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

Processo de gerenciamento de teste: Exemplo de configuração e uso de processo ágil de desenvolvimento de software com elementos para gerenciamento de teste

Matheus B. Santos

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

Orientador
Prof. Dr. Fernando Albuquerque

Brasília
2023

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

Processo de gerenciamento de teste: Exemplo de configuração e uso de processo ágil de desenvolvimento de software com elementos para gerenciamento de teste

Matheus B. Santos

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

Prof. Dr. Fernando Albuquerque (Orientador)
CIC/UnB

Prof. Dr. Edison Ishikawa
CIC/UnB

Prof.a Dr.a Edna Dias Canedo
CIC/UnB

Prof. Dr. Marcelo Grandi Mandelli
Coordenador do Bacharelado em Ciência da Computação

Brasília, 20 de dezembro de 2023

Dedicatória

Dedico este trabalho aos meus amados pais, Délia e Sérgio, cujo amor, apoio incondicional e constante incentivo foram fundamentais em cada passo deste caminho acadêmico. Ao meu irmão, Ítalo Emanuel, agradeço pelo suporte valioso e pela presença constante em todos os desafios. Aos dedicados professores, verdadeiros mentores neste percurso, expresso minha profunda gratidão pela partilha do conhecimento, pela orientação e pelo papel crucial na minha jornada de aprendizado e crescimento pessoal. Cada um de vocês foi um pilar essencial na construção deste trabalho e no meu desenvolvimento como estudante e ser humano.

Agradecimentos

Agradeço a minha família, cujo apoio inabalável e dedicação me proporcionaram uma educação de qualidade e me deram a oportunidade de me dedicar aos meus estudos. Seu apoio incondicional foi fundamental para que eu chegasse até aqui. Agradeço também aos meus professores, cuja sabedoria e orientação foram essenciais para o meu crescimento acadêmico. Um agradecimento especial ao professor Fernando Albuquerque, que aceitou ser o orientador deste trabalho. Sua paciência, apoio e orientação durante todo o processo de desenvolvimento deste trabalho foram inestimáveis. A todos vocês, meu sincero obrigado.

Resumo

As aplicações *web*, essenciais em diversas atividades diárias, estão se tornando cada vez mais complexas. Isso torna crucial a garantia de sua qualidade por meio de testes robustos e uma ampla cobertura de suas funcionalidades. O gerenciamento de testes de *software*, um processo que visa assegurar a qualidade dos testes e do *software*, contribui para o desenvolvimento do *software* dentro do prazo estipulado. Este trabalho objetiva a configuração de uma metodologia ágil de desenvolvimento de *software*, incorporando atividades e artefatos de gerenciamento de testes de *software*. Para aprimorar a garantia e a qualidade dos produtos de *software*, o *Personal Extreme Programming* (PXP) será configurado para integrar a abordagem de gerenciamento de teste *Kungfu Testing*, projetada para se harmonizar com as abordagens ágeis de desenvolvimento de *software*. A aplicação do processo de desenvolvimento configurado é exemplificada pelo desenvolvimento de uma parte de uma aplicação *web*, realizando suas atividades e confeccionando seus artefatos. Os resultados deste trabalho demonstram que a integração do gerenciamento de testes ao PXP pode melhorar a prática de teste de *software* e contribuir para a entrega de um produto mais confiável. No entanto, essa integração também apresenta desafios, como o aumento das atividades de desenvolvimento e a necessidade de documentação detalhada. Apesar desses desafios, os objetivos propostos foram alcançados, e uma parte de uma aplicação *web* foi desenvolvida com sucesso usando a abordagem proposta.

Palavras-chave: Gerenciamento de testes de software, Processo de gerenciamento de testes de software, Testes de software, Metodologia ágil, PXP, Personal Extreme Programming, Kungfu Testing

Abstract

Web applications, essential in various daily activities, are becoming increasingly complex. This makes it crucial to ensure their quality through robust testing and a wide coverage of their functionalities. Software testing management, a process that aims to ensure the quality of tests and software, contributes to the development of software within the stipulated deadline. This work aims to set up an agile software development methodology, incorporating activities and artifacts of software testing management. To enhance the assurance and quality of software products, Personal Extreme Programming (PXP) will be set up to integrate the Kungfu Testing test management approach, designed to harmonize with agile software development approaches. The application of the configured development process is exemplified by the development of part of a web application, carrying out its activities and making its artifacts. The results of this work demonstrate that the integration of testing management in PXP can improve the practice of software testing and contribute to the delivery of a more reliable product. However, this integration also presents challenges, such as the increase in development activities and the need for detailed documentation. Despite these challenges, the proposed objectives were achieved, and a part of a web application was successfully developed using the proposed approach.

Keywords: Software test management, Software test management process, Software testing, Agile methodology, PXP, Personal Extreme Programming, Kungfu Testing

Sumário

Lista de Figuras

Lista de Tabelas

Capítulo 1

Introdução

Atualmente, com o avanço constante da tecnologia, o desenvolvimento de *software* tem se tornado um processo cada vez mais importante e presente em muitas empresas. Para atender as demandas do mercado de forma ágil e eficiente, muitas organizações adotam os métodos de desenvolvimento ágil.

Com isso, a metodologia de desenvolvimento ágil é uma abordagem que vem sendo amplamente utilizada para o desenvolvimento de *software* que enfatiza a entrega rápida e contínua de valor ao cliente por meio de ciclos curtos e iterativos, adaptando-se às mudanças e às necessidades do cliente. Procura melhorar a qualidade do *software* e responder mais facilmente às mudanças [1].

No entanto, uma limitação identificada na metodologia ágil é a abordagem limitada à documentação. Segundo estudos anteriores [2], a metodologia ágil prioriza a entrega contínua de *software* funcional, muitas vezes relegando a documentação a um segundo plano. A documentação, no entanto, desempenha um papel crucial no desenvolvimento de *software*, facilitando diagnósticos externos e apoiando a manutenção e o uso a longo prazo [2].

Além disso, a metodologia Scrum, um dos métodos ágeis mais populares, apresenta um desafio específico. Como o Scrum libera produtos como incrementos, cada incremento deve ser testado. No entanto, o teste unitário, comumente usado para testar cada incremento, tem uma eficácia limitada na detecção de *bugs*, variando de 25% a 30% [3]. Em contraste, o teste de sistema tem uma eficácia muito maior na detecção de *bugs*, chegando a 85% [3]. Isso indica um problema potencial relacionado à qualidade do software no uso da metodologia ágil.

Apesar dos benefícios da metodologia ágil, as pesquisas indicam que ela não tem sido totalmente bem-sucedida. Em 2012, 46% dos pequenos projetos que utilizaram métodos ágeis tiveram sucesso, 6% falharam e 48% foram contestados [1]. Isso sugere que ainda há espaço para melhorias no processo.

Uma possibilidade para buscar o aumento na qualidade do *software* é a adição do processo de gerenciamento de teste de *software*. O processo de gerenciamento de teste é um conjunto de tarefas relativas à preparação dos planos para execução, monitoramento da execução, análise de anomalias, relatório do progresso dos processos de teste, avaliação dos resultados do teste, determinação se uma tarefa de teste está concluída e verificação dos resultados do teste quanto à integridade [4].

1.1 Problema

Métodos ágeis usados pelas organizações para desenvolver os seus produtos de *software*, muitas vezes não definem atividades e artefatos relevantes em processo de gerenciamento de *software* [5]. Isso pode levar a má elaboração de testes, além de problemas e falhas no produto final.

O processo de gerenciamento de testes pode promover a qualidade do produto final e reduzir os riscos de falhas e problemas [4].

Portanto, é relevante reconhecer a importância do processo de gerenciamento de testes e buscar maneiras de integrar o processo de gerenciamento de testes de *software* a métodos ágeis.

1.2 Objetivo geral

Configuração de metodologia ágil de processo de desenvolvimento de *software* para a incorporação de atividades e artefatos de processo de gerenciamento de testes de *software*.

1.3 Objetivos específicos

Os objetivos específicos do projeto são:

- Descrever conceitos sobre *software*, modelos de desenvolvimento de software e processo de desenvolvimento de *software*;
- Descrever conceitos sobre qualidade de *software*, garantia da qualidade de *software*, controle da qualidade de *software*, teste de *software*, processo de teste de *software* e processo de gerenciamento de teste de *software*;
- Configurar o *Personal Extreme Programming* para incorporar atividades e artefatos de gerenciamento de testes;

- Desenvolver uma aplicação *web* adotando *framework* configurado com atividades e artefato de gerenciamento de testes.

1.4 Justificativa

O gerenciamento de testes de *software* é um processo realizado em etapas que devem ser distribuídas durante todo o ciclo de desenvolvimento do *software*. Estas etapas podem ser realizadas de maneira cíclica no qual os relatórios finais do processo podem alimentar e contribuir com o planejamento dos testes para que sejam definidas melhores estratégias de aplicações e cobertura dos testes [6].

A aplicação *web* é uma classe de *software* frequentemente utilizada no dia a dia das pessoas, seja para trabalho, entretenimento ou outras atividades, agregando, assim, diversas funcionalidades que, em alguns casos, são de grande necessidade para os usuários. Portanto, a qualidade da aplicação *web* é relevante para seu sucesso. Esses produtos de *softwares* estão cada vez mais complexos e, por isso, necessitam cada vez mais de um rigoroso controle de qualidade. Para isso, os testes de aplicações *web* carecem de uma boa cobertura sobre as funcionalidades da aplicação e de robustos testes para a avaliação de seu perfeito funcionamento [7].

A integração de processo de gerenciamento de testes de *software* com metodologia ágil aplicada ao desenvolvimento das aplicações *web* pode promover o controle de qualidade em processo de desenvolvimento de aplicação *web* onde metodologia ágil seja adotada.

1.5 Estrutura do projeto

Este projeto é dividido em três partes: a fundamentação teórica, a abordagem prática e a conclusão do trabalho.

A fundamentação teórica é composta por dois capítulos, baseando-se em diversas fontes de informações. O capítulo 2 apresenta as definições e características sobre *software*, modelos de desenvolvimento de *software* e processo de desenvolvimento de *software*. No capítulo 3 são apresentadas as definições e características acerca de qualidade de *software*, garantia da qualidade de *software*, controle da qualidade de *software*, teste de *software*, processo de teste de *software* e processo de gerenciamento de teste de *software*.

A prática do trabalho é descrita em dois capítulos. No capítulo 4 é descrito a configuração para a integração do processo de gerenciamento de testes de *software* ao método ágil *Personal Extreme Programming* (PXP). No capítulo 5 são descritos os detalhes da utilização do PXP integrado com o gerenciamento de testes no desenvolvimento de uma pequena aplicação *web* para demonstração da aplicação do *framework*.

No capítulo 6 é apresentada análise dos resultados da aplicação do processo ao utilizar o PXP com sua nova configuração e conclusões obtidas com a aplicação do modelo na prática de desenvolvimento de uma aplicação *web*.

Parte I

Referencial Teórico

Capítulo 2

Processo de Desenvolvimento de *Software*

Neste capítulo, inicialmente será apresentada a definição e as categorias de *software*, detalhando suas características e classificações. Em seguida, serão descritos os modelos de desenvolvimento de *software*, explorando suas metodologias e abordagens distintas. Posteriormente, será definido o processo de desenvolvimento de *software*, juntamente com suas atividades associadas, destacando a importância de cada etapa no ciclo de vida do *software*.

2.1 *Software*

O termo *software* é amplamente conhecido e utilizado nos dias de hoje, mas muitas vezes seu significado pode não ser completamente compreendido. De acordo com a ISO/IEC/IEEE 24765:2017 [8], o *software* refere-se a todos os elementos de programas de computador, procedimentos e documentação associada que estão relacionados com a operação de um sistema de computador. Nesse contexto, podemos entender o *software* como um conjunto de instruções e dados que permitem que um computador realize tarefas específicas.

Além disso, de acordo com a definição apresentada pela IEEE 828-2012 [9], o *software* pode abranger tanto os programas e procedimentos quanto as regras e a documentação associada de um sistema de processamento de informações. Essa perspectiva ressalta que o *software* não se limita apenas aos programas executáveis, mas também inclui elementos fundamentais, como as regras que governam o funcionamento do sistema e a documentação que descreve o uso e a operação do *software*.

O *software* é um componente lógico, ou seja, não possui uma forma física tangível como o hardware. Em contraste com os componentes físicos de um computador, o *software* é

desenvolvido ou projetado, não fabricado no sentido tradicional [10]. Isso significa que sua criação envolve um processo de engenharia que requer habilidades de design e programação para garantir que o *software* funcione conforme o esperado.

Uma característica fundamental do *software* é que ele não "se desgasta" como o hardware. De acordo com Pressman [10], o *software* não está sujeito a problemas causados pelo ambiente físico, como poeira, vibração ou temperaturas extremas, que podem afetar o hardware ao longo do tempo. Em teoria, o *software* deveria apresentar uma curva de taxa de falha idealizada, na qual os defeitos iniciais seriam corrigidos e a taxa de falha se estabilizaria em um nível baixo, sem aumento gradual ao longo do tempo. No entanto, essa curva idealizada é simplificada e na prática, o *software* pode se deteriorar devido às mudanças e atualizações que são realizadas ao longo do seu ciclo de vida [10].

Outro ponto importante é que o *software* é altamente customizável, permitindo que seja projetado para atender a necessidades específicas. Embora existam esforços na indústria de tecnologia para criar componentes de *software* reutilizáveis em larga escala, ainda predominam os *software* feitos sob medida para atender aos requisitos de cada sistema e aplicação [10].

2.2 Categorias de *software*

Existem sete amplas categorias de *software* que apresentam desafios contínuos para os engenheiros de *software*. Cada uma dessas categorias desempenha um papel fundamental na evolução da tecnologia e atende a diversas necessidades do mundo moderno [10].

- *Software* de sistema é a base sobre a qual toda a computação é construída. Ele consiste em programas que servem a outros programas, processando informações complexas e determinadas, ou dados indeterminados. Compiladores, editores e utilitários de gerenciamento de arquivos são exemplos de *software* de sistema que interagem intensamente com o *hardware* do computador e são essenciais para a execução de outras aplicações.
- *Software* de aplicação engloba programas autônomos projetados para resolver necessidades específicas de negócios. Essas aplicações processam dados comerciais ou técnicos, facilitando operações comerciais e tomadas de decisões gerenciais e técnicas. Além do processamento de dados convencional, o *software* de aplicação permite controlar funções em tempo real, como o processamento de transações de ponto de venda ou o controle de processos de fabricação em tempo real.
- *Software* de engenharia/científico se destaca por seus algoritmos de "números pesados". Aplicações nessa área são utilizadas em diversos campos, desde astronomia

e vulcanologia até análise de estresse em automóveis e dinâmica orbital de ônibus espaciais. No entanto, essas aplicações têm evoluído para além dos algoritmos numéricos tradicionais e estão abraçando tecnologias como o projeto auxiliado por computador e a simulação de sistemas em tempo real.

- *Software* embarcado é encontrado em produtos e sistemas, e sua função é implementar e controlar recursos e funções para o usuário final e para o próprio sistema. Essas aplicações podem variar de funções esotéricas, como o controle do teclado de um forno de micro-ondas, até funções mais complexas e críticas, como o controle de combustível, *displays* do painel e sistemas de freios em automóveis.
- *Software* de linha de produtos é projetado para fornecer capacidades específicas para uso por muitos clientes diferentes. Ele pode se concentrar em mercados especializados, como controle de inventário, ou atender a mercados de consumo em massa, oferecendo aplicativos como processadores de texto, planilhas, gráficos de computador, multimídia, entre outros.
- Aplicações *web*, também conhecidas como "*WebApps*", compreendem uma ampla variedade de *software* centrado em rede. Desde conjuntos simples de arquivos de hipertexto até ambientes de computação sofisticados, as *WebApps* oferecem recursos autônomos, funções de computação e conteúdo para o usuário final, além de estarem integradas a bancos de dados corporativos e aplicações comerciais.
- *Software* de inteligência artificial se utiliza de algoritmos não numéricos para resolver problemas complexos que não podem ser resolvidos facilmente através de cálculos convencionais. Aplicações nessa área incluem robótica, sistemas especialistas, reconhecimento de padrões, redes neurais artificiais e jogos.

2.3 Modelos de desenvolvimento de *software*

Existem modelos que podem ser usados com o propósito de representar o processo de desenvolvimento de *software*. Assim como existem diversos processos, há vários modelos para representá-los. Entre os modelos existentes, temos os seguintes: *modelo em cascata*, *desenvolvimento Incremental* e *modelo de desenvolvimento ágil* [11].

2.3.1 Modelo em cascata

O modelo em cascata é um antigo modelo de processo. As atividades são distribuídas em fases distintas seguindo uma ordem estrita, em cascata [11].

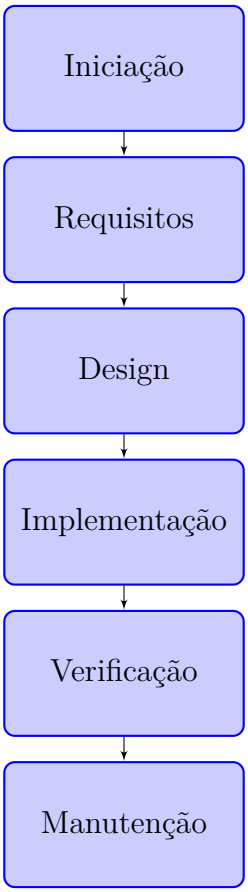


Figura 2.1: Modelo em cascata

As principais fases são as seguintes:

- Análise e definição de requisitos: nessa fase é efetuada a análise e captação das necessidades do cliente e definidos os requisitos do *software*;
- Projeto (*design*) de *software*: fase em que é definida a arquitetura do *software* de acordo com os requisitos da etapa anterior. Essa fase envolve a identificação e descrição das abstrações fundamentais do *software* e seus relacionamentos;
- Implementação e teste unitário: nessa fase é desenvolvida cada unidade do programa e testes para estas unidades com a finalidade de verificar a adequação com sua especificação;
- Integração e teste de sistema: fase em que é feita a junção das unidades do programa e o *software* integrado é testado, avaliando a sua conformidade com os requisitos definidos na primeira fase;
- Operação e manutenção: nessa fase o *software* é colocado em operação e com sua utilização podem ser identificadas falhas ou novos requisitos que resultem em manutenção.

2.3.2 Desenvolvimento incremental

O modelo de desenvolvimento incremental tem como princípio desenvolver o *software* de maneira iterativa. Nela, uma versão inicial do *software* é desenvolvida e disponibilizada para a utilização do usuário. Havendo a necessidade de modificações, inicia-se outra iteração, desenvolvendo uma nova versão do produto e seguindo os mesmos passos da anterior até que atenda às necessidades do cliente [11].

O desenvolvimento incremental vem se tornando frequente no desenvolvimento de *software*. Esse modelo é adotado, por exemplo, em algumas metodologias ágeis de desenvolvimento de *software* (*agile software development*) [11].

