklists*

Um *checklist* é, na prática, uma lista de verificação composta por um conjunto de itens e condutas a serem seguidas. No contexto de software, os *checklists* podem ser considerados um roteiro de trabalho, que auxiliam na padronização e uniformização do que será verificado (BARTIÉ, 2002). Geralmente são utilizados como uma ferramenta no processo de Teste de Software.

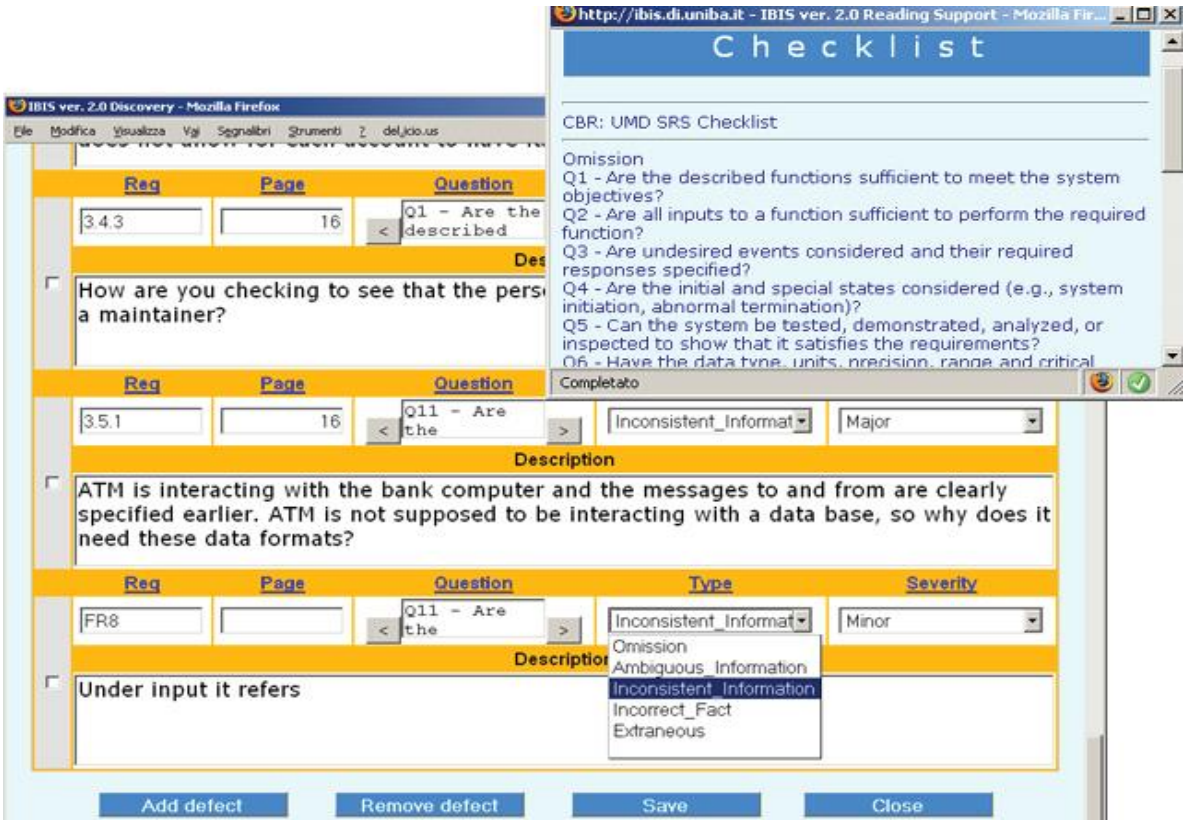
O *checklist* tem por objetivo evitar o esquecimento de itens a serem testados, bem como, garantir abrangência independentemente da pessoa que estiver testando. Nele, deverão constar os caminhos a serem percorridos, bem como o que deverá ser verificado. Quanto maior a cobertura e identificação de problemas na fase de testes, menor os custos com manutenção, menor insatisfação de clientes e menor o dano para a imagem da empresa.

Mello et al. (2012) elaboraram um estudo onde criam uma técnica de inspeção de software baseada em *checklist*, para apoiar a detecção de defeitos semânticos em *feature models*. Eles definem *feature models* como sendo modelos de características que são gerados em modelagens com abordagens de desenvolvimento para reuso. No estudo, foi elaborada a FMCheck (*Feature Model Checklist*), que tem por objetivo identificar discrepâncias e prevê um fluxo de atividades que antecedem a aplicação do *checklist*. As atividades prévias correspondem à resposta de um questionário pelo analista/projetista, configuração do *checklist* por um moderador da inspeção e, por fim, a inspeção individual aplicando o FMCheck. Eles consideram que a verificação dos artefatos de software desde as fases iniciais do desenvolvimento ajudam a promover tanto a qualidade do produto quanto a produtividade no desenvolvimento, antecipando defeitos e, desta forma, reduzindo o retrabalho.

Kalinowski e Spínola (2007) também trazem a inspeção de software como uma abordagem eficiente para encontrar defeitos, através da revisão dos artefatos produzidos ao longo do processo de desenvolvimento de software. Segundo eles, “o esforço gasto por organizações de software com retrabalho pode variar em média

entre 40% e 50% do esforço total do desenvolvimento de um projeto". Entre outras ferramentas, citam IBIS (*Internet Based Inspection System*) como ferramenta de apoio ao processo de inspeção, que permite uma inspeção de forma sistematizada não limitando a nenhum tipo de artefato, e a partir dela são gerados *checklists*. A Figura 9 mostra o visual da ferramenta IBIS.

Figura 9 - IBIS para registro de defeitos em inspeção de software



Fonte: KALINOWSKI; SPÍNOLA, 2007, p. 73

Bartié (2002) cita *checklist* como um método estruturado para verificação e que pode ser utilizado nas diversas fases do processo, como em Negócios, Requisitos, Modelagem e Implementação. As Figuras 10, 11 e 12 apresentam alguns exemplos de *checklists* específicos para fases diferentes.

Figura 10 - Exemplo de *checklist* para verificação de requisitos

Checklist de Requisitos		
Diagrama de Casos de Uso		
– Existe um modelo de casos de uso para cada subsistema identificado.	☐ Sim	☐ Não
– Todos os casos de usos estão adequadamente descritos.	☐ Sim	☐ Não
– Todos os atores estão adequadamente representados.	☐ Sim	☐ Não
Levantamento de Requisitos		
– Cada caso de uso representa um requisito funcional.	☐ Sim	☐ Não
– Existe rastreabilidade entre requisitos identificados e necessidades.	☐ Sim	☐ Não
– Requisitos foram avaliados por importância, volatilidade e criticidade.	☐ Sim	☐ Não
Especificações Funcionais		
– Cada requisito funcional possui uma especificação detalhada.	☐ Sim	☐ Não
– As especificações contemplam os fluxos básicos, alternativos e exceção.	☐ Sim	☐ Não
– As especificações contemplam pré-requisitos e pós-condições.	☐ Sim	☐ Não
Especificações Não Funcionais		
– Todas as categorias de requisitos não funcionais foram levantadas.	☐ Sim	☐ Não
– Cada requisito não funcional possui uma especificação detalhada.	☐ Sim	☐ Não
– Todas as dependências dos componentes foram estabelecidas.	☐ Sim	☐ Não

Fonte: BARTIÉ, 2002, p. 90

Figura 11 - Exemplo de *checklist* para verificação de código-fonte

Checklist do Código-fonte em Visual Basic		
Comparação do Modelo de Arquitetura do Software com o Código-fonte		
– Todas as classes do modelo foram implementadas.	☐ Sim	☐ Não
– Todos os métodos de cada classe foram implementados.	☐ Sim	☐ Não
– Todos os atributos de cada classe foram implementados.	☐ Sim	☐ Não
Mensagens Apresentadas ao Usuário Final		
– Nenhuma mensagem apresenta erros gramaticais.	☐ Sim	☐ Não
– Todas as mensagens são claras e bem objetivas.	☐ Sim	☐ Não
– Todas as mensagens apresentam ícones adequados ao contexto.	☐ Sim	☐ Não
Legibilidade do Código		
– Todas as estruturas estão adequadamente identadas.	☐ Sim	☐ Não
– Não existem linhas agrupadas com IF, SELECT, FOR NEXT e FOR EACH.	☐ Sim	☐ Não
– Tratamentos de erros e desvios sempre estão no final das rotinas.	☐ Sim	☐ Não
– Todas as declarações de variáveis e constantes estão no início da rotina.	☐ Sim	☐ Não
– Não existem vários comandos em uma única linha.	☐ Sim	☐ Não
Volume de Comentários		
– Todas as rotinas possuem descrição sobre seu comportamento.	☐ Sim	☐ Não
– Todos os desvios de rotinas possuem um comentário.	☐ Sim	☐ Não

Fonte: BARTIÉ, 2002, p. 95-96

Figura 12 - Exemplo de *checklist* para verificação de banco de dados

Checklist do Banco de Dados		
Comparação do Modelo de Dados com o Banco de Dados		
– Todas as tabelas do modelo de dados foram implementadas.	☐ Sim	☐ Não
– Todos os campos de cada tabela foram implementados.	☐ Sim	☐ Não
– Todos os índices de cada tabela foram implementados.	☐ Sim	☐ Não
– Todos os *stored procedures* de cada tabela foram implementados.	☐ Sim	☐ Não
– Todas as visões do modelo de dados foram implementadas.	☐ Sim	☐ Não
– Todos os campos de cada visão foram implementados.	☐ Sim	☐ Não

Fonte: BARTIÉ, 2002, p. 96

Estes exemplos de *checklist* contribuem para a criação de uma lista de verificação específica para testes funcionais. Mesmo com diferentes formatos e critérios, os *checklists* seguem o padrão que busca garantir que um conjunto de itens seja verificado a cada revisão do sistema. A forma de avaliar cada item pode variar para simples situações de 'Sim ou Não', como também quantificar o item de maneira mais estratificada, estabelecendo mais níveis, como por exemplo: A - Atende, P - Parcialmente e NA - Não Atende.