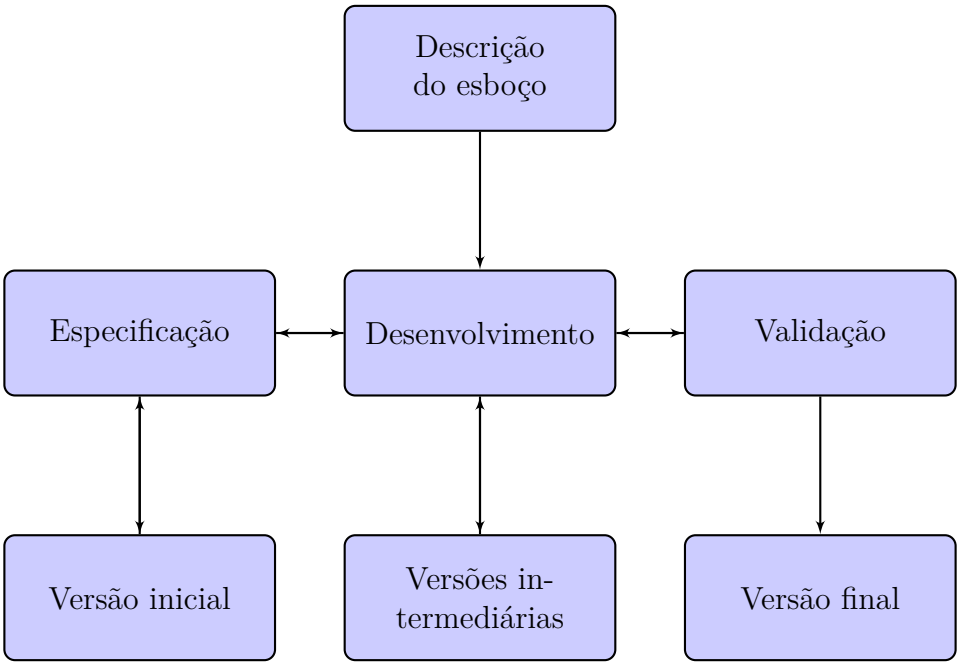


Figura 2.2: Desenvolvimento incremental

2.3.3 Modelo de desenvolvimento ágil

Na década de 1980 e início da de 1990, a abordagem dominante no desenvolvimento de software era baseada em um planejamento cuidadoso do projeto, análise e projeto formais, além de um processo de desenvolvimento rigoroso e controlado [11]. Essa abordagem pesada, direcionada por planos, era adequada para o desenvolvimento de sistemas aeroespaciais e governamentais de grande escala, que envolviam equipes grandes e dispersas geograficamente e prazos de longa duração [11].

No entanto, quando aplicada a sistemas corporativos de pequeno e médio porte, essa abordagem mostrou-se excessivamente burocrática, com um grande *overhead* no planejamento e documentação, em detrimento do desenvolvimento efetivo [10]. O retrabalho era

frequente à medida que os requisitos do sistema mudavam, levando a mais tempo gasto em análises do que na criação e teste dos programas [11].

Essa insatisfação com os métodos tradicionais impulsionou uma nova onda de pensamento na década de 1990, resultando no surgimento dos "métodos ágeis" [11]. Esses métodos buscavam permitir que as equipes de desenvolvimento se concentrassem mais no *software* em si e menos na concepção e documentação detalhadas [11]. Uma das características fundamentais dos métodos ágeis é sua abordagem incremental, em que o *software* é desenvolvido e entregue em pequenas iterações, permitindo que os requisitos mudem ao longo do processo [12].

O manifesto ágil reflete a filosofia subjacente a esses métodos, valorizando mais os indivíduos e interações do que processos e ferramentas, o *software* em funcionamento do que documentação abrangente, a colaboração do cliente do que a negociação de contrato e a resposta a mudanças do que a adesão rígida a um plano [12]. Essa abordagem flexível e adaptável é particularmente adequada para o desenvolvimento de aplicativos em que os requisitos do sistema mudam rapidamente durante o processo de desenvolvimento [11].

Apesar das diferentes abordagens dos métodos ágeis, todos eles compartilham os princípios fundamentais estabelecidos pelo manifesto ágil [11]. A ênfase em desenvolvimento incremental, entrega rápida de valor ao cliente e adaptabilidade aos requisitos em constante mudança tornou o modelo de desenvolvimento ágil amplamente adotado na indústria de *software* [12].

2.3.4 Exemplos de métodos ágeis

Ao longo dos anos, vários métodos de desenvolvimento ágil foram criados para atender às necessidades específicas de diferentes projetos. Entre os métodos ágeis mais populares estão: *Extreme Programming* (XP), *Crystal*, *Scrum*, *Lean Software Development* e *Kanban*.

***Extreme Programming* (XP)**

O método *extreme programming* (XP) é uma abordagem disciplinada que se concentra em velocidade e entrega contínua. Ele promove maior envolvimento do cliente, ciclos rápidos de *feedback*, planejamento e teste contínuos e trabalho em equipe próximo. O *software* é entregue em intervalos frequentes, geralmente de uma a três semanas. O objetivo é melhorar a qualidade do *software* e a capacidade de resposta quando confrontado com mudanças nos requisitos do cliente.

O método XP é baseado nos valores de comunicação, *feedback*, simplicidade e coragem. Os clientes trabalham em estreita colaboração com sua equipe de desenvolvimento para

definir e priorizar suas "*user stories*" solicitadas. No entanto, cabe à equipe entregar as "*user stories*" de maior prioridade na forma de *software* funcional que foi testado em cada iteração. Para maximizar a produtividade, o método XP fornece aos usuários um *framework* leve e de suporte que os orienta e ajuda a garantir o lançamento de *software* empresarial de alta qualidade.

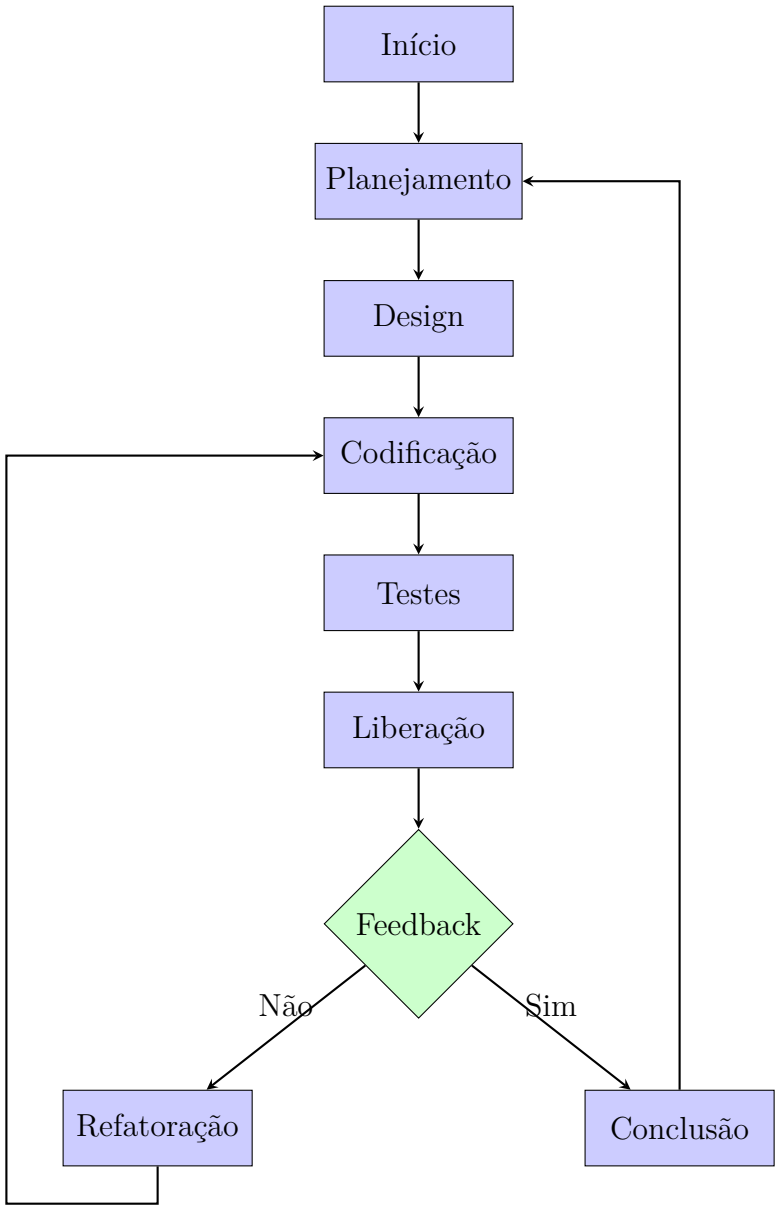


Figura 2.3: *Extreme Programming* (XP)

Crystal

Os métodos *Crystal*, desenvolvidos por Alistair Cockburn, são uma família de metodologias de desenvolvimento de *software* adaptáveis e leves, com foco nas pessoas e interações em projetos ágeis. Eles são codificados por cores para indicar o risco à vida humana. Por

exemplo, projetos que podem envolver risco à vida humana usarão *Crystal Sapphire*, enquanto projetos que não têm tais riscos usarão *Crystal Clear*. O *Crystal* enfatiza seis aspectos principais: pessoas, interação, comunidade, comunicação, habilidades e talentos. O processo é secundário, e há sete propriedades comuns que indicam sucesso, como entrega frequente e comunicação osmótica. Cada projeto requer um conjunto adaptado de políticas e práticas, sendo composto por modelos como *Crystal Orange*, *Crystal Clear* e *Crystal Yellow*. O *Crystal* segue os princípios ágeis, com ênfase na entrega frequente, envolvimento do cliente e simplicidade. A metodologia é flexível, centrada nas pessoas, evitando rigidez e adaptando-se ao ambiente e ao tamanho da equipe.

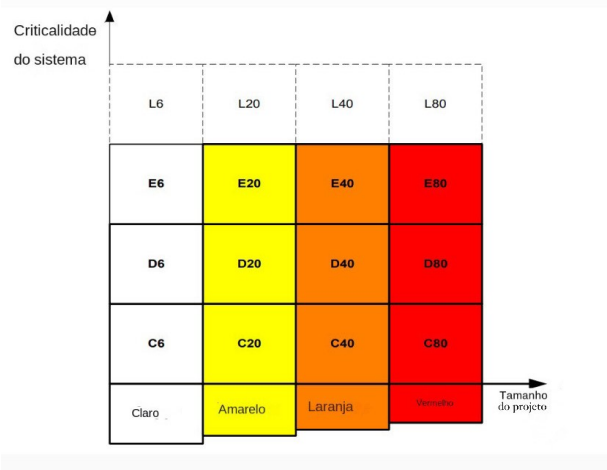


Figura 2.4: *Crystal* (Fonte: Adaptado de [13]).

Scrum

Scrum é um *framework* ágil que gerentes de projetos podem usar para gerenciar projetos iterativos e incrementais. O proprietário do produto trabalha com sua equipe para criar um *backlog* do produto, que é uma lista de tudo o que precisa ser feito para entregar um sistema de *software* bem-sucedido. Isso inclui correções de *bugs*, recursos e requisitos não funcionais. Uma vez definido o *backlog* do produto, apenas a equipe correspondente pode adicionar novas funcionalidades.

Equipes multifuncionais concordam em entregar incrementos de *software* durante cada *sprint*, geralmente em 30 dias. Após cada *sprint*, o *backlog* do produto é reavaliado e repriorizado para selecionar novas funções entregáveis para o próximo *sprint*. *Scrum* é popular por ser simples, produtivo e capaz de incorporar práticas promovidas por outros métodos ágeis.

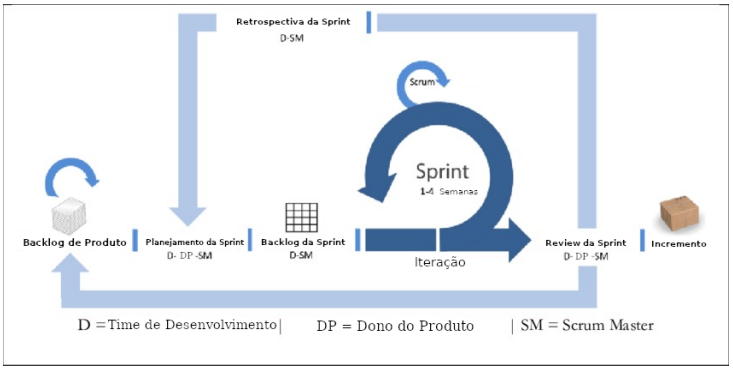


Figura 2.5: *Scrum* (Fonte: Adaptado de [14]).

Lean Software Development

O *framework Lean* é baseado nos princípios da manufatura enxuta. Ele se concentra no uso eficaz do mapeamento do fluxo de valor para garantir que a equipe entregue valor ao cliente. É flexível e evolui constantemente, sem diretrizes ou regras rígidas. Os princípios fundamentais do *Lean* incluem aumentar o aprendizado, capacitar a equipe, promover a integridade, eliminar o desperdício, compreender o todo, tomar decisões o mais tarde possível e entregar o produto o mais rápido possível.

O *Lean* depende de *feedback* rápido e confiável entre clientes e programadores para fornecer fluxos de trabalho de desenvolvimento rápidos e eficientes. Para isso, concede autoridade para tomada de decisões a indivíduos e pequenas equipes, em vez de depender de um fluxo hierárquico de controle. Para eliminar o desperdício, o *Lean* pede aos usuários que selecionem apenas os recursos verdadeiramente valiosos para seu sistema, priorizem esses recursos escolhidos e os entreguem em pequenos lotes. O desenvolvimento de *software Lean* também incentiva a escrita simultânea de testes unitários automatizados com o código e se concentra em garantir que cada membro da equipe seja o mais produtivo possível.

2.4 Processo de desenvolvimento de *software*

Processo de desenvolvimento de *software* é composto por atividades a serem executadas durante o desenvolvimento de *software*. Tais atividades podem ser executadas com o propósito de desenvolver um produto a partir do zero ou, como ocorre com frequência atualmente, desenvolver um produto por meio de extensão ou modificação de produto existente [11].

Entre as atividades que compõem processo de desenvolvimento de *software*, tem-se atividades com os seguintes propósitos:

- Especificação: definição dos requisitos que o *software* a ser desenvolvido deverá atender e quais as restrições quanto ao seu funcionamento;
- Projeto e implementação: definição da organização do *software* e implementação visando atender à especificação;
- Validação: controle realizado com o propósito de garantir que o *software* atende à especificação;
- Evolução: modificação para atender a novos requisitos.

Além de informação acerca de atividades, descrição de processo de desenvolvimento de *software*, pode também conter informação acerca dos seguintes elementos:

- Artefatos: são produtos de atividades do processo;
- Papéis: funções das pessoas envolvidas, direta ou indiretamente, no processo;
- Pré e pós-condições: declarações verdadeiras antes ou depois que as atividades forem executadas ou artefatos produzidos.

Para Sommerville[11]:

> "Os processos de *software* são complexos e, como todos os processos intelectuais e criativos, dependem de pessoas para tomar decisões e fazer julgamentos. Não existe um processo ideal, a maioria das organizações desenvolve os próprios processos de desenvolvimento de *software*. Os processos têm evoluído de maneira a tirarem melhor proveito das capacidades das pessoas em uma organização, bem como das características específicas do sistema em desenvolvimento. Para alguns sistemas, como sistemas críticos, é necessário um processo de desenvolvimento muito bem estruturado; para sistemas de negócios, com requisitos que se alteram rapidamente, provavelmente será mais eficaz um processo menos formal e mais flexível."

2.5 Atividades de processo de desenvolvimento de *software*

O processo de desenvolvimento de *software* envolve uma série de atividades intercaladas, que abrangem desde a especificação inicial até a evolução contínua do *software*. Essas atividades compreendem sequências de tarefas técnicas, colaborativas e de gerenciamento que visam criar um *software* funcional e de alta qualidade [11].

As quatro atividades básicas do processo de desenvolvimento de *software* são: especificação, desenvolvimento, validação e evolução. Essas atividades podem ser organizadas de diferentes maneiras, dependendo do processo de desenvolvimento adotado [11].

A atividade inicial é a especificação de *software*, também conhecida como engenharia de requisitos. Nesse estágio, busca-se compreender os serviços requeridos pelo *software* e identificar restrições relacionadas à sua operação e desenvolvimento [10]. Esses requisitos são apresentados em dois níveis de detalhe: um de alto nível para os usuários finais e clientes, e outro mais detalhado para os desenvolvedores de *software* [11].

Após a especificação de *software* é realizada a atividade de projeto e implementação de *software*. Nesse estágio, a especificação do *software* é convertida em um *software* executável. Os projetistas desenvolvem a estrutura do software, as interfaces entre os componentes e outras considerações, enquanto também refinam a especificação do *software* [11] [10].

Na validação de *software*, os componentes do sistema passam por testes de desenvolvimento, nos quais os próprios desenvolvedores testam os componentes de forma independente. Em seguida, os componentes são integrados para testes de *software*, que buscam identificar interações inesperadas entre os componentes e problemas de interface. Finalmente, o *software* é testado com dados do cliente no estágio de testes de aceitação, para verificar sua conformidade com os requisitos e sua utilidade no mundo real [11].

Por fim, a evolução do *software* trata da flexibilidade dos *softwares* permitindo que sejam continuamente alterados para atender a mudanças nos requisitos e nas necessidades dos clientes. A distinção entre desenvolvimento e manutenção está se tornando menos relevante, já que a engenharia de *software* é vista cada vez mais como um processo evolutivo.

Capítulo 3

Processo de Gerenciamento de Teste de *Software*

Neste capítulo, inicialmente será abordada a definição e importância da qualidade de *software*, bem como a garantia e o controle da qualidade do *software*. Em seguida, será discutida a definição, importância e as classes do teste de *software*. Posteriormente, o foco será na definição, importância e nas fases do processo de teste de *software*. Por fim, será descrita a definição, importância e as etapas do processo de gerenciamento de teste de *software*.

3.1 Qualidade de *software*

Como em qualquer ramo da engenharia, a qualidade do produto final a ser entregue para o usuário, ou cliente, é de fundamental importância e na engenharia de *software* não poderia ser diferente. Com o aumento da demanda por sistemas complexos e de grande responsabilidade nas organizações, a qualidade se torna um elemento fundamental no desenvolvimento de *software*. Devido a isso, a qualidade de *software* vem ganhando mais espaço e relevância dentro do desenvolvimento de aplicações ou de sistemas computacionais, de forma a buscar melhores metodologias ou o desenvolvimento de sistemas para tal avaliação [15].

Na década de 1960, com o desenvolvimento de produtos de *software* mais complexos, surgiram problemas com a qualidade do *software* que continuaram a afetar a engenharia de *software* ao longo do século XX. O *software* entregue era lento, pouco confiável e difícil de manter. Para resolver isso, foram adotadas técnicas formais de gerenciamento de qualidade baseadas em métodos da indústria manufatureira [11].

Segundo Ravichandran [16], a definição de qualidade de *software* é "o grau em que um sistema, componente ou processo atende aos requisitos especificados", ou seja, ela

se caracteriza como a adequação do *software* às necessidades e expectativas de quem o requer. Portanto, esse será considerado de boa qualidade quando está em conformidade com sua especificação e propósito pretendido.

No desenvolvimento de *software*, qualidade é um conjunto de características que atendem às necessidades dos usuários. A qualidade do produto não é espontânea e depende da qualidade do processo de desenvolvimento [15].

3.2 Importância da qualidade de *software*

Como visto anteriormente, desde o final dos anos 60 a qualidade de *software* vem ganhando cada vez mais importância. Com a computação e a tecnologia mais interligada à vida das pessoas, realizando de funções simples e pouco relevantes, como um aplicativo de bloco de notas, até atividades vitais como sistemas médicos. Com isso, aumenta a exigência para que eles funcionem adequadamente, como por exemplo, ao acessar um aplicativo de banco em que o cliente espera sempre que a função desejada não apresente falhas inesperadas e que a sua operação finalize de forma rápida e segura.

Conforme abordado na definição, qualidade de *software* está associada à expectativa e experiência do cliente final. Por consequência, a importância dela pode ser observada pelo fato de que todo *software* ou sistema computacional visa atender ao máximo os requisitos do cliente, ou seja, quanto maior sua qualidade, mais ele atinge seu objetivo [11].

Portanto, investir na qualidade de *software* de um projeto poderá determinar se ele terá sucesso ou não.

> "Embora se possa pensar que um excelente controle de qualidade de *software* é caro, este acaba gerando um retorno muito positivo sobre o investimento. Quando projetos de *software* cancelados e desastrosos são estudados por meio de "autópsias", todos eles têm padrões semelhantes: as fases iniciais de projetos problemáticos são tratadas descuidadosamente, sem análise adequada de requisitos ou revisões de projeto. Após passar pelas fases iniciais e parecer estar adiantado, os problemas começam a se acumular durante a codificação e o teste. Quando os testes começam para valer, problemas sérios são detectados de modo que cronogramas e metas de custo não podem ser alcançados. De fato, alguns projetos de *software* têm tantos problemas sérios - denominados bugs ou defeitos - que são cancelados sem conclusão." [17].

Logo, os projetos que dão importância à qualidade do *software* e seguem suas boas práticas costumam ser bem sucedidos, mesmo isso gerando aumento em seu tempo e custos iniciais, pois acabam por criar um produto que levará a menos problemas e retrabalhos para correções.

3.3 Garantia da qualidade de *software*

Tanto na indústria tradicional quanto na de *software*, para se certificar da boa qualidade do produto é necessário definir ou seguir recomendações de boas práticas e padrões durante a produção, a fim de garantir que este satisfaça o que dele é esperado [11].

Segundo o IEEE [18], garantia da qualidade de *software* é "um padrão planejado e sistemático de todas as ações necessárias para fornecer confiança adequada de que um item ou produto está em conformidade com os requisitos técnicos estabelecidos.".

Portanto, é na garantia da qualidade de *software* que se determina os procedimentos, processos, padrões e, em alguns casos, também o gerenciamento de configuração, atividades de verificação e validação da qualidade do produto final. Por exemplo, é na garantia da qualidade de *software* que devem ser escolhidas as ferramentas e métodos que possibilitarão o uso dos padrões que foram definidos.

Tais definições são responsáveis por orientar como a qualidade de *software* pode ser alcançada e servir de parâmetro para saber se o sistema ou programa desenvolvido atingiu o nível de qualidade requerido.

A garantia da qualidade de *software* engloba todo o processo de desenvolvimento na qual se faz o monitoramento e melhorias dos processos [16]. Assim, garantindo que os padrões e procedimentos acordados estão sendo seguidos e que problemas sejam encontrados para imediatas correções. Dessa forma, evitando que sua descoberta ocorra apenas ao final de todo o desenvolvimento.

3.4 Importância da garantia da qualidade de *software*

A garantia da qualidade de *software* é muito relevante para que um produto possa atingir um bom nível de qualidade e atender aos requisitos que lhe foi atribuído. Negligenciar ou ignorar sua aplicação no desenvolvimento de um *software* é uma das principais causas de falhas em projetos [16].

Como visto no item anterior, ela tem uma atuação importante no ciclo de vida do *software* ao estar envolvida desde a etapa de definição de procedimentos e padrões no início do projeto até a parte de validação da qualidade do produto final. Com isso, sua adequada aplicação pode aumentar consideravelmente o sucesso do projeto e reduzir riscos de falhas ou defeitos no produto final.

Sua importância também se deve ao fato de que a garantia da *software* é responsável pelo monitoramento dos processos em todo o desenvolvimento, logo, ao garantir que os

padrões e procedimentos estão sendo seguidos corretamente, ajuda a prevenir que falhas de produção ocorram, tornando esta mais eficiente [11].

3.5 Controle da qualidade de *software*

Assim como o termo garantia de qualidade, controle de qualidade é amplamente usado na indústria manufatureira. Nesta, o termo descreve a aplicação de processos os quais foram definidos na etapa da garantia de qualidade que envolvem uma série de inspeções, revisões e testes com o propósito de assegurar que todos os procedimentos e padrões sejam seguidos.

Controle de qualidade de *software* pode ser:

> "O conjunto de procedimentos usados pelas organizações para garantir que um produto de *software* atenda às suas metas de qualidade com o melhor valor para o cliente e para melhorar continuamente a capacidade da organização de produzir produtos de *software* de qualidade no futuro." [19].

O controle de qualidade de *software* é o processo responsável por rastrear, identificar, rejeitar, reduzir e corrigir possíveis defeitos e maus funcionamentos em seu projeto [20]. Em ciclo de vida de *software*, teste e revisão são processos relevantes em controle de qualidade, processos que têm por objetivo alcançar conformidade com especificações técnicas e demandas.

3.6 Importância do controle da qualidade de *software*

Com a crescente importância e complexidade dos sistemas computacionais no cotidiano da sociedade, percebe-se as várias razões pelas quais é importante o controle da qualidade de *software* para que estes tenham funcionamento adequado e cumpram seus papéis, conforme o que se espera deles.

Uma dessas razões é a sensibilidade da natureza das atividades de produtos de *softwares* aplicados em áreas críticas como: bancária, de controle aéreo ou médica. Caso esses apresentem falhas podem causar perdas de grandes quantias, graves acidentes ou perdas de vidas [19]. Logo, é de fundamental importância que estes passem por rigorosos e minuciosos controles de qualidade a fim de certificarem que atendem adequadamente aos seus requisitos e confiabilidade adequada.

Outra razão para tal importância, dá-se ao fato de que ao final do desenvolvimento de um produto de *software* é comum que seus clientes façam avaliações com critérios

pré-definidos para verificar se estes cumprem com seus requisitos. Pode ser custoso em tempo, dinheiro e esforço, remover defeitos introduzidos nas etapas do desenvolvimento, e que não foram devidamente identificados em cada etapa, em razão da não aplicação ou deficiência no controle de qualidade [19].

A organização de desenvolvimento de *software* que dá a devida importância ao controle de qualidade e preza pela boa aplicação de todos os processos dessa, pode, como visto na definição, aprender com cada produção de *software* a aperfeiçoar-se e a tornar-se mais eficiente. Assim, conseguirá lidar melhor com os problemas que ocorrerem durante o desenvolvimento e, por consequência, torná-lo mais eficiente. Além disso, terá uma melhor organização ao iniciar um novo projeto, como por exemplo, saber coletar melhores informações e requisitos dos seus clientes.

Logo, é importante a aplicação de um bom controle de qualidade de *software* para que se tenha uma maior eficiência no desenvolvimento, maior qualidade no produto gerado e, também, menores gastos com sua produção e manutenção.

3.7 Teste de *software*

Em ciclo de vida de *software*, os testes são responsáveis por demonstrar e verificar que o programa realiza suas funções da forma esperada ou, ainda, descobrir defeitos e inconsistências.

A definição do termo "teste de *software*" segundo o IEEE [18] é "uma atividade na qual um sistema ou componente é executado sob condições especificadas, os resultados são observados ou registrados e é feita uma avaliação de algum aspecto do sistema ou componente".

Portanto, o teste serve como um filtro para cada etapa na qual ele é aplicado. Se o que foi desenvolvido for aprovado, segue para a próxima fase do desenvolvimento. Porém, caso seja reprovado, será feita uma avaliação para averiguar quais foram as causas para ser feita a correção dos problemas.

O teste de *software* tem dois principais objetivos. O primeiro é demonstrar que o produto cumpre com os requisitos propostos, em que se busca criar testes para cada requisito ou característica que ele possui. Já o segundo serve para expor comportamentos anômalos ou falhas, na qual são realizados os casos de testes para demonstrar os defeitos do *software*, não necessariamente refletindo seu uso comum [21].

3.8 Importância de teste de *software*

Por servir como um filtro no desenvolvimento de *software*, buscando expor falhas, o teste é um importante processo no ciclo de vida de *software*.

Teste é importante no processo de controle de qualidade do *software*. Por meio dele é possível avaliar se um *software* está de acordo com os requisitos propostos pelas partes interessadas (*stakeholders*) no mesmo [11].

A aplicação de testes é também benéfica e, por consequência, muito importante por possibilitar a revisão da especificação, do projeto (*design*) e da codificação, assim verificando se atende o que o cliente espera dele [11].

Segundo Karabašević *et al.* [22], "É importante reconhecer a importância do teste de *software* como uma fase básica no ciclo de desenvolvimento de *software*. O teste ajuda a reduzir o risco de falha do produto e garante que o produto atenda aos requisitos comerciais e técnicos.".

3.9 Classes de teste de *software*

Com o aumento da complexidade e das funcionalidades dos produtos de *softwares*, os testes de *software* precisaram evoluir em escala para acompanhar essas mudanças.

Para melhor verificar e validar os diferentes aspectos de um *software*, tendo como foco a especialidade do que está sendo testado, tipicamente são necessários níveis de testes de *software*. Assim, é possível indicar falhas com mais precisão de onde elas ocorrem no *software* facilitando sua correção.

Segundo o IEEE Computer Society [21], "os níveis podem ser distinguidos com base no objeto do teste, que é chamado de alvo, ou no propósito, que é chamado de objetivo (do nível de teste)".

Alvo de teste

Existem diferentes alvos de teste, por exemplo, módulo, conjunto de módulos ou sistema. Devido a isto, este nível se subdivide em três classes de teste:

- Teste unitário: responsável por testar isoladamente um módulo.
- Teste de componentes: responsável por testar um conjunto de módulos que formam um componente do sistema, focando nas interfaces.
- Teste de sistema: responsável por testar o funcionamento do sistema como um todo, focando na integração entre os componentes.

Objetivo de teste

Existem diferentes objetivos de testes, que são divididos em duas classes:

- Teste funcional: responsável por testar se as funcionalidades do sistema correspondem com o que foi especificado.
- Teste não funcional: responsável por testar características que não consistem nas funcionalidades do *software*, como seu desempenho, sua capacidade, segurança, entre outros.

3.10 Processo de teste de *software*

Segundo IEEE [23], o processo de teste de *software* "é o processo de análise de um item de *software* para detectar as diferenças entre as condições existentes e necessárias (isto é, defeitos/erros/*bugs*) e para avaliar os recursos do item de *software*.".

Processo de teste de *software* é composto por atividades que têm o propósito de validar e verificar produto de *software*, e garantir que produto de *software* atende a requisitos propostos. Processo de teste de *software* é composto por atividades de planejamento, execução e registro de teste. A seguir, são relacionados objetivos de processo de teste de *software*:

- Demonstrar que o produto cumpre com os requisitos, em que se busca criar testes para cada requisito ou característica que ele possui;
- Revelar comportamentos inconsistentes em que são realizados casos de testes para indicar eventuais defeitos, de tal forma que sejam identificados e eliminados do *software*.

3.11 Importância do processo de teste de *software*

Por meio de processo de teste de *software* procura-se garantir que o produto de *software* atende aos requisitos.

Através de processo de teste, procura-se descobrir falhas e auxiliar na garantia da qualidade do *software*, tendo em vista que as atividades do processo de teste estarão presentes em várias etapas do processo de desenvolvimento do *software*. Isso ajuda a prevenir eventuais retrabalhos gerados por erros ou por discordância com os requisitos, pois permite controlar a qualidade ao longo do desenvolvimento. Segundo Ahamed [24], "um *software* exaustivamente testado mantém a qualidade.".

3.12 Fases do processo de teste de *software*

Dado que as atividades de processo de teste são executadas ao longo do processo de desenvolvimento de *software*.

- Teste unitário: nessa fase são realizados testes para as menores partes do *software* isoladamente, assegurando a correção dessas. Os testes unitários podem auxiliar na detecção de erros de lógica ou de implementação, assim garantindo o bom funcionamento dessas partes;
- Teste de Integração: nessa fase os testes verificam o funcionamento das unidades do *software* de maneira combinada. Eles podem avaliar o funcionamento das interfaces e como os dados são tratados. Esses podem ser realizados sequencialmente, incluindo cada unidade do *software*, aplicando assim todos os casos de testes;
- Teste de Sistema: essa fase ocorre após a integração do *software*, e tem por objetivo avaliar a conformidade com os requisitos do *software*, avaliando tanto os funcionais, quanto os não-funcionais, buscando simular as condições em que o *software* será aplicado.
- Testes de Aceitação: essa fase é uma extensão dos testes de sistema, na qual os usuários realizam os testes verificando se o *software* está pronto e adequado aos requisitos e necessidades, no que se refere a requisitos funcionais e não funcionais.

3.13 Processo de gerenciamento de teste de *software*

Processo de gerenciamento de teste de *software* é responsável pela gerência de atividades de teste de *software*. Parveen et al. [25] defini o processo de gerenciamento de teste como "um método de organização de ativos e artefatos de teste, como requisitos de teste, casos de teste e resultados de teste para permitir acessibilidade e reutilização".

Seu objetivo é garantir a qualidade tanto dos testes que serão realizados, quanto do *software*, auxiliando no desenvolvimento da aplicação nos prazos estabelecidos.

O processo de gerenciamento de teste é descrito pelo padrão IEEE 829-2008 [26] como um conjunto de tarefas que incluem a preparação dos planos de execução, o monitoramento da execução, a análise de anomalias, o relatório do progresso dos processos de teste, a avaliação dos resultados do teste, a determinação se uma tarefa de teste está concluída e a verificação dos resultados do teste para ver se estão completos.

3.14 Importância do processo de gerenciamento de teste de *software*

Na medida em que aumenta a complexidade dos produtos de *software*, tende a aumentar a importância do processo de teste na garantia da qualidade.

Devido à quantidade e variedade de testes, é importante o gerenciamento de teste, pois como pode ser observado em sua definição, a utilização do processo de gerenciamento de teste promove a correta execução de atividades de teste ao longo do processo de desenvolvimento. Pohjoisvirta [4] afirma que com o gerenciamento de teste é possível identificar problemas no desempenho do teste, assim levando à correção deles e ao aprimoramento do desempenho.

Outra importância desse processo é o fato dele contribuir para uma evolução na qualidade dos testes e no aumento de sua cobertura, pois com os resultados e análises obtidas ao final do processo é possível identificar melhorias a serem feitas no processo de teste, ao fornecer mais dados para a revisão da fase de planejamento dos testes.

Além disso, por planejar e projetar melhor o processo de teste, o processo de gerenciamento de teste de *software* possibilita a utilização mais eficaz dos recursos de teste, pois melhor direciona os esforços de testes.

> "As organizações de *software* visam melhorar a eficiência e a eficácia de suas atividades de teste, pois é uma parte muito cara e importante da garantia de qualidade de um produto de *software*. Uma maneira de fazer isso é implementando suporte de ferramenta dedicado em torno do teste." [4]

3.15 Etapas do processo de gerenciamento de teste de *software*

Segundo o IEEE [6] as atividades de processo de gerenciamento de teste são distribuídas em três processos. São eles: *planejamento de teste*, *monitoramento e controle de teste* e *conclusão de teste*.

- Planejamento de teste: nesse processo é realizado a elaboração do plano de teste. Ele tem por objetivo desenvolver, concordar, registrar e comunicar às partes interessadas relevantes o escopo e a abordagem que será adotada para o teste, permitindo a identificação antecipada de recursos, ambientes e outros requisitos de teste;
- Monitoramento e controle de teste: esse processo examina se o teste progride de acordo com o plano de teste e, também, com as especificações de teste organizacional, caso exista. Se houver desvios significativos do progresso planejado, atividades ou

outros aspectos do plano de teste, as atividades serão iniciadas para corrigir ou compensar as variações resultantes;

- Conclusão de teste: esse último processo é executado quando se obtém uma aceitação de que as atividades de teste estão concluídas. Ele tem por objetivo disponibilizar artefatos de teste úteis para uso posterior, deixar o ambiente de teste em uma condição satisfatória e registrar os resultados do teste.

Parte II

Prática

Capítulo 4

Configuração de processo de desenvolvimento

Neste capítulo, inicialmente serão descritos o *Personal Extreme Programming* (PXP), suas atividades e seu cartão de história de usuário. Em seguida, será descrita a abordagem *Kungfu Testing*, suas atividades, a matriz de risco a ser incluída no plano de teste, e seus artefatos. Por fim, será descrita a configuração do PXP para a incorporação das atividades e artefatos do *Kungfu Testing*.

4.1 *Personal Extreme Programming* (PXP)

O *Personal Extreme Programming* (PXP) é um processo de desenvolvimento de *software* que combina elementos do *Personal Software Process* (PSP) com práticas da *Extreme Programming* (XP). O PXP foi concebido como um processo para desenvolvedores autônomos que trabalham de forma independente, sem fazer parte de uma equipe de desenvolvimento [27].

Desenvolvedores autônomos, também conhecidos como *freelancers*, enfrentam desafios únicos ao conduzir projetos de *software* em um ambiente onde são responsáveis por todas as fases do processo de desenvolvimento [28]. Em situações em que um único desenvolvedor está envolvido no projeto, a escolha do PXP se justifica por várias razões. Primeiramente, o PXP visa simplificar o processo de desenvolvimento para um indivíduo [28]. O PXP estende os princípios do PSP, que já é destinado ao uso individual, com práticas comprovadas da XP, que oferecem abordagens colaborativaspara desenvolvimento de *software* [28].

Uma das dificuldades enfrentadas por desenvolvedores autônomos é a falta de planejamento e controle de qualidade, fatores que podem impactar o cronograma e a qualidade do produto [28]. O PXP oferece uma estrutura para melhor organizar as atividades diá-

rias do desenvolvedor, automatizando tarefas e fornecendo diretrizes para estimativas, planejamento e controle de qualidade [27]. Além disso, incorpora práticas da XP, que promovem a melhoria contínua, feedback constante e atenção à qualidade desde as fases iniciais do desenvolvimento [27].

A escolha do PXP é respaldada pelas dificuldades enfrentadas pelos desenvolvedores autônomos ao adotar metodologias tradicionais ou mesmo o PSP padrão [29]. Essas abordagens podem ser complexas, demandando muito tempo e recursos para serem implementadas, o que pode prejudicar a agilidade e a competitividade dos desenvolvedores autônomos no mercado [28]. O PXP oferece uma alternativa balanceada, permitindo que o desenvolvedor autônomo se beneficie dos princípios do PSP, combinados com práticas da XP que se adequam às necessidades específicas de um único indivíduo trabalhando em um projeto de *software* [27] [29].

4.2 Atividades do PXP

O processo PXP é iterativo e compreende algumas iterações e ciclos em sequência. Composto por sete fases distintas, o PXP apresenta uma estrutura para o desenvolvimento de projetos de *software*. Cada fase contribui para garantir que os requisitos sejam atendidos e que o processo seja constantemente aprimorado [28] [30].

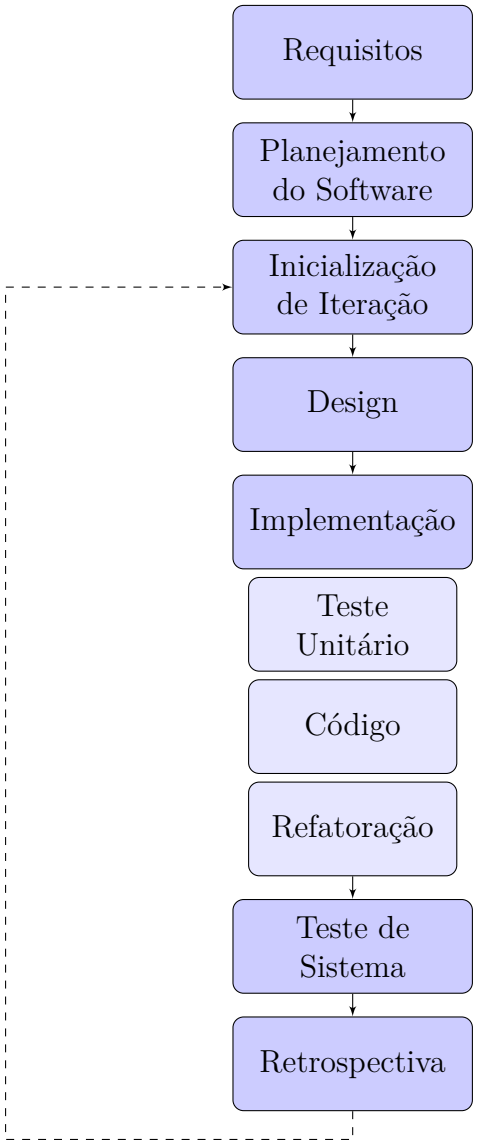


Figura 4.1: *Personal Extreme Programming* (PXP)

4.2.1 Requisitos

A fase de Requisitos é o ponto de partida do PXP. Nessa etapa, o desenvolvedor estabelece os requisitos do sistema que está prestes a ser desenvolvido [30]. Uma metáfora do sistema é criada, permitindo ao desenvolvedor ter uma compreensão clara da funcionalidade pretendida. Além disso, histórias de usuário (*user stories*) são coletadas e registradas em linguagem simples junto com definindo seus níveis de prioridade [28].