3 METODOLOGIA

Este capítulo tem por objetivo explicitar acerca da metodologia de pesquisa utilizada para desenvolvimento deste trabalho de conclusão de curso e informar sobre o estudo de caso que foi realizado.

3.1 Tipo de Pesquisa

De acordo com Gil (2008, p. 27), as pesquisas exploratórias têm como finalidade “desenvolver, esclarecer e modificar conceitos e ideias, tendo em vista a formulação de problemas mais precisos [...] proporcionar visão geral, de tipo aproximativo, acerca de determinado fato”. Ainda, Wainer (2007) se refere à pesquisa exploratória como a busca por aprovar ou desaprovar uma teoria particular para fatos de interesse, e a formulação de novas observações e métricas para estes fatos.

Portanto, de acordo com os objetivos estabelecidos, este trabalho caracteriza-se de natureza exploratória, permitindo que sejam estudados os diversos aspectos que envolvem testes estruturados através de *checklists*, em contrapartida aos testes com maior base de conhecimento proporcionados pela *expertise* do testador ao realizar testes exploratórios.

Os procedimentos técnicos que serão utilizados neste trabalho são a pesquisa bibliográfica e o estudo de caso. A pesquisa bibliográfica permite utilizar vários recursos disponíveis e publicados sobre um determinado tema, e possibilita acesso às informações de maneira mais ampla (KLEIN, et al. 2015). Ainda, Lakatos e Marconi (2010, p. 166) esclarecem que a pesquisa bibliográfica “não é mera repetição do que já foi dito ou escrito sobre certo assunto, mas propicia o exame de um tema sob novo enfoque ou abordagem, chegando a conclusões inovadoras”.

O estudo de caso, conforme Prodanov e Freitas (2013), é entendido como uma categoria de investigação, pois constitui-se de coletar e analisar informações de forma aprofundada sobre um objeto de estudo que pode ser um sujeito, uma unidade, um grupo, e pode ser um tipo de pesquisa tanto qualitativa como quantitativa. Este trabalho tem abordagem quantitativa, no que tange as comparações realizadas sobre os resultados angariados nas diferentes formas de aplicação dos testes. E, também, abordagem qualitativa, através do retorno dos envolvidos que foi obtido por meio de um questionário.

A próxima seção detalha mais informações sobre o estudo de caso que foi realizado.

3.2 Estudo de Caso

O estudo de caso se deu sobre uma empresa que pertence à área de negócio de Tecnologia da Informação, sendo sua atividade fim o desenvolvimento de sistemas de apoio à gestão corporativa. Ela se situa no Vale do Taquari, no Rio Grande do Sul, Brasil. Para que não seja identificada, neste trabalho será chamada de forma fictícia por “Empresa A”.

A Empresa A é de médio porte, constituída por 70 colaboradores, sendo 30 especificamente da fábrica de software (FS). A FS é subdividida em três setores: Análise de Produtos, Desenvolvimento de Produtos e Qualidade do Produto.

A empresa já foi certificada CMMI nível 2 e, por isso, adota algumas boas práticas de gerenciamento de projetos. Possui um processo automatizado para gestão de projetos, bem como, um repositório de documentos onde é possível controlar suas versões e definir um ciclo de aprovação para os mesmos. Alguns documentos padrão que fazem parte dos projetos são: Lista de Requisitos, Especificação Funcional e Técnica de Requisito, Planilha de Estimativas, Proposta Comercial, Aceite de Proposta (do cliente), Instruções de Entrega do Produto, Termo de Encerramento, entre outros.

Da mesma forma, a Empresa A possui um rígido controle de versão do código-fonte, utilizando para isso o apoio da ferramenta SVN. Também, realiza integração contínua, com apoio da ferramenta Hudson, e ant para automatização de *builds*. A linguagem de programação utilizada é Java, e os produtos e projetos da Empresa A se dão para plataforma Web. Seu principal produto é uma suíte composta de doze módulos.

Utiliza, em parte, métodos tradicionais no desenvolvimento de projetos, com ciclos de vida iterativo e incremental para grandes projetos e ciclo de vida cascata quando projetos pequenos. Utiliza, também, métodos ágeis, para organização das atividades de análise, desenvolvimento e teste no dia-a-dia da manutenção dos produtos (módulos da suíte). A FS é composta por 5 times *Scrum*, que realizam *Sprints* com periodicidade de duas semanas.

As demandas são organizadas por *tickets*, que podem ser cadastrados tanto por clientes externos como internos. Os tickets são categorizados em Bugs (erros ou falhas), Homologs (erros ou falhas detectados em versão de homologação), Features (melhorias no produto) e Customs (novas ou melhorias em customizações). Também, são os *tickets* que formam o escopo do produto - o *Product Backlog* e o *Sprint Backlog* dos times.

Os testes de software realizados pela Empresa A são os testes unitários - realizados pelos desenvolvedores no seu próprio código criado ou modificado; e testes exploratórios e guiados para cada *ticket* desenvolvido e em todas as versões necessárias - realizados pelos Analistas de Testes da área de Qualidade do Produto, que é composta por sete colaboradores.

Há três tipos de entregas que a Empresa A realiza: a entrega de customizações, que são projetos e seguem um processo formal automatizado, seu escopo é formado por *tickets*; a entrega de *Hotfix*, que são versões pontuais do produto principal com caráter de urgência para correção de algum erro (também é regido por *ticket*); e as entregas de *Releases* de Manutenção (RM), que são versões maiores e possuem uma periodicidade de três meses para novas implementações, correções de erros, testes e liberação aos clientes.

Os testes realizados pela área de Qualidade de Produto se dão para todos os *tickets* desenvolvidos. Todo desenvolvimento se dá para um ou mais *tickets*. Os Analistas de Testes resolvem sua atividade foco de forma totalmente manual, e guiada pela demanda documentada no *ticket*. Esse teste ocorre tanto antes das liberações de versão de Customizações e *Hotfix*, quanto nos *tickets* das *Releases* de Manutenção. As RMs também recebem um teste geral em cada um de seus módulos, mas de forma abrangente e não guiada, conduzido apenas pela intuição do Analista de Testes responsável pelo módulo.

A Empresa A não emprega nenhum tipo de teste automatizado, tanto na etapa de desenvolvimento, como na etapa de testes.

Ainda, é importante destacar que existe um Analista de Testes com uma boa carga de experiência em cada módulo. Portanto, é possível afirmar que cada módulo possui um testador especialista.

3.3 Organização do trabalho

Este trabalho conta inicialmente com uma revisão bibliográfica, a fim de resgatar os principais conceitos que envolvem o processo de desenvolvimento de software. Com isto, objetivou-se entender e buscar mais alternativas para as fases de verificação e validação da criação de sistemas.

Foi criado um *checklist* para verificação das implementações realizadas em dois módulos do produto da Empresa A. O *checklist* foi elaborado em documento do tipo planilha e, a partir dele, foram obtidos resultados que servem como subsídio para comparação com as demais aplicações do *checklist*. Esses resultados também foram analisados em conjunto com dados históricos, os quais foram levantados através da contagem de *tickets* de *releases* anteriores.

Cada rodada de testes teve dois formatos de aplicação: um testador com pouco conhecimento sobre o módulo realizando o teste com base nos itens do *checklist*; e outro testador, com larga experiência, aplicando um teste exploratório com base apenas no seu conhecimento e instinto investigativo. Além disso, após concluído o estudo de caso, foi avaliada a percepção dos envolvidos quanto à mudança no processo e sua efetividade através de um questionário.

Dessa forma, foi possível avaliar a relevância da aplicação de testes guiados via *checklist* em comparação com os testes exploratórios e intuitivos dos especialistas.

4 EXECUÇÃO DOS TESTES

Este trabalho propôs-se a confrontar a efetividade de duas formas de testes de software manuais. De um lado estão os testes guiados e estruturados via *checklist* e, por outro, o conhecimento e experiência de um testador especialista no produto realizando uma verificação exploratória e não estruturada.

Em paralelo, foram coletados dados históricos do registro de erros (Bugs e Homologs) que foram cadastrados nos períodos imediatamente posteriores à liberação de *Releases* de Manutenção, para também comparar com os registros que ocorreram após a liberação da *release* na qual ocorreu o estudo de caso.

Os testes foram aplicados sobre dois módulos que compõem a suíte da Empresa A. Foram escolhidos com base na frequência de implementações que eles recebem, expressividade na carta de clientes que os utilizam e de acordo com os Analistas de Testes da empresa disponíveis para a realização do estudo.