4.2.2 Planejamento

A fase de Planejamento é crucial para a organização do desenvolvimento. Antes de planejar as tarefas, durante esta fase, decisões importantes de design são tomadas, como a escolha da linguagem de programação a ser utilizada, a estrutura de desenvolvimento e o modelo de aplicação, entre outras [28] [30]. Estimativas de tempo e custo são atribuídas a cada história de usuário, apoiadas por dados de projetos anteriores [28]. Isso resulta em um plano que delineia como o desenvolvimento será realizado.

4.2.3 Inicialização de Iteração

A Inicialização de Iteração marca o início de uma iteração, um ciclo de desenvolvimento. Durante esta fase, um conjunto específico de recursos é selecionado para ser implementado. A duração da iteração varia, mas geralmente é de uma a três semanas. O objetivo é desenvolver um produto funcional a partir das tarefas escolhidas para a iteração [28].

4.2.4 Projeto (*design*)

A fase de projeto (*design*) é onde o desenvolvedor cria uma estrutura para as tarefas a serem realizadas. O desenvolvedor elabora a estrutura do sistema, abrangendo tanto o banco de dados quanto a interface do utilizador [31]. Um projeto (*design*) mais simplificado resulta em menor período de desenvolvimento e acarreta menores custos para a eventual substituição do código, caso tenha sido investido menos tempo nele [31]. O projeto (*design*) é adaptado para atender aos requisitos do sistema. Ferramentas familiares são usadas para facilitar o processo de projeto (*design*), contribuindo para uma abordagem eficiente [28].

4.2.5 Implementação

A fase de Implementação é onde o projeto (*design*) é transformado em código. O desenvolvedor trabalha no conjunto de recursos, priorizando os recursos de alta prioridade. Tarefas individuais são criadas a partir dos recursos e são realizadas em um processo orientado por testes. O desenvolvimento de testes de unidade é fundamental nesse estágio, garantindo a qualidade do código [28].

4.2.6 Teste do Sistema

A fase de Teste do Sistema é uma fase crucial para verificar se o produto atende aos requisitos. O sistema é testado para identificar defeitos. Os defeitos encontrados são corrigidos e registrados para referência futura [30].

4.2.7 Retrospectiva

A fase de Retrospectiva encerra a iteração e fornece uma oportunidade para análise e melhoria. O desenvolvedor analisa os dados coletados ao longo do processo, comparando as estimativas de tempo com os resultados reais. Propostas de melhoria são consideradas e implementadas conforme necessário. Essa fase pode marcar o lançamento de uma versão do produto ou o início de uma nova iteração [28].

4.3 Cartão de História de Usuário do PXP

Na metodologia PXP, durante a fase de coleta de requisitos, os requisitos do *software* são coletados e representados por meio de histórias de usuários. Essas histórias são escritas em cartões especiais chamados cartões de histórias de usuários. O formato padrão para escrever essas histórias é "Como [papel], eu posso [ação] para que [propósito]". Isso ajuda a garantir que os requisitos sejam claros e compreensíveis para todos os envolvidos no processo de desenvolvimento do *software* [32].

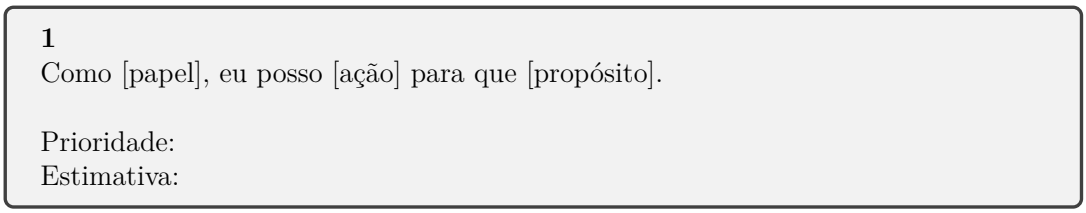


Figura 4.2: Um exemplo de Cartão de História Tradicional

As características das estruturas desses cartões são as seguintes:

- Contém uma breve descrição do comportamento do sistema sob a perspectiva do cliente;
- Utiliza a terminologia simples e não utiliza muita linguagem técnica, ou seja, o cliente usa seus próprios termos para explicar a necessidade;
- Deve conter apenas informações suficiente para que o desenvolvedor possa estimar quanto tempo levará para projetar, testar e implementar uma história específica;

- Deve ser construído um cartão para cada funcionalidade;
- Se estimarem que a história exigirá mais de três semanas de desenvolvimento, o cliente é solicitado a dividir a história em histórias menores e a atribuição de valor e custo ocorre novamente;
- A estimativa para a entrega da funcionalidade é feita no próprio cartão em local específico;
- Orienta a criação do teste de aceitação;
- O cliente define o nível de prioridade das histórias que devem ser implementadas, optando por aquelas que podem ser usadas imediatamente para apoiar os negócios;
- Novas histórias podem ser escritas a qualquer momento.

4.4 Abordagem *Kungfu Testing*

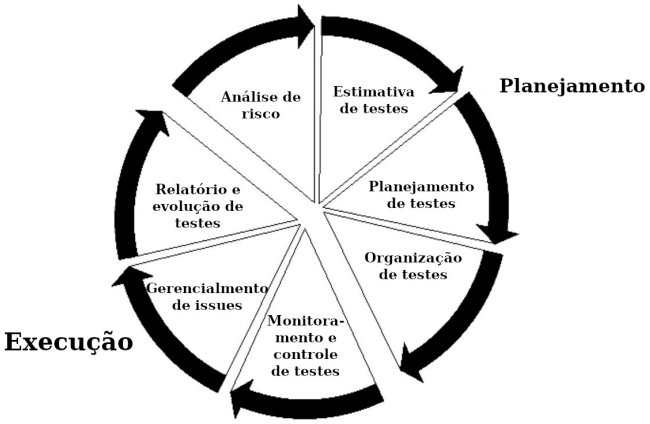


Figura 4.3: *Fases do Kungfu Testing* (Fonte: Adaptado de [33]).

O *Kungfu Testing* é uma abordagem de gerenciamento de teste baseada em conselhos e melhores práticas defendidas por especialistas na área de testes [33]. Ela enfatiza a importância de habilidades através do esforço e treinamento, motivo pelo qual recebeu esse nome, "*Kungfu*", que em chinês significa: a conquista de habilidades adquiridas por meio de trabalho árduo e prática [33]. Ele fornece uma abordagem passo a passo para gerenciar atividades de teste em um ambiente de projeto. Dada a popularidade das metodologias de desenvolvimento ágil, a abordagem apresentada é principalmente destinada a ser usada em ambientes ágeis.

Esta abordagem visa não apenas reduzir riscos associados a defeitos de *software* e falta de qualidade, mas também aprimorar a confiança tanto no *software* desenvolvido quanto nos próprios desenvolvedores. Ao focar em diferentes tipos de testes, como análises de risco, planos de teste e estimativas de testes, o *Kungfu Testing* busca melhorar a qualidade do *software*, aumentando sua estabilidade, segurança e confiabilidade [33].

O *Kungfu Testing* é especialmente adequado para ambientes ágeis, onde a ênfase está na comunicação em detrimento da documentação. Essa abordagem oferece métodos leves para conduzir análises de risco e criar planos de teste, fornecendo um referencial para os testadores que se alinha com a filosofia ágil de desenvolvimento [33].

Além disso, a flexibilidade do *Kungfu Testing* é uma de suas vantagens, pois seus elementos propostos podem evoluir junto com o projeto, adaptando-se às necessidades e requisitos em constante mudança. Isso contribui para uma maior eficiência e eficácia na gestão dos projetos de teste [33].

A escolha da abordagem *Kungfu Testing* para a gestão de testes de *software* é justificada por duas principais razões. Primeiramente, o *Kungfu Testing* é baseado em conselhos e melhores práticas defendidas por especialistas na área de testes, o que garante um certo nível de confiabilidade na abordagem [34]. Além disso, essa abordagem fornece uma estrutura de referência para os testadores, especialmente em metodologias ágeis onde a documentação muitas vezes é escassa [34].

A abordagem de gerenciamento de testes *Kungfu Testing* é dividida em duas principais fases: fase de planejamento e fase de execução.

4.5 Atividades do *Kungfu Testing*

O *Kungfu Testing* é composto por duas fases principais: fase de planejamento e fase de execução.

4.5.1 Fase de planejamento

O principal objetivo é garantir que os recursos de testabilidade sejam incluídos no cronograma e orçamento do projeto desde o início. Isso aumentará a eficiência e eficácia da equipe de teste. Além disso, os interessados devem ser questionados sobre possíveis requisitos não funcionais, como estabilidade e segurança. Esses requisitos são frequentemente negligenciados pelos clientes e gerentes de projeto [33]. Isso ajudará a evitar a falta de tempo e recursos para sua inclusão posterior.

A fase de planejamento é composta pelas seguintes etapas:

- Análise de risco: essa etapa tem como objetivo determinar os objetivos e o escopo dos testes. Para isso, é necessário identificar áreas importantes a serem testadas por meio da análise de risco. Pawlak e Poniszewska-Marańda [33] sugerem o uso do *Product Risk Management* (PRisMA) para auxiliar na elaboração de uma matriz de risco do produto. A matriz contém valores numéricos de risco para cada item identificado. Com base nesses valores, uma abordagem de teste diferente deve ser empregada.

- Estimativa de teste: essa etapa visa estimar o tempo, recursos, custos e habilidades necessários para concluir as atividades de teste. A estimativa pode ser realizada combinando diferentes métodos de estimativa de tarefas, dividindo o teste em tarefas menores e descrevendo cada uma delas em detalhes. Além disso, é importante identificar oportunidades e custos, simulando cenários para um planejamento mais eficaz. O objetivo é garantir uma execução eficiente dos testes dentro das limitações do projeto.

- Planejamento de teste: essa etapa envolve a criação de um documento de plano de teste, que serve como um quadro de referência para as atividades de teste que serão monitoradas. É importante ter um propósito claro em mente ao criar o plano de teste, pois ele pode ser uma ferramenta útil para gerenciar testes ou até mesmo um produto em si. Não há um único modelo adequado para todos os planos de teste. É importante entender cada um dos campos dentro do plano e utilizá-los corretamente, evitando criar documentação detalhada desnecessariamente.

- Organização de teste: essa etapa organiza e define a distribuição das atividades de teste a serem feitas na fase de execução.

4.5.2 Fase de execução

Nessa fase, concentra-se em cumprir o plano de teste e no controle da qualidade do *software* desenvolvido. Os testes são conduzidos em conformidade com a abordagem ágil de desenvolvimento, podendo ser automatizados e executados em paralelo ao desenvolvimento para fornecer *feedback* contínuo. Isso permite correções rápidas e melhoria constante do *software* em desenvolvimento.

A fase de execução é composta pelas seguintes etapas:

- Monitoramento e controle de teste: Essa fase envolve o monitoramento de quatro parâmetros-chave: custo, cronogramas, recursos e qualidade. Esses parâmetros podem ser monitorados por meio da coleta de várias medições e métricas. Custo, cronogramas e recursos podem ser acompanhados verificando regularmente o estado

do orçamento, se o trabalho está sendo entregue no prazo e se os recursos estão disponíveis na quantidade necessária.

- Gerenciamento de problemas: nessa fase, os problemas identificados devem ser registrados e atribuídos prioridade, status de rastreamento e delegação a alguém. Os problemas relatados devem ser monitorados e soluções devem ser buscadas para eles.
- Relatório e Avaliação: essa fase tem como objetivo elaborar relatórios dos testes e avaliá-los. Os relatórios devem ser escritos, numerados, simples, compreensíveis, reproduzíveis e legíveis. A avaliação é baseada na experiência e habilidade da equipe de teste para decidir quando encerrar os testes, garantindo que informações suficientes tenham sido coletadas para uma avaliação precisa da qualidade do produto de *software* final.

4.6 Matriz de risco do *Kungfu Testing*

ID	Descrição do Risco	Probabilidade	Impacto	Prioridade
1	Falha ao listar livros	2	5	1
2	Falha ao adicionar novos livros	3	5	2
3	Falha ao buscar um livro pelo ID	2	3	3
4	Falha ao editar detalhes de um livro	3	3	4
5	Falha ao remover um livro da lista	2	3	5
6	Frontend não é visualmente agradável ou fácil de usar	3	2	6
7	Backend não consegue lidar com todas as operações CRUD	5	5	7

Figura 4.4: Exemplo de matriz do PRisMA

Conforme mencionado na seção anterior sobre a fase de análise de riscos, Pawlak e Poniszewska-Marańda [33] recomendam o uso do método *Product Risk Management* (PRisMA). A principal ideia desse método é criar uma matriz de risco do produto, que contém valores numéricos de risco para cada item identificado. Com base nesses valores, deve-se empregar uma abordagem de teste diferente. Essa matriz deve ser incluída no plano de teste no campo correspondente.

A primeira fase desse método é a coleta de informações e artefatos do projeto, que inclui:

- reunião de documentos iniciais - identificação e reunião de informações pertinentes para a avaliação de riscos do produto, juntamente com a condução de uma análise estática dos documentos adquiridos;

- identificação dos elementos suscetíveis a riscos - listados com uma identificação exclusiva e explicação correspondente; sendo recomendada a presença de no máximo 30 a 35 itens de risco a fim de manter a fluidez do processo;
- determinação do impacto e fatores de análise - descritos em um documento abrangente direcionado a toda a organização;
- (opcional) definição de pesos para cada fator - a escala preferencial é a escala Likert, que abrange valores de 1 a 5;
- seleção das partes envolvidas - envolvendo várias funções de negócios e produção, tais como o gerente de projeto, desenvolvedores, arquiteto de sistema e usuário final;
- estipulação das diretrizes para a pontuação.

A fase subsequente de avaliação do risco por parte das partes envolvidas se desdobra na atribuição de pontuações aos elementos de risco de acordo com as orientações fornecidas. Tais pontuações baseiam-se em suposições individuais das partes interessadas, as quais também devem ser documentadas. A fase de coleta e verificação das avaliações de risco engloba a reunião dos resultados provenientes das avaliações feitas pelas partes envolvidas, seguida pela verificação da precisão das mesmas [33] [35].

Subsequentemente, delineia-se uma estratégia de teste distinta com base na posição dos riscos na matriz. Essa abordagem permite a identificação e priorização dos elementos a serem testados. Desse ponto, o escopo dos testes do projeto é definido, servindo como ponto de partida para a dedução das técnicas de concepção de teste necessárias [33] [35].

O passo seguinte compreende a seleção dos fatores que influenciam cada risco. Uma vez determinados os elementos suscetíveis a riscos e os fatores associados, é crucial escolher uma escala de classificação. Uma vez que não há um documento de nível superior delineando o procedimento de avaliação, pressupõe-se que qualquer escala possa ser selecionada. Seguindo os princípios delineados pelo Kungfu Testing, optou-se pela escala Likert, abarcando valores de 1 a 5, sem restrições suplementares quanto à pontuação [33].

4.7 Artefatos do *Kungfu Testing*

Durante a execução da abordagem *Kungfu Testing* são produzidos três tipos de artefatos: o plano de teste, o registro de problemas e o relatório de teste.

4.7.1 Plano de Teste

O plano de teste é um documento que descreve, de forma detalhada, como os testes serão conduzidos para garantir que o *software* atenda aos requisitos e funcione conforme o esperado.

Como descrito na etapa de planejamento de teste, o *Kungfu Testing* não define um modelo de plano de teste específico. Portanto, um modelo de plano de teste que pode ser seguido é a estrutura informada na ISO/IEC/IEEE [36].

A ISO/IEC/IEEE [36] propõe um plano de teste composto pelos seguintes campos:

- Contexto dos testes: campo que fornece informações sobre o contexto em que os testes serão realizados. Inclui informações sobre os projetos, níveis de teste e tipos de teste para os quais o plano está sendo escrito, os itens de teste a serem testados, o escopo dos testes e a base para o projeto e implementação dos testes.
- Suposições e restrições: campo que aborda as suposições e restrições que afetam os testes. Inclui padrões regulatórios, políticas de teste, práticas organizacionais, requisitos contratuais, limitações de tempo e orçamento do projeto, ferramentas e ambientes.
- Partes interessadas: campo que lista as partes interessadas e sua importância para os testes.
- Comunicação de testes: campo que aborda as formas de comunicação entre os testes, outras atividades do ciclo de vida e dentro da organização.
- Registro de riscos: campo que inclui a matriz de risco que foi elaborada utilizando o PRisMA.
- Estratégia de testes: campo que descreve a abordagem a ser adotada para a realização dos testes. Inclui a definição dos níveis de teste que serão executados, a identificação dos tipos de testes a serem realizados, a determinação dos entregáveis de teste, a especificação das técnicas de projeto de teste, a definição dos critérios de início e término dos processos de teste, a definição dos critérios de conclusão dos testes, a definição do grau de independência dos testadores, a coleta de métricas durante as atividades de teste, os requisitos de dados e ambiente de teste necessários para o projeto, o tratamento de pontos como reteste e teste de regressão, a definição de critérios para suspensão e retomada das atividades de teste e a definição de qualquer desvio das práticas de testes organizacionais.
- Atividades de teste e estimativas: campo que identifica todas as atividades de teste necessárias para implementar a estratégia de teste. Inclui a abordagem e o número

esperado de iterações para retrabalho e teste de regressão, quaisquer dependências entre as atividades, estimativas para cada atividade de teste e, quando apropriado, o orçamento de teste alocado e as estimativas de custo.

4.7.2 Registro de Problemas

Com base nas informações e elementos descritos por Pawlak e Poniszewska-Marańda [33] que devem estar contidos no registro de problemas, um modelo que pode ser utilizado é o *template* de *issues* do GitHub.

O *template* do GitHub Issues [37] possui os campos adequados para cada uma das informações que devem ser registradas:

- *Assignees*: Nesse campo, deve ser informado a pessoa a qual é delegado o problema;
- *Comment*: Nesse campo, deve ser descrito o problema;
- *Labels*: Nesse campo, deve ser marcado um dos status do rastreamento do problema que foram definidos para o projeto.

Para definir o nível de prioridade, deve ser criada uma *Milestone*. Todas as issues com os problemas registrados devem ser marcadas para esta *Milestone*.

Com isso, é possível ordenar as issues arrastando-as de acordo com sua prioridade.

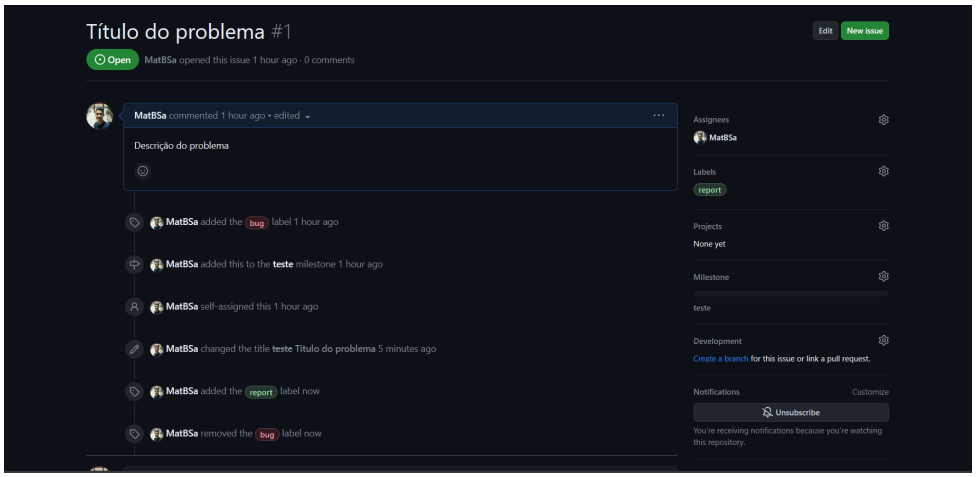


Figura 4.5: GitHub *Issue*.

4.7.3 Relatório de Teste

O relatório de teste é produzido na última etapa do ciclo da abordagem de gerenciamento de teste *Kungfu Testing*, conforme visto na seção anterior. Ele fornece um resumo dos testes realizados em um projeto ou em níveis e tipos de teste específicos [36].

Pawlak e Poniszewska-Marańda não descrevem um modelo específico para esse relatório de teste [33]. No entanto, um modelo que pode ser seguido é o descrito pela ISO/IEC/IEEE. A ISO/IEC/IEEE [36] propõe um relatório de teste composto pelos seguintes campos:

- Resumo dos testes realizados: essa seção fornece um resumo dos testes realizados em um projeto ou em níveis e tipos de teste específicos. Ele fornece detalhes sobre o que foi testado e descreve quaisquer restrições sobre como os testes foram realizados.
- Desvios dos testes planejados: essa seção descreve quaisquer desvios dos testes planejados. Se houver algum desvio, ele será detalhado nesta seção. Além disso, esta seção pode fazer referência à seção sobre riscos residuais para quaisquer riscos que os desvios possam representar para os testes e seus respectivos tratamentos de risco.
- Avaliação de conclusão do teste: essa seção descreve o grau em que o teste atendeu aos critérios de conclusão do teste especificados e também explica por que os critérios não foram atendidos.
- Fatores que bloquearam o progresso: essa seção descreve os fatores que impediram o progresso e as soluções correspondentes que foram implementadas para removê-los.
- Medidas de teste: essa seção descreve as medidas de teste agrupadas.
- Riscos residuais: essa seção relaciona os riscos que permaneceram sem tratamento após o término do teste. Isso pode incluir riscos que não foram completamente resolvidos pelo teste e/ou novos riscos descobertos durante o monitoramento final e encerramento do teste.
- Resultados do teste: essa seção relaciona todos os produtos de teste gerados a partir do teste e onde eles estão localizados.
- Ativos de teste reutilizáveis: essa seção listada todos os ativos de teste reutilizáveis e sua localização.
- Lições aprendidas: essa seção descreve os resultados da reunião de lições aprendidas.

4.8 Processo de desenvolvimento resultante

Como abordado na seção sobre o *Personal Extreme Programming* (PXP), essa processo tem como uma de suas virtudes a flexibilidade de sua estrutura [27]. Para melhorar a garantia e a qualidade de seus produtos de *software*, o PXP incorporará a abordagem de gerenciamento de teste *Kungfu Testing*, que é projetada para ser integrada às abordagens ágeis de desenvolvimento de *software* [33].

4.8.1 Configuração das atividades

Para incorporar a abordagem de teste *Kungfu Testing* ao PXP, é necessário integrar suas atividades ao fluxo existente. Uma maneira de realizar essa integração das atividades do *Kungfu Testing* ao PXP é ordenando as atividades de acordo com o contexto delas.

Para a configuração do PXP com as atividades do *Kungfu Testing*, deve ser incorporado uma nova fase ao seu fluxo, que deve ser nomeada de fase de Planejamento de Teste, que conterá a atividade realizada na fase de Planejamento do *Kungfu Testing*. Ela deve ser realizada após a execução do Planejamento, que nessa configuração será renomeada para Planejamento do Software. Essa atividade deve ter uma fase dedicada para que seja dado foco no que será realizado nela e na elaboração do plano de teste que será o artefato produzido em sua conclusão.

A fase de Teste de Sistema do PXP deve ser substituída pela fase de Execução do *Kungfu Testing*, que deve ser renomeada por Execução de Teste. Isso se deve pela fase Execução do *Kungfu Testing* incluir o que é realizado na fase de Teste de Sistema do PXP juntamente com outras etapas que a compõem, como o monitoramento e controle de teste, o gerenciamento de problemas e a etapa de relatório e avaliação.

4.8.2 Atividades

O novo fluxo de atividades que deve ser formado pela configuração do PXP com a incorporação das atividades da abordagem de gerenciamento de teste *Kungfu Testing* será:

- Requisitos: nessa fase, é necessário estabelecer os requisitos do software. A realização dessa atividade deve resultar, assim como o PXP original, na confecção de cartões com o registro de cada uma das histórias de usuário, incluindo o seu nível de prioridade no campo indicado no *template*;
- Planejamento do *Software*: nessa fase, são tomadas decisões cruciais de *design*, como a escolha da linguagem de programação, a estrutura de desenvolvimento e o modelo de aplicação. Também são realizadas estimativas de tempo e custo para cada tarefa, com base em projetos anteriores, resultando em um plano que define a execução do desenvolvimento;
- Planejamento de Teste: nessa fase, os procedimentos que são realizados na fase de Planejamento do *Kungfu Testing* são executados. Nela, são realizadas a análise de risco, estimativa de teste e organização do teste. Ao final dessa fase, é produzido um Plano de Teste conforme o modelo apresentado na seção de descrição do *Kungfu Testing*;

- Inicialização de Iteração: nessa fase, são escolhidos recursos específicos para serem implementados, e a duração da iteração normalmente varia de uma a três semanas. O objetivo principal é criar um produto funcional com as tarefas selecionadas para essa iteração;
- *Design*: nessa fase, é criada uma estrutura para as tarefas a serem realizadas no desenvolvimento. O *design* é adaptado para atender aos requisitos do sistema, e ferramentas familiares são usadas para tornar o processo mais eficiente;
- Implementação: nessa fase é feita a implementação do *software* e dos testes para os mesmos;
- Execução de teste: essa fase receberá a fase de Execução do *Kungfu Testing*. Nessa serão feitas além da execução do teste, o monitoramento e controle de teste, gerenciamento de problemas, relatório e avaliação;
- Retrospectiva: essa fase marca o fim de uma iteração e oferece a oportunidade de análise e aprimoramento. Nela, os dados coletados durante o processo são revisados, incluindo as estimativas de tempo e os resultados reais.

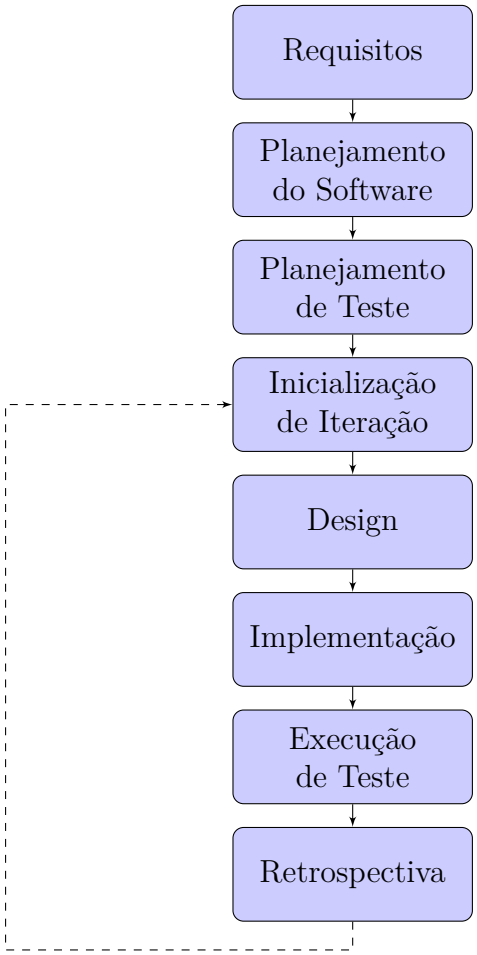


Figura 4.6: PXP configurado com o Kungfu Testing

4.8.3 Artefatos

O PXP configurado com a abordagem de gerenciamento de teste *Kungfu Testing* terá os mesmos artefatos produzidos no PXP original, além dos artefatos gerados nas atividades do *Kungfu Testing*.

Os artefatos a serem produzidos, como no PXP original, incluem cartões de história do usuário, código de implementação do *software* e, quando aplicável, códigos de testes automatizados. No caso dos cartões de história do usuário, eles devem seguir o mesmo modelo utilizado pelo PXP original, com os mesmos campos.

No que diz respeito às atividades originárias do *Kungfu Testing*, os artefatos a serem gerados, seguindo o *Kungfu Testing*, são o Plano de Teste, o Registro de Problemas e o Relatório de Teste.

Capítulo 5

Exemplo de aplicação de processo de desenvolvimento

Neste capítulo, inicialmente serão apresentados propósitos da aplicação desenvolvida. Em seguida, é descrito exemplo de aplicação do processo de desenvolvimento de software que foi configurado no capítulo anterior.

5.1 Propósitos da aplicação

A aplicação *web* LeiaMais foi desenvolvida no contexto deste trabalho com o objetivo de exemplificar a execução do processo de desenvolvimento de *software* configurado no capítulo anterior. O título da aplicação usa um imperativo para incentivar o usuário a ler mais livros e registrar informações sobre eles.

O LeiaMais consiste de um site no qual o usuário pode armazenar, gerenciar e acessar dados bibliográficos de qualquer local ou dispositivo de forma remota. O desenvolvimento da aplicação teve inspiração em produtos de *software* para gerenciamento de referências, como o Zotero e o Mendeley.

Por meio da aplicação desenvolvida, o usuário pode realizar as seguintes ações:

- Cadastrar novos livros que foram lidos;
- Visualizar a lista de livros lidos;
- Visualizar os dados dos livros;
- Atualizar os dados dos livros;
- Buscar por algum livro que tenha sido cadastrado;
- Remover os livros cadastrados.

Por meio da aplicação, o usuário também pode realizar as seguintes ações:

- Cadastrar novos autores que foram conhecidos;
- Visualizar a lista de autores;
- Visualizar os dados dos autores;
- Atualizar os dados dos autores;
- Buscar por algum autor que ele tenha cadastrado;
- Remover autores cadastrados.

5.2 Atividades realizadas

Nesta seção, serão descritas as atividades realizadas de acordo com o processo de desenvolvimento configurado no capítulo anterior, "*Personal Extreme Programming* (PXP) configurado com o "*Kungfu Testing*", no contexto da aplicação desenvolvida. Note que, considerando o tempo alocado ao desenvolvimento da aplicação, foi realizada apenas uma iteração.

5.2.1 Requisitos

Primeiramente, foi necessário identificar os requisitos. No processo de desenvolvimento de *software* configurado no capítulo anterior, esse é um procedimento realizado, assim como no PXP, por meio da coleta e elaboração de histórias de usuário em colaboração com as partes interessadas. No entanto, no contexto do desenvolvimento da aplicação em questão, as histórias de usuário não seguem a mesma abordagem de colaboração com as partes interessadas. Os requisitos que serão apresentados a seguir foram baseados em outros produtos de *software* similares, que serviram de inspiração para o projeto. Considerando isso, segue a relação das histórias de usuário elaboradas para a plataforma proposta:

1
Como usuário do produto, quero poder visualizar a lista de livros registrados.
Prioridade:
Estimativa de Tempo:

2
Como usuário do produto, quero poder adicionar novos livros à lista.
Prioridade:
Estimativa de Tempo:

3
Como usuário do produto, quero poder adicionar novos autores de livros ao sistema.
Prioridade:
Estimativa de Tempo:

4
Como usuário do produto, quero poder visualizar as informações de um livro que selecionei.
Prioridade:
Estimativa de Tempo:

5
Como usuário do produto, quero poder editar os detalhes de um livro registrado.
Prioridade:
Estimativa de Tempo:

6
Como usuário do produto, quero poder remover um livro da lista.
Prioridade:
Estimativa de Tempo:

7
Como usuário do produto, quero poder filtrar os livros por editora.
Prioridade:
Estimativa de Tempo:

5.2.2 Planejamento do projeto

Após a identificação dos requisitos, foi realizado o planejamento do projeto. Nessa fase, as tecnologias utilizadas no projeto foram definidas.

Para o armazenamento de dados, foi escolhido o MongoDB, um sistema de banco de dados orientado a documentos. Este sistema é conhecido por sua flexibilidade e escalabilidade, permitindo lidar com grandes volumes de dados de forma eficiente. Para modelagem de objetos MongoDB, utilizou-se a biblioteca Mongoose, facilitando o trabalho com o MongoDB. Além disso, para garantir a disponibilidade e segurança dos dados, foi definido o uso do serviço de banco de dados em nuvem MongoDB Atlas.

No que diz respeito ao desenvolvimento do *backend*, as tecnologias escolhidas foram: JavaScript, Node.js e Express.js. O JavaScript foi selecionado como a linguagem de programação devido à sua versatilidade e ampla adoção na comunidade de desenvolvimento. O Node.js foi utilizado como ambiente de execução para o JavaScript, permitindo que o código seja executado pelo servidor. Adicionalmente, o Express.js foi empregado como *framework* para o Node.js, facilitando a criação de rotas e a manipulação de solicitações e respostas HTTP.

Para o desenvolvimento do *frontend* deste projeto, as tecnologias escolhidas foram: JavaScript, Vue.js, Vuetify e Axios.js. O JavaScript foi escolhido para o desenvolvimento do *frontend* por ser uma linguagem amplamente utilizada no desenvolvimento de páginas *web* dinâmicas. O Vue.js foi selecionado por sua facilidade de uso e flexibilidade na criação de interfaces de usuário ricas e interativas. O Vuetify foi escolhido para proporcionar agilidade e facilidade na montagem dos elementos das páginas da aplicação. Por fim, o Axios.js foi escolhido para facilitar a manipulação das solicitações HTTP.

O ambiente de desenvolvimento será composto pelo sistema operacional Windows 11 e pelo editor de texto Visual Studio Code. Também serão utilizados o Git para o versionamento do código e o GitHub para o armazenamento em nuvem e *backup* dos códigos.

Nesta fase também foram definidas as prioridades e estimativas de tempo para cada uma das histórias de usuários. Os valores foram atribuídos conforme abaixo:

Atividade	**Prioridade**	**Data de Início**	**Data de Término**
Cartão 1	Prioridade 1	10/09	11/09
Cartão 2	Prioridade 2	12/09	12/09
Cartão 3	Prioridade 3	13/09	13/09
Cartão 4	Prioridade 4	14/09	14/09
Cartão 5	Prioridade 5	15/09	15/09
Cartão 6	Prioridade 6	16/09	16/09
Cartão 7	Prioridade 7	17/09	17/09

Tabela 5.1: Cronograma de Desenvolvimento de Funcionalidades

5.2.3 Planejamento do Teste

Seguindo o processo, a próxima fase consiste na análise de risco, estimativa de teste e organização do teste para o projeto da aplicação, visando à elaboração do Plano de Teste.

Para essa fase, foi utilizado o modelo de Plano de Teste do IEEE indicado no capítulo para o processo de desenvolvimento PXP configurado com o *Kungfu Testing*. Devido ao projeto não possuir cliente ou outras partes interessadas, apenas o próprio desenvolvedor, o tópico Partes Interessadas não houve necessidade de ser preenchido. Também não foram necessários os tópicos Comunicação de Testes e Alocação de Pessoal, uma vez que o projeto é realizado por um único desenvolvedor e não há necessidade de um meio de comunicação ou alocação.

Portanto, foram realizadas as análises e coletadas as informações necessárias, e o modelo foi preenchido conforme será apresentado a seguir:

Plano de Teste

Contexto dos testes

Este plano de teste é para o site LeiaMais. O escopo dos testes inclui todas as funcionalidades do site descritas nas histórias de usuário, incluindo a visualização da lista de livros, adição de novos livros à lista, visualização das informações de um livro selecionado, edição dos detalhes de um livro e remoção de um livro da lista.

Suposições e restrições

Os testes serão realizados de acordo com as melhores práticas de teste de *software*. As restrições incluem tempo e recursos limitados. As ferramentas de teste incluirão o Cypress para o *frontend* e o Jest para o *backend*.

Registro de riscos

ID	Descrição do Risco	Probabilidade	Impacto	Prioridade
1	Falha ao listar livros	2	5	1
2	Falha ao adicionar novos livros	3	5	2
3	Falha ao buscar um livro pelo ID	2	3	3
4	Falha ao editar detalhes de um livro	3	3	4
5	Falha ao remover um livro da lista	2	3	5
6	Frontend não é visualmente agradável ou fácil de usar	3	2	6
7	Backend não consegue lidar com todas as operações CRUD	5	5	7

Estratégia de testes

A estratégia de testes para o *site* LeiaMais tem o propósito de prover uma cobertura de teste que garanta a qualidade do *software*. Ela é composta pelos seguintes elementos:

Níveis de Teste

Os níveis de teste serão definidos de acordo com a complexidade das funcionalidades do sistema. Isso incluirá testes unitários, testes de integração e testes de ponta a ponta.

Entregáveis de Teste

Os entregáveis de teste incluirão planos de teste, *scripts* de teste, dados de teste e relatórios de teste.

Critérios de Início e Término

A seguir são definidos critérios de início e término para diferentes níveis de teste (unitário, integração e ponta a ponta).

- **Critérios de Início:**
  - *Testes unitários:* Os testes unitários começarão assim que um módulo de código estiver pronto para teste.

  - *Testes de integração:* Os testes de integração começarão após a conclusão bem-sucedida dos testes unitários.
  - *Testes de ponta a ponta:* Os testes de ponta a ponta começarão após a conclusão bem-sucedida dos testes de integração.

- **Critérios de Término:**

  - *Testes unitários:* Os testes unitários serão considerados concluídos quando todos os módulos de código passarem nos testes unitários e uma cobertura de código de 60% for alcançada.
  - *Testes de integração:* Os testes de integração serão considerados concluídos quando todas as interfaces entre os módulos funcionarem conforme esperado e uma cobertura de código de 60% for alcançada.
  - *Testes de ponta a ponta:* Os testes de ponta a ponta serão considerados concluídos quando a aplicação como um todo funcionar conforme esperado, todos os cenários de uso real forem cobertos e passarem nos testes, e uma cobertura de código de 60% for alcançada.
  - Além disso, cada fase de teste será considerada concluída quando todos os defeitos encontrados durante essa fase forem removidos. Se um defeito não puder ser removido imediatamente, ele será documentado e rastreado para correção em uma data posterior.

Coleta de Métricas

As métricas serão coletadas durante as atividades de teste para monitorar o progresso do teste e a qualidade do *software*. As métricas coletadas em cada fase de teste são as seguintes:

- **Testes unitários:** Durante os testes unitários, as métricas coletadas incluem o número de testes realizados, o número de testes que não resultaram em falha, o número de testes que resultaram em falha, a cobertura de código e o tempo necessário para executar os testes.
- **Testes de integração:** Durante os testes de integração, as métricas coletadas incluem o número de interfaces testadas, o número de interfaces que não apresentaram falha, o número de interfaces que apresentaram falha, a cobertura de código e o tempo necessário para executar os testes.

- **Testes de ponta a ponta:** Durante os testes de ponta a ponta, as métricas coletadas incluem o número de cenários de uso testados, o número de cenários de uso que não apresentaram falha, o número de cenários de uso que apresentaram falha, a cobertura de código e o tempo necessário para executar os testes.

Critérios para Suspensão e Retomada das Atividades de Teste

Os critérios para suspensão e retomada das atividades de teste são definidos para lidar com problemas que possam surgir durante o processo de teste.

- **Critérios de Suspensão:**
  - Se um número significativo de defeitos de alta prioridade for encontrado, as atividades de teste podem ser suspensas até que esses defeitos sejam corrigidos. Defeitos de alta prioridade são aqueles que têm um impacto significativo na funcionalidade ou desempenho do software e precisam ser corrigidos o mais rápido possível.
  - Se o ambiente de teste ficar indisponível ou instável, as atividades de teste serão suspensas até que o ambiente esteja novamente disponível e estável.
  - Se os dados de teste necessários não estiverem disponíveis ou forem insuficientes, as atividades de teste serão suspensas até que estejam disponíveis. Dados de teste são usados para testar o *software*, passando para as funcionalidades do mesmo.
- **Critérios de Retomada:**
  - As atividades de teste serão retomadas assim que os defeitos de alta prioridade forem corrigidos e verificados.
  - As atividades de teste serão retomadas assim que o ambiente de teste estiver disponível e estável.
  - As atividades de teste serão retomadas assim que os dados de teste adequados estiverem disponíveis.