Nas próximas seções, serão apresentadas mais informações sobre os módulos que foram objeto de estudo, a ferramenta (*checklist*) elaborada e utilizada nos testes e os resultados encontrados após as aplicações.

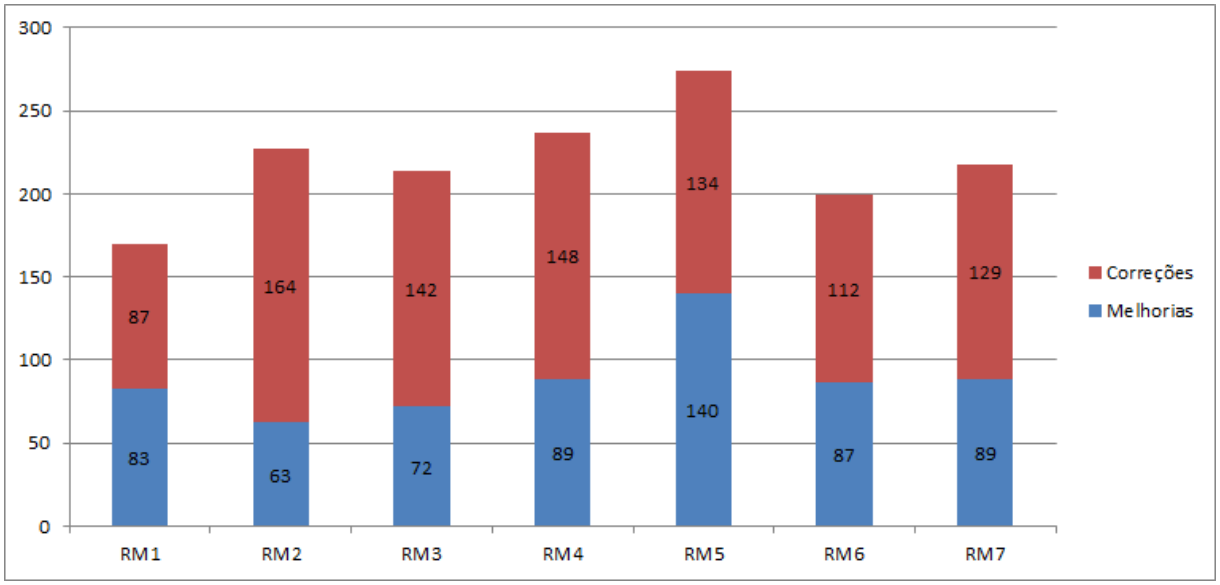
4.1 Características do sistema estudado

A suíte da Empresa A é composta por doze módulos (ferramentas) que apoiam a gestão estratégica e corporativa. Os módulos são comercializados de forma independente, mas quando utilizados juntos, possuem interação.

A plataforma em que os módulos executam é Web, com algumas extensões para mobile (*apps*). A suíte inteira possui aproximadamente cinco milhões de linhas de código em linguagem Java e suporta os bancos de dados MySql, Oracle e SQL Server.

Como as *Releases* de Manutenção (RM) possuem uma periodicidade trimestral, são liberadas em torno de três a quatro atualizações por ano. Cada *release* contempla centenas de *tickets*. O Gráfico 1 ilustra a quantidade de melhorias e correções realizadas em toda suíte nas últimas sete *releases* de manutenção, nos anos de 2016 e 2017.

Gráfico 1 - Quantidade de melhorias e correções por RM em 2016 e 2017



Fonte: Elaborado pela autora (2017).

A Empresa A possui cerca de 250 clientes. Destes, 89% utilizam o módulo M1 e 70% utilizam o módulo M2. M1 e M2 foram os módulos escolhidos para aplicação

do estudo de caso. Cada um deles possui mais de 15 anos de vida no mercado, sofrendo inúmeras melhorias ao longo desse tempo, e são os módulos que sempre se destacam entre os cinco que mais receberam alterações em todas as *releases*.

Tanto M1 quanto M2 possuem um Analista de Teste especialista na ferramenta, cada um com cinco anos de experiência nos referidos módulos. Da mesma forma, a área de Qualidade do Produto possui outros dois testadores para cada módulo, que atendem ao papel de testador com menos experiência e algum conhecimento básico nos módulos.

4.2 *Checklist*

O *checklist* foi criado em uma planilha eletrônica e estruturado de forma a agrupar as aplicações dos módulos, para que pudessem ser inseridos tantos itens quanto necessário para cada agrupamento, de acordo com as funcionalidades existentes e sua constante evolução. Além disso, o modelo de *checklist* permite que sejam adicionadas histórias de usuários e outras questões relacionadas a *layout*, usabilidade e outras categorias, conforme desejado, sendo totalmente flexível quanto ao teor dos itens de verificação.

Após estudar os módulos escolhidos, de forma a conhecer a totalidade atual de suas funcionalidades, foi elaborada uma primeira versão do *checklist* para cada módulo, contendo todas as suas aplicações (agrupamentos) e funcionalidades (itens). Os *checklists* dos dois módulos M1 e M2 ficaram com um total em torno de 130 e 150 itens de verificação, respectivamente.

Os itens do *checklist* foram criados através de navegação no sistema. Todas as telas foram acessadas e então verificadas as ações disponíveis e, com apoio do manual do produto, foram criados itens que abrangessem cadastros, edições e exclusões, até mesmo casos de uso mais complexos de forma a forçar situações. Neste último caso, inclusive foi necessário apoio dos analistas de produto da empresa para confirmar qual seria o comportamento esperado.

A Figura 13 apresenta um fragmento do *checklist* elaborado para M2. Entretanto, algumas informações foram omitidas com uma tarja roxa para preservar a identificação da empresa.

Figura 13 – Fragmento inicial do *checklist* do Módulo 2

	A	B	C	D	E	F
1	DC 356	Template Checklist [redacted]	v.1			
2			**Checklist** [redacted]		APLICAÇÃO Data:16/10/2017 Responsável:DK Versão: [redacted] Build: [redacted]	OBSERVAÇÕES
3		*Área:* Qualidade do Produto	*Código:* CKL-001	*Versão:* 01		
4	**1 – Configuração**					
5	ÍNDICE	ITEM	AÇÃO	COMPORTAMENTO ESPERADO	OK?	S = sim N = não P = parcialmente
6	1	Categorias				
7	1.1	Navegação	Realizar busca pelo campo acima da estrutura de categorias	Os resultados devem receber o foco e ficarem selecionados a cada <enter>	s	
8	1.2		Expandir e Retrair os itens da estrutra, sendo que ao menos um nó da árvore esteja diferente dos demais (fechado ou aberto)	Ao clicar em um dos botões, a estrutura primeiro iguala seus nós e depois continua aplicando a todos ao mesmo tempo a ação escolhida	s	
9	1.3	Categoria	Excluir categoria na qual tenha documentos publicados ou em elaboração.	Deve permitir excluir e possibilitar a escolha de uma outra categoria para vincular os documentos existentes.	s	
10	1.4		Excluir categoria na qual hajam documentos em Quarentena.	Deve também possibilitar a escolha de uma nova categoria, porém, seu fluxo de Quarentena deve seguir o fluxo que tinha na categoria antiga. A não ser que seja abortado e liberado novamente, então seguirá o fluxo da nova categoria.	p	Permitiu escolher outra categoria, porém, documentos em verificação ficaram órfãos de reponsável.
11	1.5	Categoria > Reprocessar	Em uma categoria existente com Distribuidores obrigatórios e documentos liberados para Distribuição, adicionar novos distribuidores obrigatórios e, então, clicar em Reprocessar.	Os documentos desta categoria devem aparecer na aplicação Distribuidor para os novos responsáveis definidos.	n	Ocorreu erro ao Reprocessar.

Fonte: Elaborado pela autora (2017).

A Figura 14 apresenta um fragmento maior do *checklist* do Módulo 2, com o intuito de exemplificar como foram descritos os cenários a serem executados. Da mesma forma que a Figura 13, a tarja roxa preserva a informação de algumas nomenclaturas do módulo.