Atividades de Teste e Estimativas

A implementação dos testes serão realizadas após as implementações das histórias de usuário, seguindo o seguinte cronograma:

Cronograma

Atividade	Data de Início	Data de Término
Desenvolvimento de testes unitários para o backend e frontend	18/09	19/09
Desenvolvimento de testes de integração para o backend e frontend	20/09	21/09
Desenvolvimento de testes de ponta a ponta abrangendo cenários de uso real	22/09	23/09
Execução dos testes automatizados no ambiente de desenvolvimento e produção	23/09	23/09

5.2.4 Inicialização da Iteração

Nesta fase do processo de desenvolvimento, o ambiente de desenvolvimento foi preparado e as histórias de usuário foram organizadas para a realização das atividades de projeto (*design*), implementação, execução de testes e retrospectiva. Conforme informado anteriormente, foi realizada apenas uma iteração.

5.2.5 Projeto (*design*)

Projeto (*design*) da aplicação

Nessa fase, foi realizado o projeto (*design*) do banco de dados e do *backend*, conforme as figuras mais abaixo. Também foram definidas arquitetura e organização de arquivos de códigos.

No projeto do banco de dados do LeiaMais, a estrutura é composta por duas entidades principais: *Livro* e *Autor*. A entidade *Livro* possui os atributos *id*, *titulo*, *editora*, *paginas* e *autor*. O atributo *autor* é uma referência à entidade *Autor*, que possui os atributos *id*, *nome* e *nacionalidade*. A relação entre *Livro* e *Autor* é representada pelo relacionamento *possui*, indicando que cada *Livro* possui um *Autor*. A Figura 5.2 ilustra o diagrama de classes do banco de dados. Esta organização permite uma estrutura de dados eficiente e flexível, facilitando operações de CRUD (Criação, Leitura, Atualização e Deleção) nos dados do LeiaMais.

No caso do *backend*, foi definida a arquitetura *Model View Controller* (MVC). A Figura 5.1 mostra as relações entre os principais arquivos, destacando relações entre **app.js**, **dbConnect.js**, **index.js**, **autorController.js**, **livroController.js**, **Autor.js** e **Livro.js**.

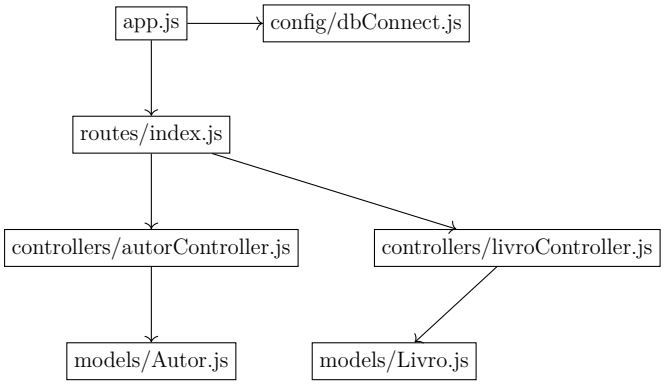


Figura 5.1: Relações entre arquivos do *backend* do LeiaMais.

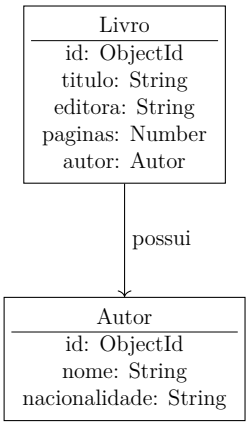


Figura 5.2: Diagrama de classes do banco de dados.

Projeto (*design*) dos testes automatizados

No que tange ao *backend* do LeiaMais, a arquitetura de teste unitário seguirá uma abordagem modular, sendo a execução dos testes conduzida pelo Jest. A estrutura dos testes unitários refletirá a organização do código-fonte, onde cada arquivo de teste corresponderá a um módulo específico do *backend*. Essa metodologia busca proporcionar uma cobertura abrangente dos casos de uso.

Em relação à arquitetura de teste de integração do *backend* com o banco de dados do LeiaMais, ela também será implementada por meio do Jest. Os testes de integração se concentrarão em verificar a correta interação entre o banco de dados e o *backend* da aplicação, garantindo que suas interações produzam os resultados esperados. Além disso, esses testes verificarão se as chamadas de API retornarão as respostas esperadas.

No âmbito do *frontend* do projeto LeiaMais, a arquitetura de teste unitário será implementada por meio do Cypress. Cada componente Vue.js será submetido a testes individuais, visando garantir o seu comportamento adequado em diversas condições. Cada teste unitário isolará um componente, verificando uma pequena unidade de funcionalidade e assegurando o funcionamento independente de cada parte do *frontend*. O Cypress permitirá a simulação de interações do usuário, possibilitando a verificação das alterações resultantes no estado do aplicativo e na interface do usuário. Adicionalmente, o Cypress oferece recursos para simular chamadas de API, permitindo a execução dos testes unitários em um ambiente controlado.

Por fim, a arquitetura de teste de ponta a ponta para o LeiaMais é elaborada com o uso do Cypress, com o objetivo de validar a integração e funcionalidade de todo o sistema. Os testes de ponta a ponta abrangem cenários completos de interação do usuário, simulando desde a navegação por diferentes páginas até a execução de operações complexas, como cadastro e atualização de autores e livros. O Cypress permite a criação de *scripts* claros e concisos, facilitando a manutenção e extensão dos casos de teste. Ao utilizar *mocks* e *fixtures*, é possível controlar o ambiente de teste, garantindo consistência e previsibilidade nos resultados. Assim, a arquitetura de teste de ponta a ponta para o LeiaMais visa garantir a qualidade do sistema desde a interface do usuário até a interação com o *backend*.

A Figura 5.3 representa as diferentes camadas do sistema (*frontend*, *backend* e banco de dados) e os diferentes tipos de testes realizados em cada camada. Os Testes Unitários do *frontend* e do *backend* são realizados em cada camada individualmente, enquanto os Testes de Integração são realizados entre o *frontend* e o *backend*. Os Testes de Ponta a Ponta, por outro lado, abrangem todo o sistema, desde o *frontend* até o banco de dados.

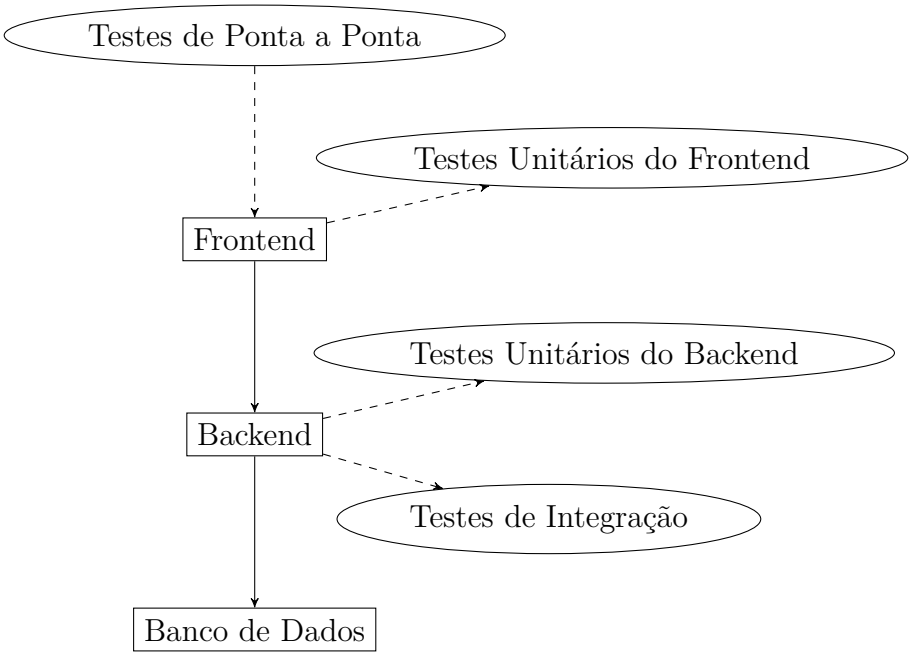


Figura 5.3: Diagrama de testes

5.2.6 Implementação

Nessa fase, foram implementadas as *features* das histórias de usuário e os seus testes automatizados.

Implementação da aplicação

No contexto da aplicação, foi implementado o *endpoint* de cada *feature*, seguida pela implementação de sua interface de usuário no *frontend*.

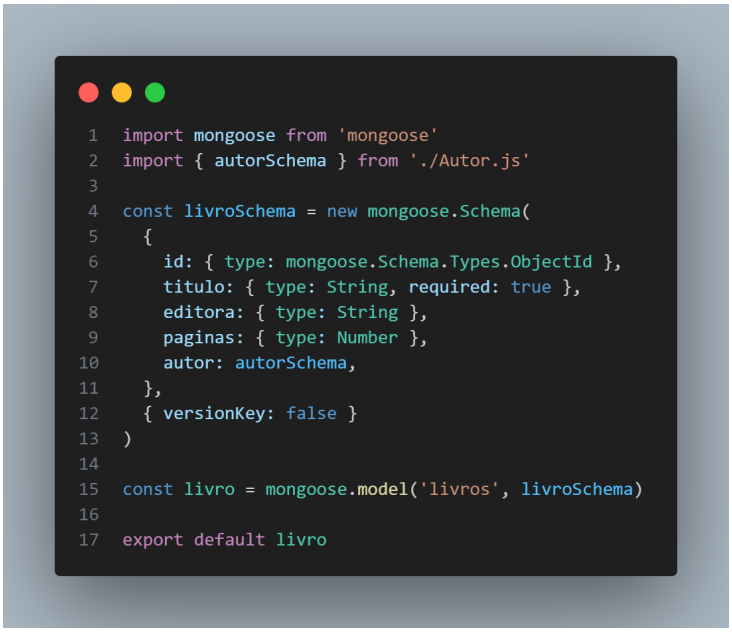
No backend, temos o arquivo `dbConnect.js` (Figura 5.4), que é responsável por estabelecer a conexão com o banco de dados MongoDB. O arquivo `Livro.js` (Figura 5.6) define o esquema do livro no banco de dados. O arquivo `livroController.js` (Figura 5.5) é um controlador que define várias funções para manipular os livros no banco de dados, como listar todos os livros, encontrar um livro por ID, adicionar um novo livro, atualizar um livro existente e remover um livro. Finalmente, o arquivo `livrosRoutes.js` (Figura 5.7) define as rotas para o controlador do livro.

```
1  import mongoose from 'mongoose'
2
3  async function conectaDatabase() {
4    const username = process.env.DB_USER
   NAME
5    const password = process.env.DB_PASS
   WORD
6    const dbUrl = process.env.DB_URL
7
8    mongoose.connect(`mongodb+srv://${us
   ername}:${password}@${dbUrl}`)
9
10   return mongoose.connection
11 }
12
13 export default conectaDatabase
14
```

Figura 5.4: Código do arquivo `dbConnect.js` que estabelece a conexão com o banco de dados MongoDB.

```
import { autor } from '../models/Autor.js'
import livro from '../models/Livro.js'

class LivroController {
  static async listarLivros(req, res) {
    try {
      const listaLivros = await livro.find({})

      res.status(200).json(listaLivros)
    } catch (error) {
      res
        .status(500)
        .json({ message: `${error.message} - falha na requisição` })
    }
  }

  static async listarLivroPorId(req, res) {
    try {
      const id = req.params.id
      const livroEncontrado = await livro.findById(id)

      res.status(200).json(livroEncontrado)
    } catch (error) {
      res
        .status(500)
        .json({ message: `${error.message} - falha na requisição do livro` })
    }
  }

  static async listaLivrosPorEditora(req, res) {
    const editora = req.query.editora

    try {
      const livrosPorEditora = await livro.find({ editora: editora })
      res.status(200).json(livrosPorEditora)
    } catch (error) {
      res.status(500).json({ message: `${error.message} - falha na busca` })
    }
  }
```

Figura 5.5: Código do arquivo `livroController.js` que define o esquema do livro no banco de dados.



```
import mongoose from 'mongoose'
import { autorSchema } from './Autor.js'

const livroSchema = new mongoose.Schema(
  {
    id: { type: mongoose.Schema.Types.ObjectId },
    titulo: { type: String, required: true },
    editora: { type: String },
    paginas: { type: Number },
    autor: autorSchema,
  },
  { versionKey: false }
)

const livro = mongoose.model('livros', livroSchema)

export default livro
```

Figura 5.6: Código do arquivo `Livro.js` que define várias funções para manipular os livros no banco de dados.

```
import express from 'express'
import LivroController from '../controllers/livroController.js'

const routes = express.Router()

routes.get('/livros', LivroController.listarLivros)

routes.get('/livros/busca_por_editora', LivroController.listaLivrosPorEditora)

routes.get('/livros/buscar/:id', LivroController.listarLivroPorId)

routes.post('/livros/cadastrar', LivroController.cadastrarLivros)

routes.put('/livros/atualizar/:id', LivroController.atualizarLivroPorId)

routes.delete('/livros/remover/:id', LivroController.removerLivroPorId)

export default routes
```

Figura 5.7: Código do arquivo `livrosRoutes.js` que define as rotas para o controlador do livro.

No frontend, temos o arquivo `AuthorForm.vue` (Figura 5.9), que é um componente Vue.js para um formulário de autor, no caso da figura, apenas a parte do *template* está sendo apresentada. Este formulário permite ao usuário inserir informações sobre um autor, como nome e nacionalidade. O arquivo `BookService.js` (Figura 5.8) é um serviço que faz requisições HTTP para a API de livros e autores usando o Axios.

```
import axios from 'axios'

const apiClient = axios.create({
  baseURL: process.env.VUE_APP_BASE_URL,
  headers: {
    'Content-type': 'application/json',
  },
})

export default {
  getBooks() {
    return apiClient.get('/livros')
  },
  getBook(id) {
    return apiClient.get(`/livros/buscar/${id}`)
  },
  postBook(book) {
    return apiClient.post('/livros/cadastrar', book)
  },
  postAuthor(author) {
    return apiClient.post('/autores/cadastrar', author)
  },
  putBook(id, book) {
    return apiClient.put(`/livros/atualizar/${id}`, book)
  },
  deleteBook(id) {
    return apiClient.delete(`/livros/remover/${id}`)
  },
  getAuthors() {
    return apiClient.get('/autores')
  },
}
```

Figura 5.8: Código do arquivo `BookService.js` que é um componente Vue.js para um formulário de autor.

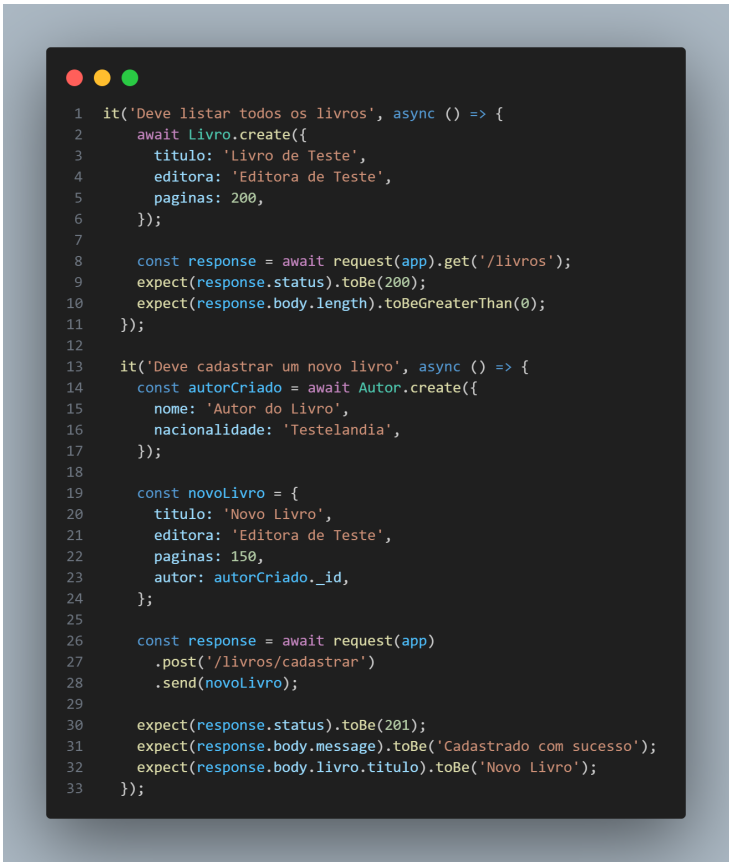
```vue
<template>
  <v-dialog v-model="dialog" max-width="400">
    <template v-slot:activator="{ on, attrs }">
      <v-btn color="green" dark v-bind="attrs" v-on="on">Novo Autor</v-btn>
    </template>
    <v-card>
      <v-card-title>
        <span class="headline">Adicionar novo autor</span>
      </v-card-title>
      <v-card-text>
        <v-container>
          <v-row>
            <v-col cols="12">
              <v-text-field
                id="author_name_id"
                v-model="author.nome"
                label="Nome"
                required
              ></v-text-field>
            </v-col>
            <v-col cols="12">
              <v-text-field
                id="author_nationality_id"
                v-model="author.nacionalidade"
                label="Nacionalidade"
              ></v-text-field>
            </v-col>
          </v-row>
        </v-container>
      </v-card-text>
      <v-card-actions>
        <v-spacer></v-spacer>
        <v-btn id="close_id" color="blue darken-1" text @click="resetForm">Fechar</v-btn>
        <v-btn id="save_id" color="blue darken-1" text @click="saveAuthor">Salvar</v-btn>
      </v-card-actions>
    </v-card>
  </v-dialog>
</template>
```

Figura 5.9: Código do arquivo `AuthorForm.vue` que é um serviço que faz requisições HTTP para a API de livros e autores usando o Axios.

Implementação dos testes automatizados

Após a implementação das *features* da aplicação, foram implementados os testes determinados no plano de testes tanto para o *backend* quanto para o *frontend*.

Dentre os vários *scripts* de testes implementados para o *backend*, a figura 5.10 apresenta parte dos testes do arquivo `livrosRoutes.test.js`. Este arquivo contém testes para listar todos os livros e cadastrar um novo livro. Esses testes verificam se as rotas retornam o status HTTP correto e se o corpo da resposta contém as informações esperadas.



```
it('Deve listar todos os livros', async () => {
    await Livro.create({
      titulo: 'Livro de Teste',
      editora: 'Editora de Teste',
      paginas: 200,
    });

    const response = await request(app).get('/livros');
    expect(response.status).toBe(200);
    expect(response.body.length).toBeGreaterThan(0);
  });

  it('Deve cadastrar um novo livro', async () => {
    const autorCriado = await Autor.create({
      nome: 'Autor do Livro',
      nacionalidade: 'Testelandia',
    });

    const novoLivro = {
      titulo: 'Novo Livro',
      editora: 'Editora de Teste',
      paginas: 150,
      autor: autorCriado._id,
    };

    const response = await request(app)
      .post('/livros/cadastrar')
      .send(novoLivro);

    expect(response.status).toBe(201);
    expect(response.body.message).toBe('Cadastrado com sucesso');
    expect(response.body.livro.titulo).toBe('Novo Livro');
  });
```

Figura 5.10: *Scripts* de testes do Jest para as rotas de livros no *backend*.

Na figura 5.11 é apresentado um exemplo de *script* de teste contido no arquivo `author_form_e2e.cy.js`, que simula a submissão de um novo autor. Este teste verifica se a requisição *POST* enviada contém as informações corretas e se a interface do usuário responde adequadamente à ação do usuário. Esses testes ajudam a garantir que a experiência do usuário seja suave e livre de erros.

```
describe('AuthorForm.vue', () => {
  beforeEach(() => {
    cy.visit('/')
  })

  it('submits new author', () => {
    const authorName = 'Novo Autor'
    const authorNationality = 'Brasileira'

    cy.intercept('POST', '/autores/cadastrar', (req) => {
      assert.equal(req.body.nome, authorName)
      assert.equal(req.body.nacionalidade, authorNationality)
    }).as('postAuthor')

    cy.get('button').contains('Novo Autor').click()
    cy.get('#author_name_id').type(authorName)
    cy.get('#author_nationality_id').type(authorNationality)
    cy.get('#save_id').click()

    cy.wait('@postAuthor')
  })
```

Figura 5.11: *Script* de teste do Cypress para a submissão de um novo autor no *frontend*.

5.2.7 Execução de teste

Nesta fase, foram executados os testes automatizados implementados na fase de Implementação e avaliados os resultados desses testes para identificar e acompanhar eventuais problemas. Os testes foram executados conforme definido no plano de teste.

Primeiro foram realizados os testes unitários, seguidos pelos testes de integração do *backend*, ambos foram realizados com o Jest. Depois foram realizados os testes unitários, seguidos pelos testes de ponta a ponta (E2E) do *frontend* realizados com o Cypress. Estes testes automatizados foram complementados por alguns testes manuais para elementos da aplicação que não eram adequados para testes automatizados, bem como para verificar se os problemas informados pelos testes realmente ocorriam ou se tratavam de problemas na execução do teste.

Os resultados dos testes do Jest para o *backend*, que incluíam testes de integração e também testes de unidade, mostraram que todos os testes passaram com sucesso. No entanto, algumas linhas em **autorController.js** e **livroController.js** não foram cobertas pelos testes, indicando áreas para melhorias futuras nos testes. A Figura 5.12 apresenta os resultados desses testes.

Os resultados dos testes de ponta a ponta (E2E) do *frontend* do projeto mostraram que a maioria dos testes passou, com exceção de um teste em **book_info_modal_e2e.cy.js**. Este teste indicou um problema na funcionalidade de editar informações do livro. Após perceber o problema, foi realizado um teste manual para verificar se o problema de fato ocorria. Em seguida, o problema foi registrado em um *issue* do GitHub para avaliação e

correção na próxima iteração do processo de desenvolvimento. A Figura 5.14 mostra os resultados dos testes E2E, enquanto a Figura 5.15 ilustra o problema indicado por um dos testes do Cypress e a Figura 5.16 mostra o registro de problema feito no *issue* do GitHub.

Finalmente, os resultados dos testes de unidade do *frontend* mostraram que todos os testes passaram, indicando que as funcionalidades testadas estão sendo providas como esperado. A figura 5.13 apresenta os resultados desses testes.

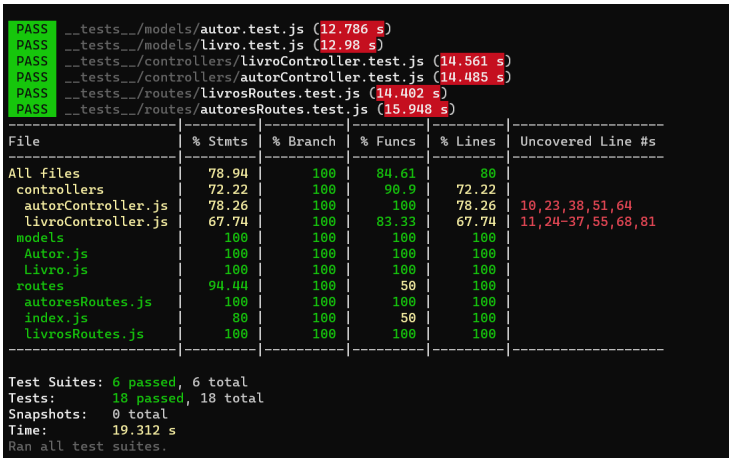


Figura 5.12: Resultados dos testes para o *backend* realizados com o Jest

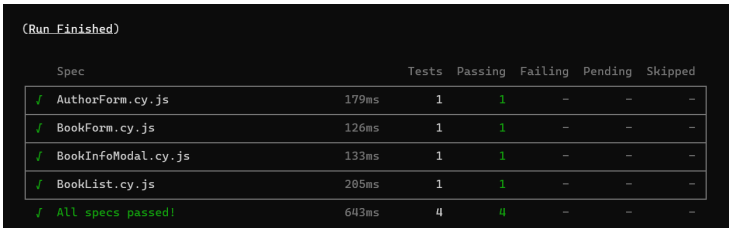


Figura 5.13: Resultados dos testes de unidade para o *frontend* realizados com o Cypress

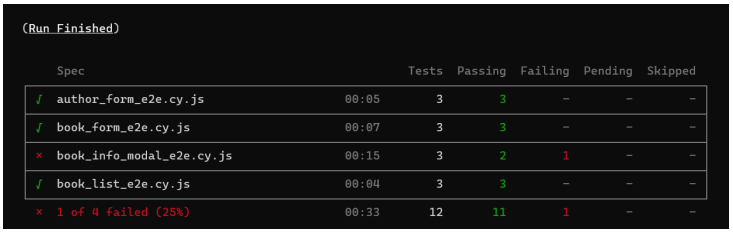


Figura 5.14: Resultados dos testes de ponta a ponta (E2E) para o *frontend* realizados com o Cypress

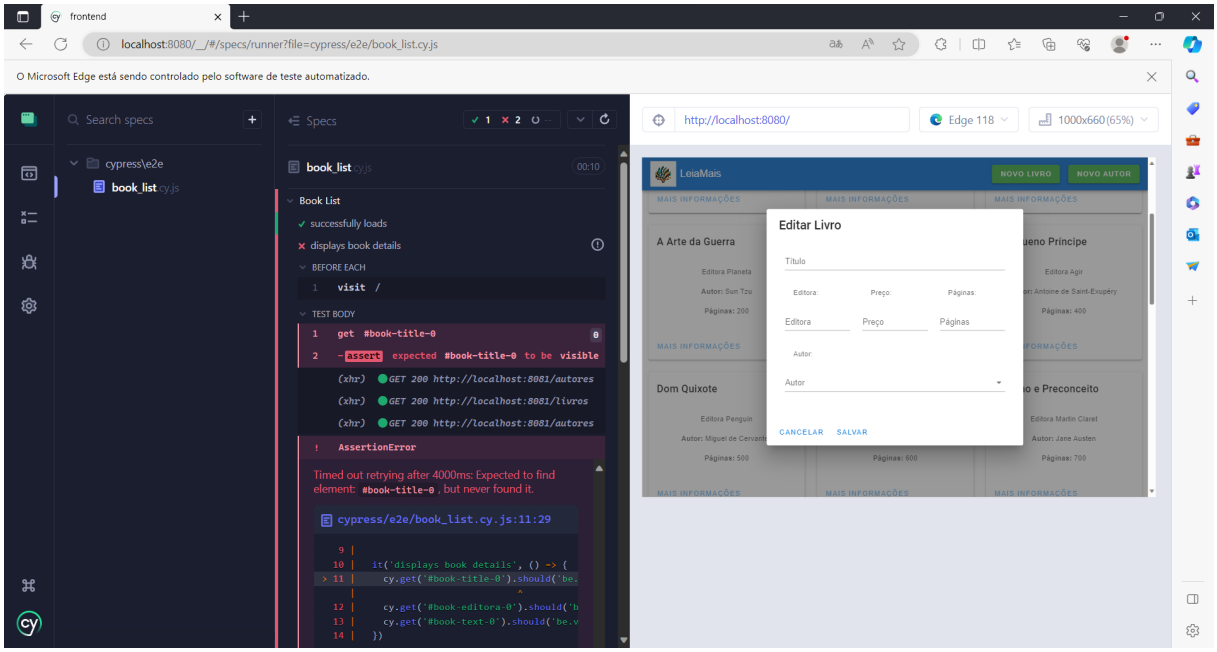


Figura 5.15: Problema indicado por um dos testes do Cypress

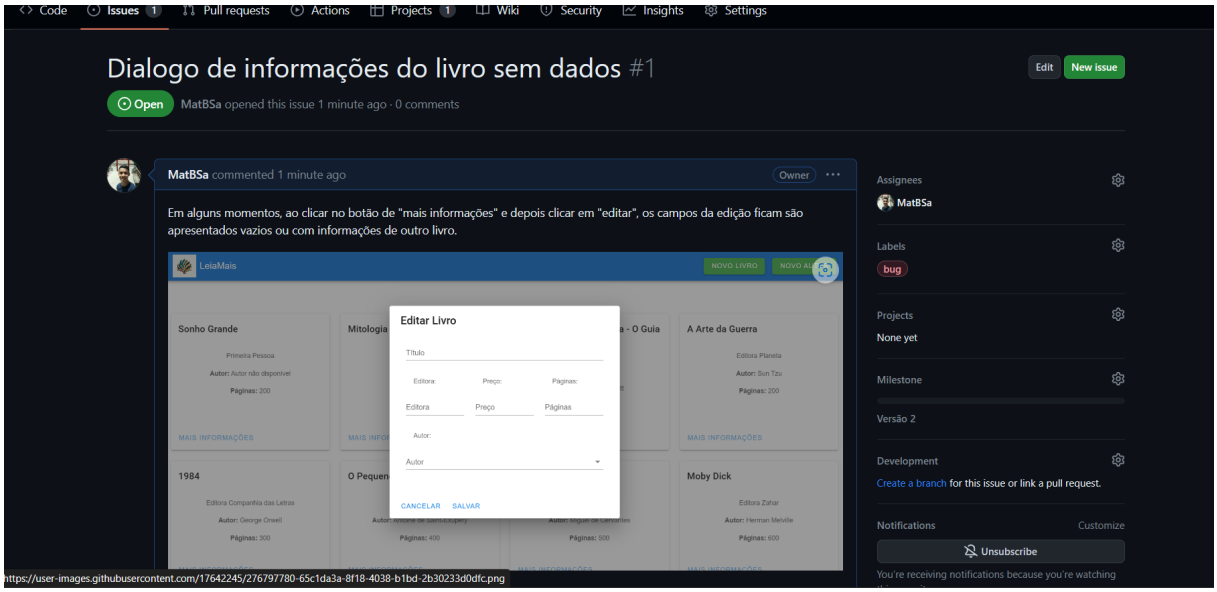


Figura 5.16: Registro de problema feito no *issue* do GitHub

Por fim, foi elaborado um Relatório de Teste com base nos resultados dos testes automatizados e manuais que foram realizados nessa fase.

Relatório de Teste

Resumo dos Testes Realizados

Os testes abrangeram os seguintes aspectos:

- Testes de unidade no *backend*.
- Testes de integração no *backend*.
- Testes de aceitação no *frontend* usando Cypress.

Desvios dos Testes Planejados

Durante os testes, identificamos os seguintes desvios dos testes planejados:

- O teste de de edição de informações de livro no *frontend* revelou um defeito que faz com que os campos da caixa de diálogo não sejam preenchidas ou apresente informações de outro livro.

Avaliação de Conclusão do Teste

Os testes foram concluídos em conformidade com os critérios especificados, exceto pelo defeito na caixa de diálogo de edição de livro. O defeito não estava dentro do escopo dos critérios de conclusão do teste.

Fatores que Bloquearam o Progresso

Nenhum fator bloqueou o progresso dos testes.

Medidas de Teste

As medidas de teste incluem:

- Execução de testes unitários no *backend* com Jest.
- Execução de testes unitários no *frontend* com Cypress.
- Execução de testes de integração no *frontend* com Cypress.

Riscos Residuais

Após a conclusão dos testes, foi identificado apenas, como risco residual, o defeito com a edição dos dados de livros, o qual é um mal comportamento da aplicação e gera um prejuízo na experiência do usuário e pode levá-lo a editar incorretamente os dados do livro.

Resultados do Teste

Os resultados dos testes foram registrados em arquivos de *logs* gerados pelas próprias ferramentas de testes utilizadas.

Ativos de Teste Reutilizáveis

Os ativos de teste reutilizáveis incluem *scripts* de teste, cenários de teste e dados de teste. Eles estão disponíveis para futuros testes.

5.2.8 Retrospectiva

Nessa fase, foi avaliado quais as funcionalidades da aplicação haviam sido implementadas. Além disso, foi revisado o projeto da aplicação para a realização de algumas mudanças do que foi planejado inicialmente. Por exemplo, a funcionalidade de cadastrar o autor separadamente dos livros deve ser desfeita, e o registro junto ao livro deve ser a forma de funcionamento, a fim de facilitar o uso pelo usuário final da aplicação *web* LeiaMais.

Também foi realizada a análise do Relatório de Teste para avaliar se a aplicação estava apta a ter seu *release* realizado. Nela, foi definido que o problema registrado sobre o defeito com a funcionalidade de edição dos dados dos livros comprometeria a experiência do usuário.

Logo, a aplicação não teve seu *release* feito para que, na próxima iteração, seja corrigido o problema encontrado e as mudanças que foram repensadas sejam implementadas.

5.3 Interface do usuário

Após a conclusão da iteração do processo de desenvolvimento realizada na seção anterior, o LeiaMais tem parte de sua aplicação *web* construída e usável, na qual o usuário pode realizar as operações propostas nos cartões de história de usuário.

Ao acessar o endereço da aplicação, conforme pode ser visualizado na figura 5.17, o usuário terá, de imediato, acesso à sua lista de livros cadastrados, os quais podem ser

visualizados em um cartão para cada livro com algumas informações: título, editora, autor e quantidade de páginas. Os cartões dos livros também possuem o botão MAIS INFORMAÇÕES", que terá sua caixa de diálogo apresentada mais abaixo, onde o usuário encontrará mais informações sobre o livro e também a possibilidade de editá-las. Além disso, há, na barra superior, no canto direito, os botões para NOVO LIVRO" e "NOVO AUTOR", os quais terão suas caixas de diálogos descritas mais abaixo.

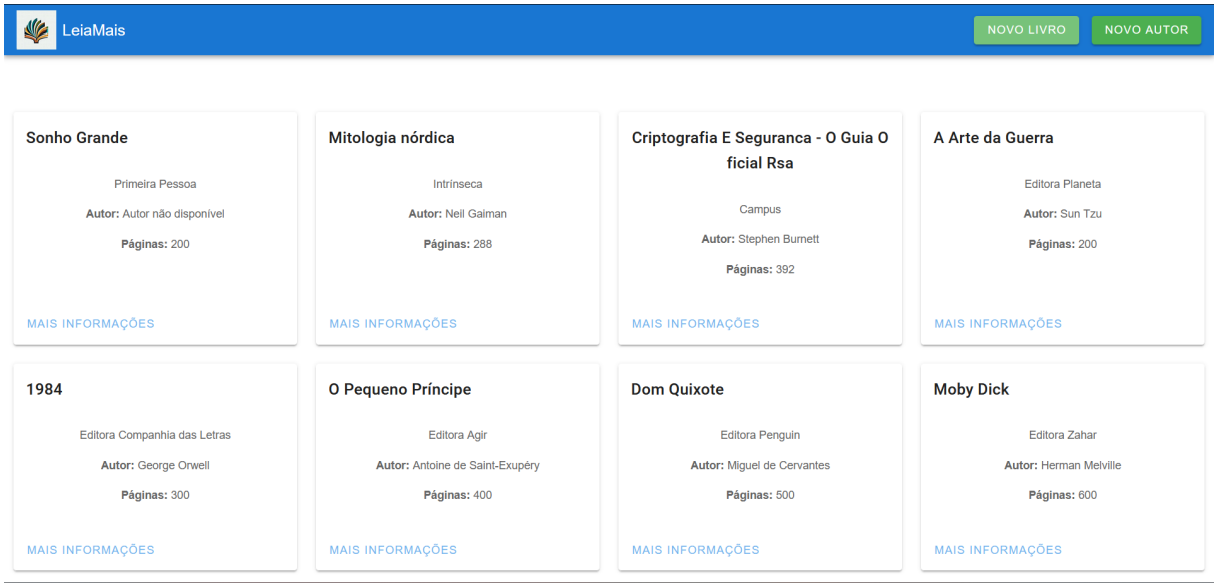


Figura 5.17: Página principal do LeiaMais com listagem dos livros.

Ao clicar no botão "NOVO LIVRO", o usuário abrirá uma caixa de diálogo que pode ser vista na figura 5.18, onde encontrará os campos de texto para inserir as informações de título e editora, o campo de texto modificado para inserir o número de páginas do livro e, por fim, o seletor para escolher o autor do livro com base nos autores que foram cadastrados na aplicação anteriormente. Após inserir as informações, o usuário deverá clicar no botão "SALVAR" para cadastrar o livro na aplicação. Caso o usuário queira desistir da operação, ele poderá clicar em qualquer área fora da caixa de diálogo ou clicar no botão "FECHAR" para que a caixa de diálogo seja fechada e os dados inseridos apagados.

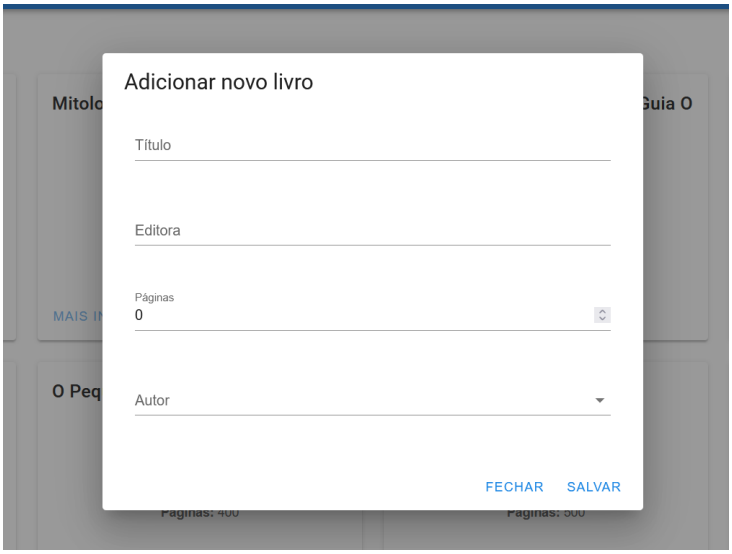


Figura 5.18: Caixa de diálogo para cadastrar novo livro.

Quando o usuário clicar no botão “NOVO AUTOR”, abrirá uma caixa de diálogo que pode ser vista na figura 5.19, onde encontrará os campos de texto para inserir as informações de nome e nacionalidade. Após inserir as informações, o usuário deverá clicar no botão “SALVAR” para cadastrar o autor na aplicação. Caso o usuário queira desistir da operação, ele poderá clicar em qualquer área fora da caixa de diálogo ou clicar no botão “FECHAR” para que a caixa de diálogo seja fechada e os dados inseridos apagados.

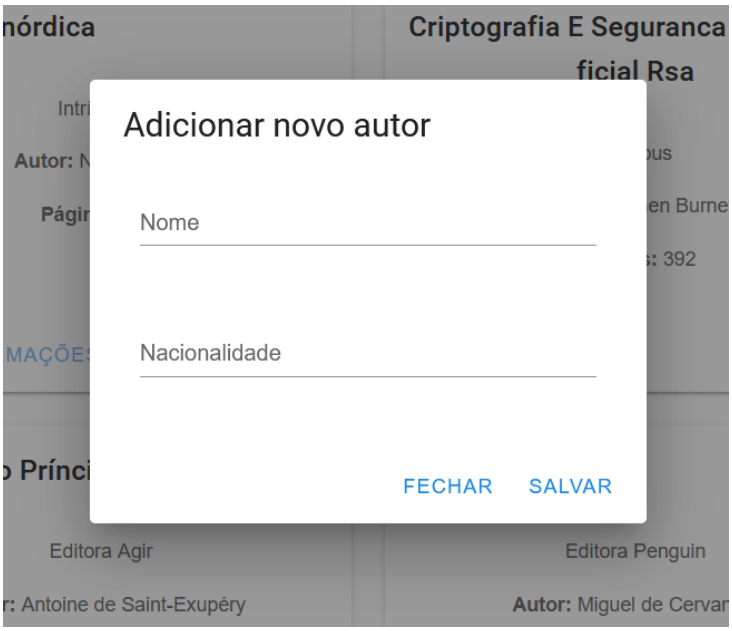


Figura 5.19: Caixa de diálogo para cadastrar novo autor.