Figura 14 - Fragmento maior do *checklist* do Módulo 2

	A	B	C	D
35	**2 – [redacted]**			
36	ÍNDICE	ITEM	AÇÃO	COMPORTAMENTO ESPERADO
37	1	Local		
38	1.1	[redacted]	Criar e publicar um documento Registro	Documento publicado no repositório e informações do Resumo coerentes.
39	1.2	[redacted]	Criar um documento que seja conjunto de outro. Liberar e gerar nova versão dele.	Não deve ser possível gerar nova versão dele pois como é conjunto de outro, será criada nova versão apenas quando o documento 'pai' for versionado. Então é criada pendência para elaboração de nova versão do documento filho automaticamente.
40	1.3	Gerar Código	Ao criar um documento com categoria que possua Numeração, clicar em gerar código. Após inativar o botão pelo propriedade 'Habilita/Desabilita o botão Gerar Código'.	1) Deve queimar os códigos. 2) Deve desabilitar o botão.
41	1.4	Versionamento	Criar documento com categoria que tenha configurado prazo de Vencimento, e publicar.	Após publicado, o documento deve ter no campo Validade a respectiva data de acordo com a publicação e os meses configuados na categoria.
42	1.5	Referência	Adicionar um documento referência a outro e marcar o checkbox "Auto". Gerar nova versão da Referência e inspecioná-lo no primeiro documento conferindo se a versão ficou a mesma ou atualizada. Repetir o procedimento com outra referência e o check "Auto" desmarcado, gerando nova versão da referência e inspecionando o primeio documento novamente.	O checkbox "Auto" se refere a manter a referência com versão atualizada ou não. O exemplo que estava marcado deve mostrar o doc referência sempre na última versão. Enquanto o exemplo desmarcado deve mostrar sempre com a versão que estava corrente quando adicionado.
43	1.6	Atributos	Atribuir atributos à Categoria e criar um documento com esta categoria.	Deve adicionar os atributos da Categoria ao Documento.
44	1.7	Autorizações	Adicionar alguns usuários às autorizações e liberar para Quarentena onde exista um controlador que não esteja nas Autorizações.	O controlador deve receber a pendência em sua agenda e conseguir abrir o doc, pois as autorizações devem considerar os controladores implicitamente.
45			Adicionar na Subscrição um usuário que não esteja nas Autorizações.	Ao publicar o documento, o subscrito recebe uma notificação com o link para acessar o documento. Porém, ao abri-lo, o sistema deve mostrar que é um item restrito, impedindo esse usuário não autorizado de vê-lo.
46	1.8	Substituir	Selecionar documento Em Elaboração e clicar em Substituir, escolher outro documento para ser o substituto.	Ao descarregar o documento, deve abrir o substituto.
47	1.9	Abortar	Selecionar um documento que esteja em Quarentena e clicar em Abortar.	O status do documento deve voltar para Em Elaboração e as pendências de Quarentena devem ser removidas.
48	1.10	Correções	Selecionar documento rejeitado e clicar em Correções	Deve ser possível visualizar as observações da rejeição.
49	1.11	Subscrever	Em qualquer documento, adicionar usuários na subscrição através do atalho do menu "Subscrever".	No resumo do documento os novos subscritos devem constar.

Fonte: Elaborado pela autora (2017).

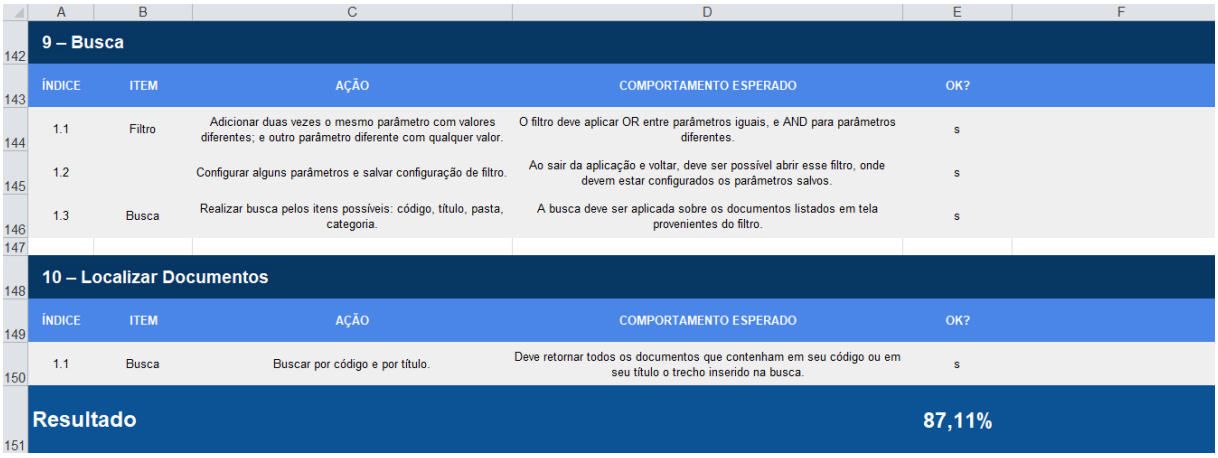
Como pode ser visto no item 1.7 da Figura 14, sugere-se a seguinte ação sobre o item “Autorizações”: “Adicionar alguns usuários às autorizações e liberar para Quarentena onde exista um controlador que não esteja nas Autorizações”. Percebe-se que todo o passo a passo para realizar a ação não é explícito, pois implicitamente é necessário entrar em alguns editores, selecionar itens, confirmar seleção e confirmar edição para que a ação como um todo possa ser executada. Entretanto, o *checklist* atém-se a um enunciado menos detalhista e mais focado no objetivo da ação e resultado esperado. Já a coluna Comportamento Esperado do item 1.7, explica que o sistema já deve considerar automaticamente na lista de autorizados os usuários controladores da Quarentena dos documentos.

Assim como o item 1.7, todos os demais se referem ao comportamento do sistema em determinadas situações, ou seja, foco nas regras de negócio. Contudo, futuras versões de *checklist* podem passar a contemplar outros tipos de verificação, como: validação dos campos obrigatórios e tipos de dados aceitos em cada campo; clareza e corretude das mensagens exibidas ao usuário; confirmação das edições em tela comparando com atualização dos registros em banco de dados; usabilidade e padrões de nomenclaturas de campos e botões, bem como, ícones padrões.

Ao final da planilha, para obter um resultado geral dos testes, foi definida uma métrica de avaliação para os itens do *checklist*. Cada item possui uma coluna onde deve ser indicada uma classificação resultante do teste, que pode ser: Sim (S), atende Parcialmente (P) e Não atende (N). Estas classificações possuem o seguinte valor atribuído: 1, 0,5 e 0, respectivamente.

No final do *checklist*, uma linha com o total indica o resultado geral do teste, ou seja, contabilização do percentual de conformidade dos itens, aplicando uma média simples: a soma de todos os resultados dividida pelo total de itens e multiplicando por 100. A Figura 15 demonstra este resultado.

Figura 15 - Resultado geral do *checklist*



	A	B	C	D	E	F
142	**9 – Busca**					
143	ÍNDICE	ITEM	AÇÃO	COMPORTAMENTO ESPERADO	OK?	
144	1.1	Filtro	Adicionar duas vezes o mesmo parâmetro com valores diferentes; e outro parâmetro diferente com qualquer valor.	O filtro deve aplicar OR entre parâmetros iguais, e AND para parâmetros diferentes.	s	
145	1.2		Configurar alguns parâmetros e salvar configuração de filtro.	Ao sair da aplicação e voltar, deve ser possível abrir esse filtro, onde devem estar configurados os parâmetros salvos.	s	
146	1.3	Busca	Realizar busca pelos itens possíveis: código, título, pasta, categoria.	A busca deve ser aplicada sobre os documentos listados em tela provenientes do filtro.	s	
147						
148	**10 – Localizar Documentos**					
149	ÍNDICE	ITEM	AÇÃO	COMPORTAMENTO ESPERADO	OK?	
150	1.1	Busca	Buscar por código e por título.	Deve retornar todos os documentos que contenham em seu código ou em seu título o trecho inserido na busca.	s	
151	**Resultado**				**87,11%**	

Fonte: Elaborado pela autora (2017).

Desta forma, o modelo de *checklist* também está apto a receber constantes atualizações, visto que os módulos recebem constantemente novas funcionalidades; bem como, a equipe de Qualidade do Produto pode adicionar cenários específicos, que são conhecidos ao realizar análises de erros de situações muito pontuais apontadas eventualmente por clientes.

Os *checklists* dos dois módulos foram versionados e publicados na ferramenta de gerenciamento eletrônicos de documentos que a Empresa A utiliza. Assim, os responsáveis podem avaliar a continuidade de sua aplicação ou não.

4.3 Aplicações dos testes e análise dos resultados

A aplicação dos testes ocorreu em quatro momentos durante dois meses, no segundo semestre de 2017. Essa aplicação foi feita na *Release* de Manutenção que esteve em desenvolvimento no período de execução deste trabalho, e está sendo nomeada de RM7 nas análises abaixo.

A seguir, são detalhados os dados colhidos a partir das quatro aplicações dos testes nos dois módulos.

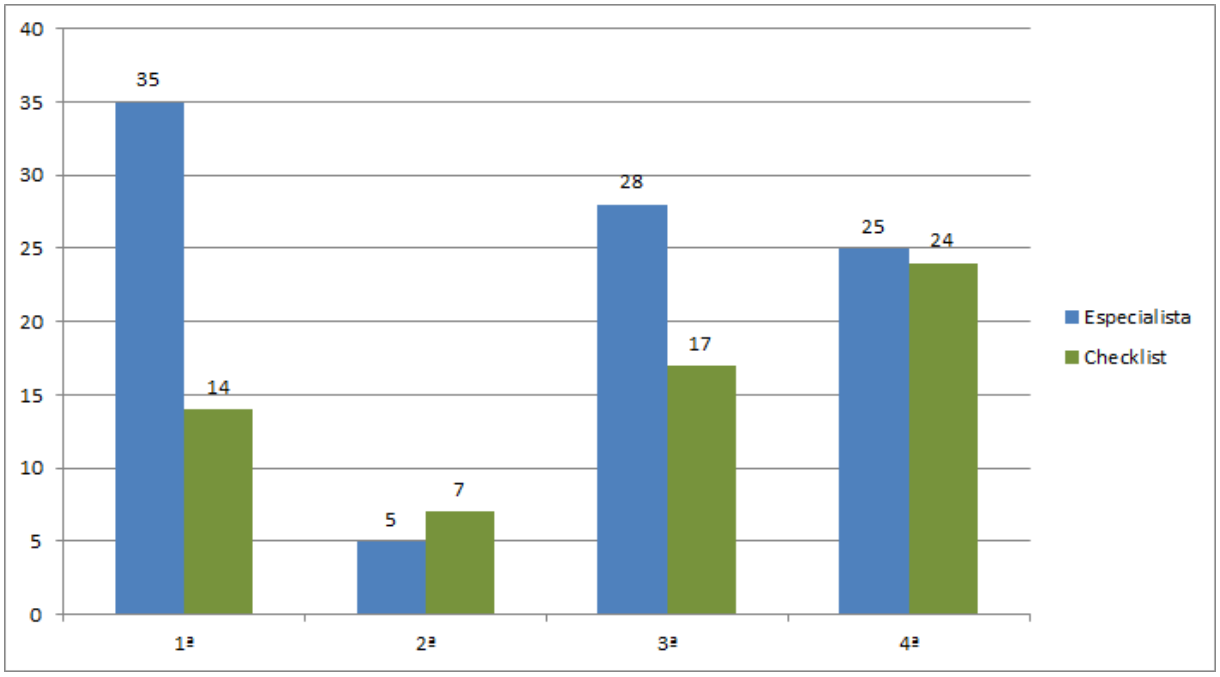
4.3.1 Aplicações Módulo 1

Para o Módulo 1, após quatro rodadas de testes, pode-se observar como resultados um total maior de erros encontrados pelo testador especialista em comparação aos resultados do testador que utilizou o *checklist* e possui menos conhecimento no módulo.

O Gráfico 2 ilustra os totais encontrados em cada rodada, por cada testador.

A segunda aplicação dos testes teve uma exceção devido a um conflito de demandas na Empresa A, onde, os dois testadores acabaram testando apenas dois agrupamentos dos 5 que o módulo possui.

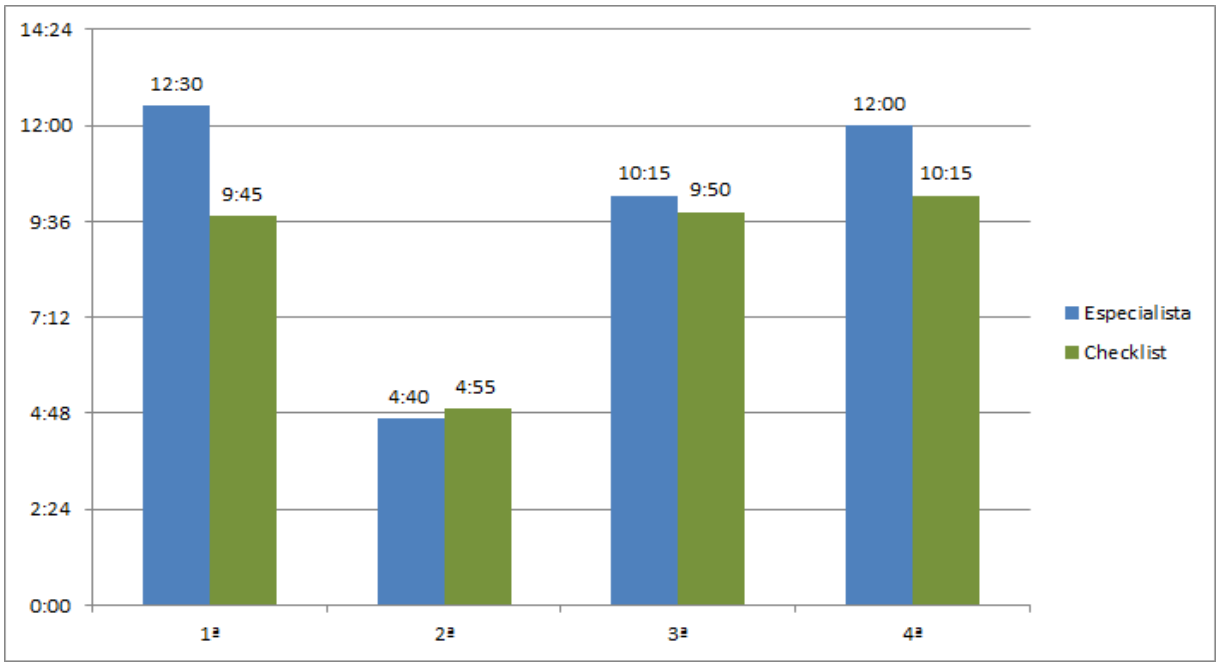
Gráfico 2 - Resultados testes M1 - Erros



Fonte: Elaborado pela autora (2017).

Entretanto, através do Gráfico 3, onde é possível visualizar o tempo despendido por cada testador em cada rodada, pode-se constatar que os testes do especialista também levam, em geral, mais tempo. A escala de tempo é apresentada em horas e minutos.

Gráfico 3 - Resultados testes M1 - Tempo



Fonte: Elaborado pela autora (2017).

No entanto, dada a proporção de erros encontrados e o maior tempo despendido pelo especialista, percebe-se que, em média, houve um ganho considerável neste formato de teste; visto que, mesmo levando 13,67% mais tempo, o total de erros encontrados foi 50% maior.

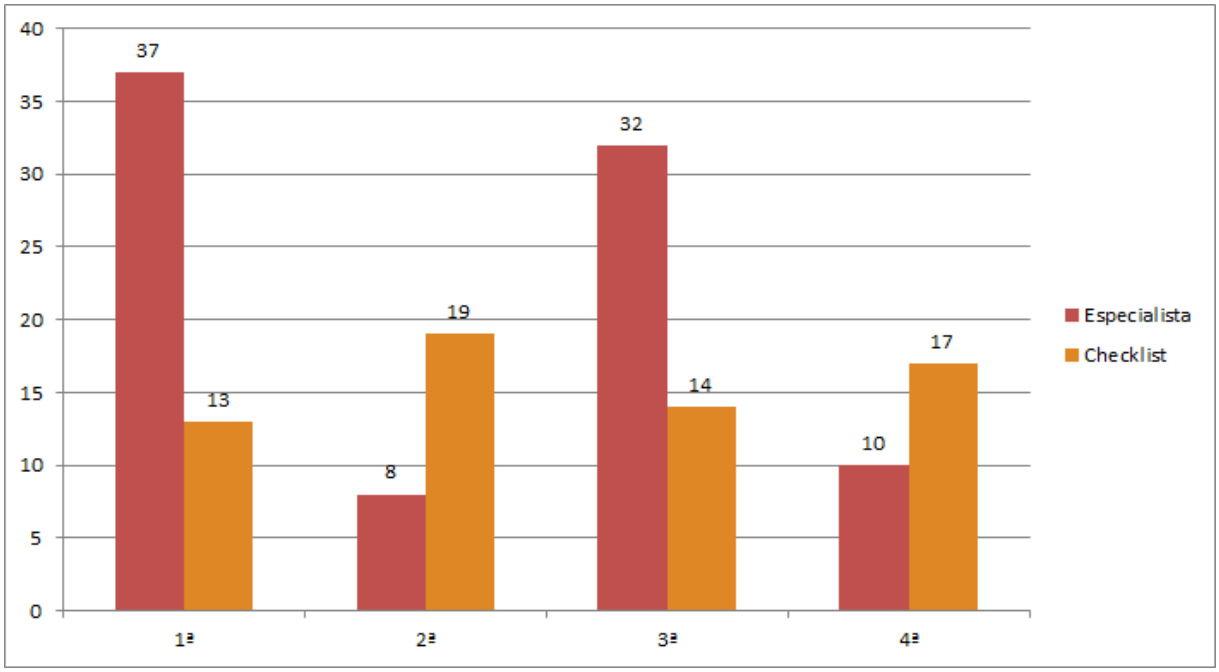
4.3.2 Aplicações Módulo 2

O Módulo 2 também passou pelas quatro rodadas de teste e todos se deram de forma completa. Porém, o papel de testador especialista foi exercido por uma pessoa diferente na 2ª e 4ª aplicações. Isto se deu devido ao fato de este módulo possuir um segundo especialista.

Contudo, o Módulo 2 não teve uma uniformidade na proporção de erros encontrados pelo especialista e pelo *checklist* nas quatro rodadas.

No Gráfico 4 nota-se que, enquanto na primeira rodada o resultado do especialista foi mais de 100% maior que o erros encontrados pelo *checklist*, na segunda rodada ocorreu o inverso. A última rodada também teve um resultado significativo para o formato de testes com *checklist*, onde chegou a encontrar 70% mais erros. Acredita-se que isso possa ter ocorrido devido ao nível de experiência do segundo especialista ser inferior ou insuficiente.