Clicando no botão “MAIS INFORMAÇÕES”, o usuário abrirá uma caixa de diálogo que pode ser vista na figura 5.20, onde encontrará as informações editora, páginas e autor.

Essa caixa de diálogo deve receber mais informações com a evolução da aplicação. Caso o usuário queira fechar a caixa de diálogo, ele poderá clicar em qualquer área fora da caixa de diálogo ou clicar no botão “CANCELAR” para que a caixa de diálogo seja fechada. Caso o usuário deseje editar alguma das informações visualizadas nessa caixa de diálogo, ele poderá clicar no botão “EDITAR” e os campos com as informações serão habilitados como campos de texto para modificar as informações título, editora e páginas, no caso do campo autor aparecerá o seletor com as opções de autores que estão cadastrados na aplicação, como pode ser visto na figura 5.21. A caixa de diálogo de edição fica com os campos preenchidos com as informações vistas na parte de detalhes do livro, como pode ser visto na figura 5.21.

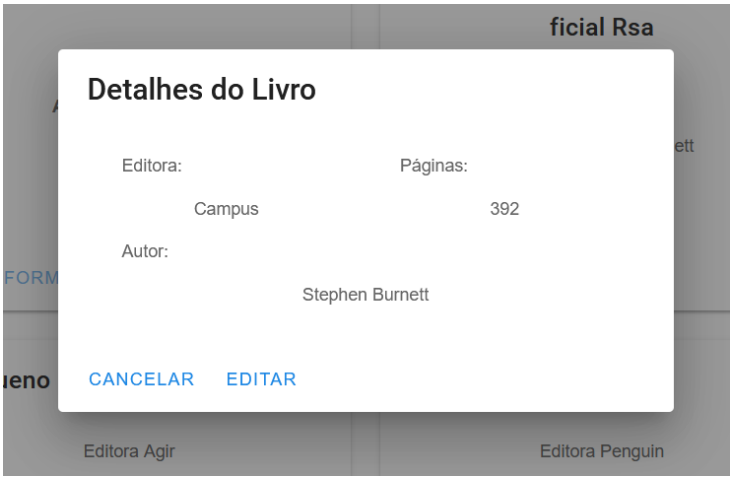


Figura 5.20: Caixa de diálogo para visualização das informações de um livro.

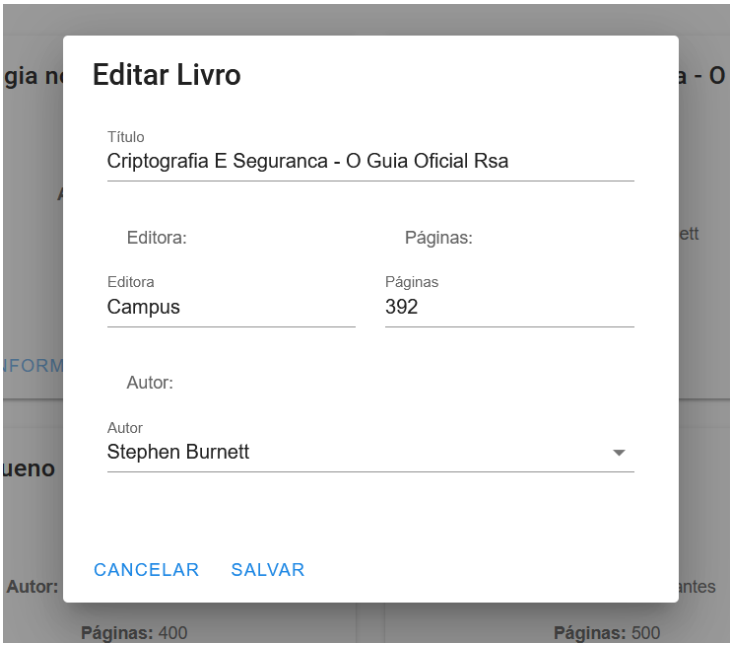


Figura 5.21: Caixa de diálogo para editar informações de um livro.

Parte III

Considerações finais

Capítulo 6

Considerações finais

Neste capítulo, serão apresentados os objetivos alcançados no trabalho. Ele é composto por três seções. A primeira discute o alcance dos objetivos. A segunda seção aponta algumas limitações encontradas ao longo do estudo. Por fim, a terceira seção propõe sugestões para trabalhos futuros sobre o tema.

6.1 Alcance dos objetivos

O objetivo geral deste trabalho era estabelecer uma metodologia ágil para o processo de desenvolvimento de *software*, incorporando atividades e artefatos do processo de gerenciamento de testes de *software*. Os objetivos específicos incluíam a descrição de conceitos relacionados a processo de desenvolvimento de *software* e processo de gerenciamento de teste de *software*, além de configurar o *framework* da metodologia *Personal Extreme Programming* (PXP) para incorporar as atividades e artefatos do gerenciamento de testes, e desenvolver uma aplicação *web* utilizando o processo configurado com o processo de gerenciamento de testes.

O trabalho inicia com a parte teórica, na qual são incluídos capítulos conceituais para definir termos e conceitos importantes para a compreensão do contexto do projeto e das atividades realizadas durante a parte prática. Os processos e métodos utilizados na parte prática também são descritos.

Na parte prática, é inicialmente descrita a configuração do processo de desenvolvimento resultante da incorporação das atividades e artefatos do gerenciamento de testes no PXP. Posteriormente, é realizada a aplicação do processo de desenvolvimento configurado no desenvolvimento de uma aplicação *web*, demonstrando assim sua utilização.

Os objetivos propostos foram alcançados, uma vez que o processo foi configurado de acordo com os critérios definidos e parte de uma aplicação *web* foi desenvolvida seguindo a abordagem proposta.

6.2 Conclusões

A integração do processo de gerenciamento de testes, por meio da abordagem *Kungfu Testing*, no desenvolvimento do *software* PXP, gerou diversas reflexões sobre as práticas adotadas e os desafios encontrados. Dentre as vantagens, é possível perceber a melhoria na prática de teste de *software*. A abordagem do *Kungfu Testing* incentiva o desenvolvedor a adotar melhores práticas para identificar falhas precocemente e analisar partes críticas do *software*, contribuindo para a entrega de um produto mais confiável.

Adicionalmente, a adoção dessa metodologia resultou em um aprimoramento significativo no planejamento e nos critérios de realização dos testes. Diretrizes mais claras e abordagens estruturadas foram implementadas, melhorando o processo de teste da metodologia ágil utilizada.

Outro benefício foi a capacidade de coletar informações para a melhoria contínua do processo de teste. O gerenciamento de testes permitiu a elaboração de artefatos que identificaram áreas de aprimoramento, fornecendo dados para ajustes contínuos e otimização dos processos de teste.

No entanto, é importante considerar as desvantagens associadas a essa integração. Houve um aumento perceptível nas atividades a serem realizadas durante o desenvolvimento do *software*. O tempo e os recursos adicionais necessários para implementar o *Kungfu Testing* impactaram o cronograma do projeto, exigindo uma gestão cuidadosa para mitigar possíveis atrasos.

Outra desvantagem foi o aumento na quantidade de artefatos a serem produzidos. A necessidade de documentação detalhada, embora benéfica para o gerenciamento de testes, gerou uma carga de trabalho adicional, impactando os recursos disponíveis.

Entre as dificuldades encontradas, é importante ressaltar a escassez de informações disponíveis para a elaboração dos artefatos associados ao *Kungfu Testing*. Essa situação resultou na necessidade de recorrer a modelos propostos pelo IEEE para a sua elaboração.

6.3 Limitações do trabalho

O trabalho teve como uma de suas limitações a aplicação desenvolvida para exemplificar a execução do processo de desenvolvimento configurado, que é uma aplicação de pequena escala, pois uma aplicação de grande escala possibilitaria explorar mais problemas que ocorrem no dia a dia do desenvolvimento de software.

Outro fator limitante desse trabalho é o tempo para a realização da aplicação do processo de desenvolvimento configurado, dado que, com mais tempo seria possível a

realização de mais iterações no processo de desenvolvimento, assim, aproximando mais das condições de um projeto de uma aplicação real.

6.4 Sugestões para trabalhos futuros

Uma sugestão para trabalhos futuros é a configuração de outras metodologias ágeis. Essa pesquisa poderia explorar como as atividades e artefatos de gerenciamento de testes, abordados neste trabalho concentrado no PXP, pode ser integrados de maneira eficaz em outras metodologias ágeis.

Outra sugestão relevante é a investigação de Testes em Ambientes Distribuídos. Este campo de pesquisa poderia abordar como o gerenciamento de testes pode ser otimizado e adaptado para equipes distribuídas. Isso incluiria a exploração de ferramentas e técnicas específicas para facilitar a comunicação e colaboração em testes realizados em ambientes distribuídos, proporcionando uma compreensão mais aprofundada dos desafios e soluções nesse cenário.

Adicionalmente, uma linha de pesquisa interessante seria a avaliação da execução de um processo ágil configurado para incorporar as atividades e artefatos de gerenciamento de testes em uma aplicação de médio ou grande porte. Esta avaliação poderia oferecer *insights* sobre a eficácia prática dessa integração em contextos mais complexos, proporcionando uma compreensão mais aprofundada dos benefícios e desafios enfrentados em cenários de desenvolvimento de maior escala.

Referências

[1] Arcos-Medina, Gloria e David Mauricio: *Aspects of software quality applied to the process of agile software development: a systematic literature review*. International Journal of System Assurance Engineering and Management, 10(5):867–897, outubro 2019, ISSN 0976-4348. `https://doi.org/10.1007/s13198-019-00840-7`, acesso em 2023-05-07. 1

[2] Awad, MA: *A comparison between agile and traditional software development methodologies*. University of Western Australia, 30:1–69, 2005. 1

[3] Rajasekaran, Vijayanand: *Issues in Scrum Agile Development Principles and Practices in software development*. Indian Journal of Science and Technology, 8, dezembro 2015. 1

[4] Pohjoisvirta, Lassi: *Choosing a tool for improved software test management*. Tese de Mestrado, Tampere University of Technology, 2013. `https://trepo.tuni.fi/handle/123456789/21872`, acesso em 2023-05-10, Accepted: 2013-11-20T13:02:48Z. 2, 25

[5] Baumgartner, Manfred, Martin Klonk, Christian Mastnak, Helmut Pichler, Richard Seidl e Siegfried Tanczos: *Agile Testing: The Agile Way to Quality*. Springer Nature, setembro 2021, ISBN 978-3-030-73209-7. Google-Books-ID: omNCEAAAQBAJ. 2

[6] IEEE: *ISO/IEC/IEEE International Standard - Software and systems engineering —Software testing —Part 2:Test processes*. ISO/IEC/IEEE 29119-2:2013(E), páginas 1–68, setembro 2013. Conference Name: ISO/IEC/IEEE 29119-2:2013(E). 3, 25

[7] Lucca, Giuseppe A. Di e Anna Rita Fasolino: *Testing Web-based applications: The state of the art and future trends*. Information and Software Technology, 48(12):1172–1186, 2006, ISSN 0950-5849. `https://www.sciencedirect.com/science/article/pii/S0950584906000851`. 3

[8] IEEE: *ISO/IEC/IEEE International Standard - Systems and software engineering–Vocabulary*. ISO/IEC/IEEE 24765:2017(E), páginas 1–541, agosto 2017. Conference Name: ISO/IEC/IEEE 24765:2017(E). 6

[9] IEEE: *IEEE Standard for Configuration Management in Systems and Software Engineering*. IEEE Std 828-2012 (Revision of IEEE Std 828-2005), páginas 1–71, março 2012. Conference Name: IEEE Std 828-2012 (Revision of IEEE Std 828-2005). 6

[10] Pressman, Roger S.: *Software Engineering: A Practitioner's Approach*. McGraw-Hill Education, 2010, ISBN 978-0-07-337597-7. https://www.mlsu.ac.in/econtents/16_EBOOK-7th_ed_software_engineering_a_practitioners_approach_by_roger_s._pressman_.pdf, Google-Books-ID: y4k_AQAAIAAJ. 7, 10, 16

[11] Sommerville, Ian: *Engenharia de software*. Pearson, 9ª edição, janeiro 2012, ISBN 978-85-7936-108-1. 8, 10, 11, 14, 15, 16, 17, 18, 19, 20, 22

[12] Torrecilla-Salinas, C. J., J. Sedeño, M. J. Escalona e M. Mejías: *Agile, Web Engineering and Capability Maturity Model Integration: A systematic literature review*. Information and Software Technology, 71:92–107, março 2016, ISSN 0950-5849. https://www.sciencedirect.com/science/article/pii/S095058491500186X, acesso em 2023-06-18. 11

[13] Abrahamsson, Pekka, Outi Salo, Jussi Ronkainen e Juhani Warsta: *Agile Software Development Methods: Review and Analysis*, setembro 2017. http://arxiv.org/abs/1709.08439, acesso em 2023-06-25, arXiv:1709.08439 [cs]. 13

[14] Baham, Corey: *Improving Business Product Owner Commitment in Student Scrum Projects*. Journal of Information Technology Education: Research, 19:243–258, janeiro 2020. 14

[15] Duarte, Katia e Ricardo Falbo: *Uma Ontologia de Qualidade de Software*. Repositorio Institucional de Universidade Federal do Espírito Santo, janeiro 2000. 17, 18

[16] Ravichandran, Dr: *QA PROCESS, METRICS AND FACTORS AFFECTING THE SOFTWARE QUALITY TO ENHANCE THE CUSTOMER SATISFACTION LEVEL*. International Journal of Computer Engineering and Technology, 3:667–674, agosto 2012. 17, 19

[17] Kan, Stephen H.: *Metrics and Models in Software Quality Engineering*. Addison-Wesley Professional, 2003, ISBN 978-0-201-72915-3. 18

[18] IEEE: *IEEE Standard Glossary of Software Engineering Terminology*. IEEE Std 610.12-1990, páginas 1–84, dezembro 1990. Conference Name: IEEE Std 610.12-1990. 19, 21

[19] Clapp, Judith A., Saul F. Stanten, W. W. Peng, D. R. Wallace, Deborah A. Cerino e Roger J. Dziegiel Jr: *Software Quality Control, Error, Analysis*. William Andrew, 1995, ISBN 978-0-8155-1363-6. 20, 21

[20] IEEE: *ISO/IEC/IEEE International Standard - Systems and software engineering – Vocabulary*. ISO/IEC/IEEE 24765:2010(E), páginas 1–418, dezembro 2010. Conference Name: ISO/IEC/IEEE 24765:2010(E). 20

[21] Society, IEEE Computer: *Software Engineering Body of Knowledge*. IEEE Computer Society, 2014, ISBN 978-0-7695-5166-1. Google-Books-ID: 1YMPjwEACAAJ. 21, 22

[22] Karabašević, Darjan M., Mlađan V. Maksimović, Dragiša M. Stanujkić, Goran B. Jocić e Dušan P. Rajčević: *Selection of software testing method by using ARAS method*. Tehnika, 73(5):724–729, 2018, ISSN 0040-2176. https://scindeks.ceon.rs/Article.aspx?artid=0040-21761805724K&lang=en, acesso em 2022-11-24. 22

[23] IEEE: *IEEE Guide for Software Verification and Validation Plans*. IEEE Std 1059-1993, páginas 1–87, abril 1994. Conference Name: IEEE Std 1059-1993. 23

[24] Ahamed, S. S. Riaz: *Studying the Feasibility and Importance of Software Testing: An Analysis*. International Journal of Engineering Science and Technology, janeiro 2010. https://arxiv.org/abs/1001.4193v1, acesso em 2022-12-15. 23

[25] Parveen, Tauhida, Scott Tilley e George Gonzalez: *A case study in test management*. Proceedings of the 45th annual southeast regional conference, 2007. https://www.academia.edu/9927512/A_case_study_in_test_management, acesso em 2023-06-04. 24

[26] IEEE: *IEEE Standard for Software and System Test Documentation*. IEEE Std 829-2008, páginas 1–150, julho 2008. Conference Name: IEEE Std 829-2008. 24

[27] Iyawa, Gloria Ejehiohen: *Personal Extreme Programming: Exploring Developers' Adoption*. Em Anderson, Bonnie Brinton, Jason Thatcher, Rayman D. Meservy, Kathy Chudoba, Kelly J. Fadel e Sue Brown (editores): *26th Americas Conference on Information Systems, AMCIS 2020, Virtual Conference, August 15-17, 2020*. Association for Information Systems, 2020. https://aisel.aisnet.org/amcis2020/it_project_mgmt/it_project_mgmt/1. 28, 29, 40

[28] Dzhurov, Yani, Iva Krasteva e Sylvia Ilieva: *Personal Extreme Programming–An Agile Process for Autonomous Developers*. Proceedings of the International Conference on Software, Services & Semantic Technologies, janeiro 2009. 28, 29, 30, 31, 32

[29] Agarwal, Ravikant e David Umphress: *Extreme programming for a single person team*. Em *Proceedings of the 46th Annual Southeast Regional Conference on XX*, ACM-SE 46, páginas 82–87, New York, NY, USA, março 2008. Association for Computing Machinery, ISBN 978-1-60558-105-7. https://doi.org/10.1145/1593105.1593127, acesso em 2023-07-26. 29

[30] Moyo, Sibonile: *A software development methodology for solo software developers: leveraging the product quality of independent developers*. Thesis, University of South Africa, fevereiro 2020. https://uir.unisa.ac.za/handle/10500/27292, acesso em 2023-08-16, Accepted: 2021-05-04T13:12:29Z. 29, 30, 31, 32

[31] Asri, S. A., I. G. A. M. Sunaya, E. Rudiastari e W. Setiawan: *Web Based Information System for Job Training Activities Using Personal Extreme Programming (PXP)*. Journal of Physics: Conference Series, 953(1):012092, janeiro 2018, ISSN 1742-6596. https://dx.doi.org/10.1088/1742-6596/953/1/012092, acesso em 2023-08-23, Publisher: IOP Publishing. 31

[32] Marthasari, Gita, Wildan Suharso e Frendy Ardiansyah Ardiansyah: *Personal Extreme Programming with MoSCoW Prioritization for Developing Library Information System*. Proceeding of the Electrical Engineering Computer Science and Informatics, 5(5):537–541, novembro 2018, ISSN 2407-439X, 2407-439X. http://journal.portalgaruda.org/index.php/EECSI/article/view/1701, acesso em 2023-08-15. 32

[33] Pawlak, M. e A. Poniszewska-Marańda: *Software test management approach for agile environments*. Information Systems in Management, Vol. 7, No. 1, 2018, ISSN 2084-5537. http://yadda.icm.edu.pl/baztech/element/bwmeta1.element.baztech-9c34c96f-c9d0-4586-8e27-2175cb057c0e, acesso em 2023-07-15. 33, 34, 35, 36, 37, 39, 40

[34] Pawlak, Michał e Aneta Poniszewska-Marańda: *Software Testing Management Process for Agile Approach Projects*. Em Kryvinska, Natalia e Michal Greguš (editores): *Data-Centric Business and Applications: Evolvements in Business Information Processing and Management (Volume 2)*, Lecture Notes on Data Engineering and Communications Technologies, páginas 63–84. Springer International Publishing, Cham, 2020, ISBN 978-3-030-19069-9. https://doi.org/10.1007/978-3-030-19069-9_3, acesso em 2023-12-20. 34

[35] Veenendaal, Erik van: *Practical Risk-Based Testing Product Risk Management: The PRISMA Method*. 2011. https://www.ctqb.org/files/content/ctqb/downloads/others/papers/e-book-PRISMA.pdf. 37

[36] IEEE: *IEEE/ISO/IEC International Standard for Software and systems engineering–Software testing–Part 3:Test documentation*. ISO/IEC/IEEE 29119-3:2021(E), páginas 1–98, outubro 2021. Conference Name: ISO/IEC/IEEE 29119-3:2021(E). 38, 39, 40

[37] Github: *Issues*, 2023. https://docs.github.com/en/issues/tracking-your-work-with-issues/about-issues, acesso em 2023-09-01. 39