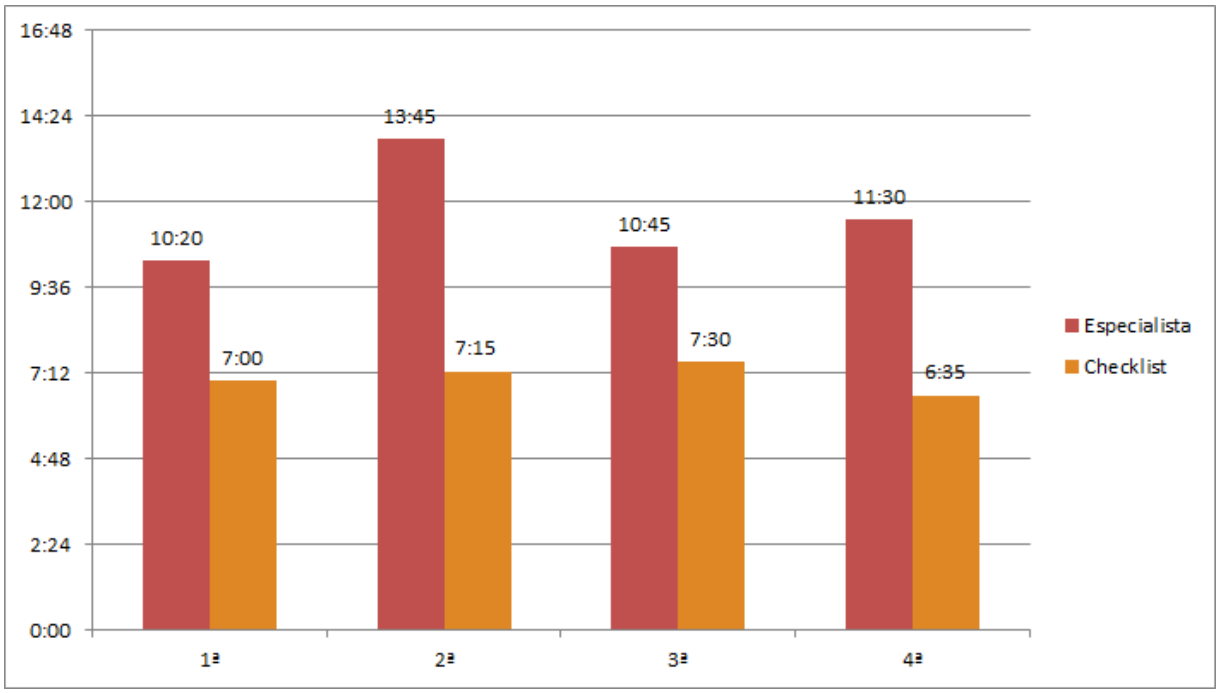
Gráfico 4 - Resultados testes M2 - Erros



Fonte: Elaborado pela autora (2017).

Já o tempo empregado no Módulo 2 se manteve equivalente para aplicação do *checklist*, enquanto que o tempo do especialista foi relativamente maior em todas as rodadas, chegando a atingir 89% mais tempo na segunda rodada, conforme demonstra o Gráfico 5.

Gráfico 5 - Resultados testes M2 - Tempo



Fonte: Elaborado pela autora (2017).

De forma geral, para o Módulo 2, o formato de teste com especialista não foi tão vantajoso. Embora este tenha encontrado em média 38% mais erros que o testador com *checklist*, o especialista levou 63% mais tempo para executar o teste de todo módulo.

4.4 Avaliação dos participantes

A aplicação do novo formato de testes guiados na empresa alvo do estudo de caso gerou algumas mudanças na rotina dos participantes enquanto executavam o proposto. Toda mudança pode gerar desconfortos, dúvidas, insatisfações; ou, pelo contrário, motivações.

Como todo o processo de teste na Empresa A ocorre de forma totalmente manual, foram elaborados dois questionários para que pudesse ser averiguada a opinião dos testadores envolvidos na execução dos testes. Os questionários estão explícitos na íntegra nos Apêndices A e B.

Em relação aos testes de especialistas, conforme questionário do Apêndice A, os principais relatos foram de desconforto ao ter que realizar o teste de todo módulo sem nenhum guia. Entretanto, um especialista relatou preferir este formato, visto que “É possível sentir mais o módulo”, cita ele. Este comentário coincide com o especialista que encontrou mais erros, e também relatou conseguir aplicar o teste com a mesma profundidade em todas as funcionalidades. Diferentemente dos demais, que confirmaram haver diferença na uniformidade dos testes nos diversos itens do módulo.

Já quanto a conclusão dos testes e um resultado geral de conformidade do módulo, os especialistas consensam que é difícil mensurar e conseguir dar um *feedback* quantitativo aos superiores quanto à estabilidade dos módulos.

Em relação aos testadores que realizaram as aplicações com *checklist*, conforme questionário do Apêndice B, os mesmos relataram que a planilha foi um guia e facilitou a execução da atividade, visto que estes participantes não possuíam muito conhecimento nos módulos. Por outro lado, também houveram relatos de que o *checklist* poderia estar limitando a profundidade dos testes, bem como, pode prejudicar a completude quando o módulo não está totalmente detalhado nos itens do *checklist*.

Quanto aos resultados proporcionados pelo *checklist*, os testadores os consideraram um pouco superficiais, basicamente por dois motivos:

- estão elencados na planilha as funcionalidades básicas e principais, desta forma é testada a realidade dos usuários de forma mais simples, sem situações inesperadas;
- e, pelo fato de os erros apontados possuírem todos os mesmo peso, muitas vezes podendo ser repetitivo para vários itens. Isto acaba somando um número maior de erros, entretanto, podem ser de correção simples e única, ao passo que um único outro erro pode ser mais grave e mais complexo. Ou seja, na planilha não há essa distinção, portanto o percentual foi considerado não fidedigno à situação real do módulo.

Contudo, a resposta de um dos testadores que utilizou *checklist* complementou de forma mais resignada “A sensação não é ruim, pois apesar de ser um modelo engessado de teste a quantidade de erros encontrados foi razoável e é a realidade do módulo”.

4.5 Erros reportados após liberação da RM7

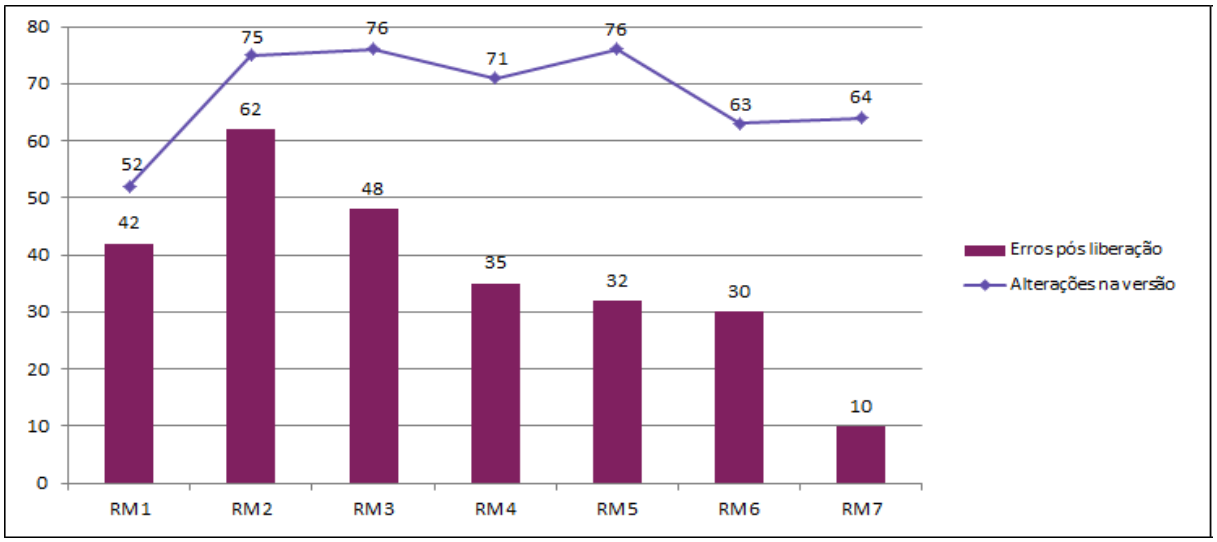
Além de realizar a comparação dos resultados dos testes nos dois formatos propostos, averiguou-se os resultados perante os clientes, após a liberação da *Release* de Manutenção que recebeu a execução dos dois formatos de testes. No Gráfico 6 essa *release* é representada pela RM7.

Entende-se que quanto maior a diferença entre o total de alterações de uma *release* e seu respectivo total de erros reportados após liberação, mais eficiente foi seu processo de desenvolvimento de software.

Nos dados apresentados nos Gráficos 6 e 7, visualizam-se os resultados de alterações e erros referentes a sete *Releases* de Manutenção, que correspondem às releases dos dois últimos anos (2016 e 2017), até o presente momento.

No Gráfico 6 pode-se observar a evolução da eficiência no processo para o Módulo 1. Nota-se que com o passar do tempo a diferença entre os dois dados foi aumentando. Especialmente para a RM7, que foi a *release* que recebeu as rodadas de teste do estudo de caso. Porém, essa diferença ainda não pode ser atribuída a um possível ganho de qualidade devido aos testes empregados pelo estudo de caso. Pois, conforme os dados históricos, essa melhoria nos resultados já era uma tendência. Além disso, a RM7 está liberada para clientes há pouco tempo; portanto, esse resultado ainda pode ser alterado.

Gráfico 6 - Relação Alterações realizadas X Erros reportados - Módulo 1

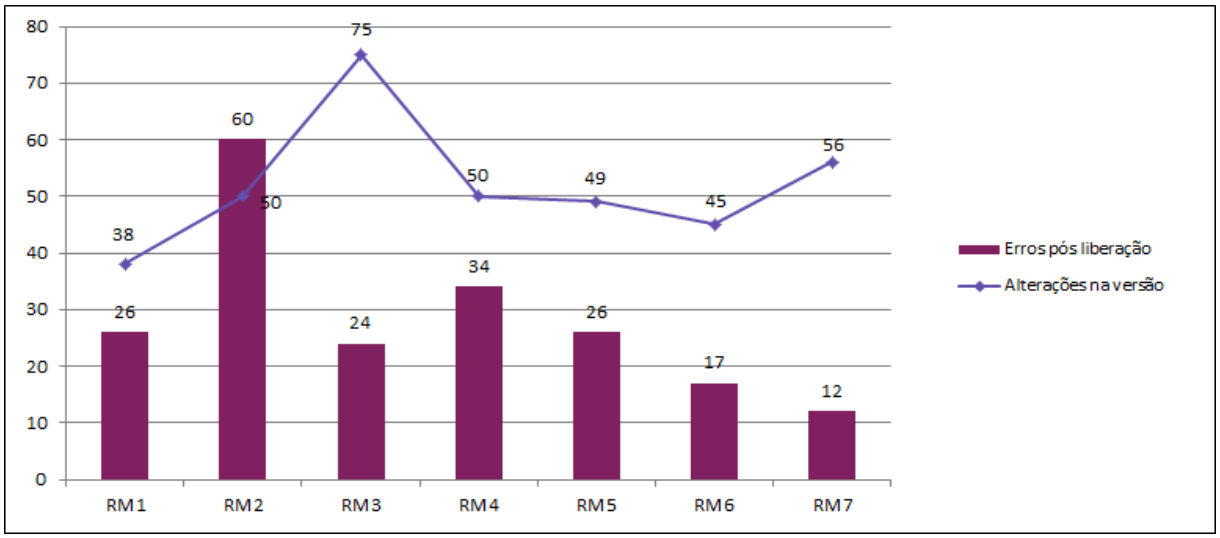


Fonte: Elaborado pela autora (2017).

Já o Módulo 2, conforme Gráfico 7, apresentou uma situação crítica na RM2 (2016), quando foram realizadas 50 alterações e, após liberada, foram reportados 60 *bugs*. Ou seja, as alterações tiveram muito impacto no módulo e o processo de desenvolvimento de software não foi eficiente na época.

Contudo, desde a RM4 houve uma estabilização na relação alterações *versus* erros reportados e, desde então, vem apresentando uma melhoria crescente na eficiência do processo. Espera-se que, após mais algumas semanas, o resultado de erros para RM7 também permaneça com uma boa diferença em relação às suas alterações.

Gráfico 7 - Relação Alterações realizadas X Erros reportados - Módulo 2



Fonte: Elaborado pela autora (2017).

Apesar de já se ter em mãos o total de alterações realizadas nos dois módulos na RM7, ainda não é possível analisar sua relação com os erros reportados. Como a *release* foi liberada recentemente e ainda não são todos os clientes que atualizaram para esta versão, novos erros podem ser cadastrados no decorrer das próximas semanas. Ao passar de pelo menos mais um mês, será possível fazer uma análise mais realista deste resultado e sua diferença para com as alterações.

4.6 Comparação dos formatos de teste

Com base nas análises dos resultados dos dois formatos de testes e nas respostas ao questionário dos participantes do estudo, pode-se determinar alguns prós e contras para cada formato, como pode ser visualizado no Quadro 4.

Quadro 4 - Comparação entre os formatos de testes

Pontos	**Testes exploratórios com Especialista**	**Testes estruturados via *checklist***
Positivos	Em média, encontram mais erros; Possibilitam um aprofundamento maior; Dão liberdade à criatividade do testador para simular cenários complexos;	Encontram muitos erros também, mesmo que não tantos quanto o Especialista; Permitem que não se dependa dos Especialistas, outros testadores podem aplicar os testes; Garantem padronização (sempre serão testados todos os itens do *checklist* de acordo com seu comportamento esperado que também consta no *checklist*); Informam um resultado final quanto à estabilidade dos módulos.

Negativos	Em geral, levam mais horas para serem realizados; A maioria dos testadores sentiu dificuldade de se auto-organizar para realização do teste do módulo completo; Não são uniformes (alguns itens são mais investigados que outros); Fatores externos (particulares ou concorrência de demandas) também prejudicam a uniformidade dos testes e até sua realização por completo; Não permite medir a estabilidade dos módulos.	O aprofundamento dos testes é menor, pois seguindo um guia são testadas apenas as ações sugeridas nele; Necessidade de investir tempo em atualização do *checklist*; O resultado final pode não ser muito realista, pois um único e superficial problema que afeta vários pontos do módulo, porém, é de correção simples e única, pode acabar demonstrando uma situação muito crítica ao final.

Fonte: Elaborado pela autora (2017).

A partir disso, conclui-se que o formato de testes estruturado é uma boa alternativa para situações onde a empresa não pode contar com especialistas, ou quando não os possui. Mesmo que não encontrem todos os erros, ainda encontram alguns e garantem a completude dos testes básicos em um módulo.

Ainda assim, quando é possível executar testes exploratórios de especialistas, estes devem ser executados também, pois trazem resultados melhores.

Indiscutivelmente, são necessários testes integrados, seja no formato com especialista ou com *checklist*, pois atualmente os testes da Empresa A atingem apenas funcionalidades específicas implementadas, devido ao teste ser orientado via ticket. Porém, isso não garante que as demais funcionalidades do módulo que não sofreram alterações não tenham sido impactadas, resultando em erros encontrados pelos próprios clientes e dando margem para sua insatisfação.

Quanto ao resultado final do *checklist* não ser tão realista, é importante que, se utilizado, deve receber adição de mais itens de verificação, principalmente, cenários mais complexos de testes.

Para organizar a prática de atualização do *checklist*, podem ser incorporadas ao processo de desenvolvimento atividades que preveem a necessidade de atualização de item de *checklist*. Como, por exemplo, durante análises de bugs, pode haver uma etapa que considere a adição do cenário do mesmo ao *checklist* do respectivo módulo. Outro momento que pode haver essa atividade é durante testes de melhorias; ao passo que é testada e liberada uma melhoria, um novo item é acrescentado ao *checklist*.

Com o passar do tempo, os *checklists* poderão se tornar extensos. Contudo, haverão momentos em que não será necessário testar todo módulo. Pode-se averiguar, de acordo com os *commits* e classes do produto afetadas, quais foram as aplicações que mais sofreram alterações. A partir disso, realiza-se o teste via *checklist* apenas para alguns agrupamentos. Ou, em outra situação, pode-se aplicar uma espécie de auditoria, sorteando apenas alguns itens e testando por amostragem.

5 CONSIDERAÇÕES FINAIS

Com base na pesquisa bibliográfica realizada, sobre processo de desenvolvimento de software, metodologias que qualificam e agilizam os processos, percebe-se que todos os fatores são importantes para a busca por maior qualidade nos sistemas gerados. A qualidade dos produtos não pode ficar dependente dos testes de software, visto que estes não conseguem detectar todos os problemas possíveis.

A integração contínua, as entregas periódicas, além de testes de regressão automatizados, são todos itens importantes que contribuem diretamente na qualidade final dos sistemas. Contudo, a figura do testador ainda é fundamental, a sensibilidade e criatividade humana permitem avançar para dimensões inesperadas de cenários de testes.

Através das comparações realizadas neste trabalho, pode-se detectar que até mesmo o mínimo de estruturação, através da criação das planilhas como guia para testes, limitou a interação mais livre e aprofundada que o humano pode realizar nos testes de software.

Também, por conta disso, constata-se que apenas formas automatizadas de testes podem ser precárias para manter a qualidade de um sistema. É preciso que em alguma etapa do processo os testes passem aos olhos humanos.

Percebeu-se, também, uma diferença nos resultados dos testes de acordo com o perfil dos participantes. O perfil não era foco do estudo, mas conforme

descrito no capítulo de Introdução do presente trabalho, algumas características pessoais podem indicar melhores perfis para exercer a função de testador nas empresas fabricantes de software.

Para fins operacionais da Empresa A, o estudo de caso apontou que os testes gerais de um módulo completo podem levar de 1,5 a 2 dias dedicados. Considerando que uma *Release* de Manutenção possui uma periodicidade de 3 meses e a suíte é composta de doze módulos, os testes completos para todos os módulos, no formato especialista levariam em torno de um mês. Entretanto, estes dependem de pessoas específicas: os testadores especialistas. Em contrapartida, os testes via *checklist*, mesmo demorando um tempo parecido - porém, menor - não ficam engessados à disponibilidade de algumas pessoas para realização. Bem como, os especialistas podem por vários motivos ficar ausentes da empresa em momentos cruciais, quando torna-se ainda mais importante a existência de um *checklist* atualizado, contendo todas as funcionalidades principais com seus respectivos comportamentos esperados.

É necessário que a Empresa A avalie também o tempo necessário para manutenção dos *checklists*, pois deverão sofrer alterações a cada *release* para que se mantenham atualizados. Caso não sejam atualizados adequadamente, a aplicação de um *checklist* com exposição do resultado pode se tornar um problema, pois o dado será infiel à realidade do sistema, visto que nem todas as funcionalidades estarão elencadas.

Contudo, através de todos os dados levantados neste trabalho e análises realizadas, constatou-se que a Empresa A realiza melhorias no seu processo de desenvolvimento ao longo do tempo, buscando cada vez mais atingir uma eficiência maior e, consequentemente, mais qualidade nos seus produtos. Acredita-se que após este estudo de caso, suas práticas de testes venham sofrer melhorias, principalmente quanto a testes integrados no período de quarentena das *releases*.

Mesmo assim, ainda sugere-se que seja reavaliada a possibilidade de automatização de testes de regressão, pois muitos dos itens dos *checklists* são ações básicas que, se automatizadas, reduziriam consideravelmente os testes integrados durante a quarentena das *releases*.

REFERÊNCIAS

AGILE MANIFESTO. Manifesto for Agile Software Development. Disponível em <http://agilemanifesto.org/>. Acesso em: 03 jun. 2017.

ASSOCIAÇÃO BRASILEIRA DE NORMAS TÉCNICAS. **NBR ISO/IEC 25010:** Engenharia de software - Qualidade de produto de software. Rio de Janeiro: ABNT, 20011.

BARBOSA, Filipe B.; TORRES, Isabelle V. O Teste de Software no Mercado de Trabalho. **Revista Tecnologias em Projeção**. Brasília, v.2, p. 49-52, Junho 2011.

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

CRUZ, Fábio. **Scrum e Agile em projetos:** guia completo. Rio de Janeiro: Brasport, 2015. E-book. Disponível em <https://books.google.com.br/books?id=NW2LBgAAQBAJ&printsec=frontcover&hl=pt-BR#v=onepage&q&f=false>. Acesso em: 12 nov. 2017

FOWLER, Martin, 2006. **Continuous Integration**. Disponível em <https:// martinfowler.com/articles/continuousIntegration.html>. Acesso em: 24 mai. 2017.

GIL, Antonio C. **Métodos e técnicas de pesquisa social**. 6. ed. São Paulo: Atlas, 2008.

KALINOWSKI, Marcos; SPÍNOLA, Rodrigo O. Introdução à Inspeção de Software: Aumento da qualidade através de verificações intermediárias. **Engenharia de Software Magazine**. Edição Especial, p. 68-74, 2007. Disponível em <https://profandreluisbelini.files.wordpress.com/2016/02/revista-engenharia-de-software-ano-1-1c2ba-edic3a7c3a3o.pdf>. Acesso em: 26 mai. 2017.

KLEIN, Amarolinda Z. et al 2015. **Metodologia da pesquisa em administração:** uma abordagem prática. São Paulo: Atlas, 2015. E-book. Disponível em <https://integrada.minhabiblioteca.com.br/#/books/9788522495313/cfi/0!/4/4@0.00:9.21>. Acesso em: 29 mai. 2017.

KOSCIANSKI, André; SOARES, Michel S. **Qualidade de Software:** aprenda as metodologias e técnicas mais modernas para o desenvolvimento de software. 2. ed. São Paulo: Novatec Editora, 2007.

MARCONI, Maria de A.; LAKATOS, Eva M. **Fundamentos de metodologia de pesquisa**. 7. ed. São Paulo: Atlas, 2010. E-book. Disponível em <https://integrada.minhabiblioteca.com.br/#/books/9788522484867/cfi/0!/4/2@100:0.00>. Acesso em: 29 mai. 2017.

MECENAS, Ivan; DE OLIVEIRA, Viviane. **Qualidade de Software:** uma metodologia para homologação de sistemas. Rio de Janeiro: Alta Books, 2005.

MELLO, Rafael M. de et al. 2012. **Checklist-based Inspection Technique for Feature Models Review**. Disponível em <http://s3.amazonaws.com/academia.edu.documents/34714573/0015.pdf?AWSAccessKeyId=AKIAIWOWYYGZ2Y53UL3A&Expires=1495738258&Signature=vBYuLB09O99yRG%2Bt6Oi62lTBIrA%3D&response-content-disposition=inline%3B%20filename%3DChecklist-based_Inspection_Technique_for.pdf>. Acesso em: 25 mai. 2017.

PAPO, José. **O fim do "Mas na minha máquina funciona!"**. Seminário Mineiro de Qualidade de Software, 2010.

PFLEEGER, Shari L. **Engenharia de Software:** teoria e prática. 2. ed. São Paulo: Prentice Hall, 2004.

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

PRESSMAN, Roger S.; MAXIM, Bruce R. **Engenharia de Software:** uma abordagem profissional. 8. ed. Porto Alegre: AMGH, 2016. E-book. Disponível em <https://books.google.com.br/books?id=wexzCwAAQBAJ&printsec=frontcover&hl=pt-BR#v=onepage&q&f=false>. Acesso em: 17 mai. 2017.

PRIKLADNICKI, Rafael; WILLI, Renato; MILANI, Fabiano. **Métodos Ágeis para Desenvolvimento de Software**. Porto Alegre: Bookman, 2014. E-book. Disponível em <https://books.google.com.br/books?id=8rQABAAAQBAJ&printsec=frontcover&hl=pt-BR#v=onepage&q&f=false>. Acesso em: 17 mai. 2017.

PRODANOV, Cleber C.; FREITAS Ernani C. **Metodologia do trabalho científico:** métodos e técnicas da pesquisa e do trabalho acadêmico. 2. ed. Novo Hamburgo: Feevale, 2013. E-book. Disponível em <http://www.feevale.br/Comum/midias/8807f05a-14d0-4d5b-b1ad-1538f3aef538/E-book%20Metodologia%20do%20Trabalho%20Cientifico.pdf>. Acesso em: 29 mai. 2017.

REZENDE, Denis A. **Engenharia de Software e Sistemas de Informação**. 3. ed. Rio de Janeiro: BRASPORT, 2005. E-book. Disponível em <https://books.google.com.br/books?id=rtBvl_L-1mcC&printsec=frontcover&hl=pt-BR&source=gbs_ge_summary_r&cad=0#v=onepage&q&f=false>. Acesso em: 16 mai. 2017.

RIOS, Emerson; MOREIRA, Trayahú. **Teste de Software**. 3. ed. Rio de Janeiro: Alta Books, 2013. E-book. Disponível em <https://books.google.com.br/books?id=I2a2BAAAQBAJ&printsec=frontcover&hl=pt-BR#v=onepage&q&f=false>. Acesso em: 23 mai. 2017.

RUBIN, Kenneth S. **Essential Scrum:** a practical guide to the most popular agile process. New Jersey: Addison-Wesley Professional, 2012. E-book. Disponível em <https://books.google.com.br/books?id=HkXX65VCZU4C&printsec=frontcover&hl=pt-BR#v=onepage&q&f=false>. Acesso em: 17 mai. 2017

SALAZAR, Wagner. **Gerenciamento de projetos utilizando métodos ágeis:** teoria e prática. 1. ed. Belo Horizonte, MG, 2016. E-book. Disponível em <https://books.google.com.br/books?id=x7QoDQAAQBAJ&printsec=frontcover&hl=pt-BR#v=onepage&q&f=false>. Acesso em: 17 mai. 2017

SOFTWARE ENGINEERING INSTITUTE. **CMMI para Desenvolvimento – Versão 1.2**. Pittsburgh: SEI, 2006. Disponível em <http://www.sei.cmu.edu/library/assets/whitepapers/cmmi-dev_1-2_portuguese.pdf>. Acesso em: 22 mai. 2017.

SOMMERVILLE, Ian. **Engenharia de Software**. 8. ed. São Paulo: Pearson Addison-Wesley, 2007.

WAZLAWICK, Raul S. **Engenharia de Software:** conceitos e práticas. Rio de Janeiro: Elsevier, 2013.

APÊNDICES

APÊNDICE A - Questionário aplicado aos testadores especialistas

Para averiguar os impactos de uma mudança de processo no formato de aplicação dos testes, bem como, verificar a opinião das pessoas envolvidas, foram elaboradas as perguntas abaixo e aplicadas aos participantes que realizaram o papel de testador especialista.

1. Qual o sentimento em relação ao teste de especialista executado sem nenhum tipo de guia?
2. Todas as funcionalidades foram igualmente abordadas e aprofundadas durante o teste?
3. Após finalizado o teste, qual o sentimento em relação ao total de problemas encontrados? é possível mensurar sua proporção em relação ao módulo todo?

APÊNDICE B - Questionário aplicado aos testadores que utilizaram *Checklist*

Para averiguar os impactos de uma mudança de processo no formato de aplicação dos testes, bem como, verificar a opinião das pessoas envolvidas, foram elaboradas as perguntas abaixo e aplicadas aos participantes que realizaram o papel de testador com pouco conhecimento no módulo e utilizaram a ferramenta de *checklist*.

1. Qual o sentimento ao ter que executar o teste de um módulo inteiro, entretanto, com um guia?
2. Houve distinção no aprofundamento do teste de cada item do *checklist*?
3. Após finalizado o teste, qual o sentimento quanto ao total de erros encontrados? É possível afirmar que o percentual final é fidedigno à realidade do módulo?