

Projeto Integrador

Plano de Testes

Versão 1.2

Esse documento refere-se ao plano de testes da Equipe 5 do Projeto Integrador. Pretende descrever como a equipe de projeto testará o que foi implementado e quais são os testes a serem realizados. Ele define cada caso de teste, seus padrões, as datas dos testes e como foram aplicados, visando assegurar a qualidade do produto final e de suas partes, garantindo que os requisitos e padrões de qualidade sejam atendidos.

Autor(es): Aroldo Augusto Barbosa Simões.

Data(última modificação): 06/11/2023

Histórico da Revisão

Data	Versão	Descrição	Autor
24/10/2023	<0.1>	Primeira versão.	Aroldo Augusto Barbosa Simões
26/10/2023	<1.0>	Primeira versão estável	Aroldo Augusto Barbosa Simões
29/10/2023	<1.1>	Correções pontuais	Aroldo Augusto Barbosa Simões
06/11/2023	<1.2>	Correções em relação ao relatório e ao plano de testes	Aroldo Augusto Barbosa Simões

Sumário

Projeto Integrador	1
Plano de Testes	1
Histórico da Revisão	2
Sumário	3
Introdução	4
Code review	5
Tipos de testes aplicados	7
Testes Unitários	7
Testes funcionais	7
Teste de usabilidade	8
Planos e relatórios de testes	10
Referências	11

Introdução

Um Plano de Teste é uma parte fundamental do ciclo de desenvolvimento de software. Também conhecido como Plano de Teste de Software, é um documento formal que descreve como as atividades de teste de software serão conduzidas em um projeto. Ele serve como um guia detalhado para a execução de testes de software e ajuda a garantir que o software seja testado de maneira abrangente e rigorosa. No caso deste, trata-se de um plano de teste geral para o projeto como um todo. Os testes que serão feitos estão abaixo descritos, bem como seus objetivos e datas de aplicação ou *sprint* na qual será realizado.

Além de encontrar falhas, testes objetivam aumentar a confiabilidade de um sistema de software, isto é, aumentar a probabilidade de que um sistema continue funcionando sem falhas durante um período de tempo. Embora seja desejável testar um sistema por completo, deve-se ter em mente que a atividade de teste assegura apenas encontrar falhas se elas existirem, mas não asseguram sua ausência. Portanto, as atividades devem ser disciplinadas a fim de identificar a maioria dos erros existentes.

Ademais, é importante ressaltar uma importante tarefa a ser realizada pela equipe: *code review*. *Code review*, ou revisão de código, é um processo no desenvolvimento de software no qual membros da equipe revisam o código fonte escrito por seus colegas para identificar erros, melhorias e conformidade com padrões de codificação. É uma prática fundamental para garantir a qualidade do código e do software em geral. É uma etapa importante do processo de desenvolvimento e será melhor abordada no tópico adequado abaixo.

Code review

Essa tarefa será realizada principalmente pelo analista de qualidade do projeto. A revisão de código é importante pois ajuda a identificar e corrigir problemas no código, como *bugs*, vulnerabilidades de segurança, e problemas de desempenho, além de promover a colaboração e o aprendizado na equipe, à medida que os membros compartilham conhecimentos e experiências. Garante, também, que o código esteja em conformidade com padrões de codificação e diretrizes da equipe e ajuda a manter um código limpo e legível.

Dentre os benefícios pode-se destacar a melhora a qualidade do código e, por consequência, a qualidade do software, o aumento da segurança(identificando potenciais vulnerabilidades), a facilitação da identificação de problemas de desempenho e ineficiências, a prevenção de bugs e problemas de manutenção no futuro e a promoção do compartilhamento de conhecimento e as melhores práticas dentro da equipe.

Para aplicação dessa atividade, foi necessário definir o foco da análise, a sua objetivação e padronização. Para isso, o seguinte *checklist* foi utilizado. O documento “Padronizações de código - Java” foi o parâmetro a ser seguido para a análise e revisão de código.

“ Arquivo: Exemplo.java

- Classe “Exemplo”:

- ☐ **Pacotes e importações:**

- ☐ Uso de importações explícitas;
- ☐ Importações agrupadas por pacote e em ordem alfabética;
- ☐ Somente importações utilizadas;
- ☐ Nomes de importações adequados.

- ☐ **Indentação, formatação e espaçamento:**

- ☐ Uso de ‘*tab*’ para indentação;
- ☐ Uso de chaves para todos os blocos;
- ☐ Uso de espaços em branco em torno de operadores e após vírgulas;
- ☐ Sem espaços entre parênteses e nome de métodos;
- ☐ Abertura de chaves na mesma linha da declaração e fechamento alinhado;
- ☐ Métodos separados por linha em branco;
- ☐ Evitar linhas muito longas(mais de 80 caracteres);
- ☐ Linhas em branco após comentários JavaDoc.

- ☐ **Comentários e JavaDoc:**

- ☐ Uso extensivo de comentários;
- ☐ Comentários claros e concisos;
- ☐ JavaDoc arquivo;
- ☐ JavaDoc classes;
- ☐ JavaDoc métodos.

☐ **Nomenclatura:**

- ☐ Métodos;
- ☐ Classes;
- ☐ Variáveis;
- ☐ Constantes;
- ☐ Pacotes.

☐ **Composição de classes:**

- ☐ Documentação;
- ☐ Declaração;
- ☐ Variáveis;
- ☐ Instâncias;
- ☐ Construtores;
- ☐ Métodos. “

Tipos de testes aplicados

Todos os testes e análises de implementações serão realizados, majoritariamente, pelo analista de qualidade do projeto. Desenvolvedores juniores definidos pela equipe também participarão, quando requisitados, assim como o desenvolvedor sênior e/ou o desenvolvedor líder. Os testes serão aplicados nas máquinas pessoais dos stakeholders responsáveis. As IDEs escolhidas para aplicação dos testes foram o IntelliJ e o Visual Studio Code. Todos os testes aplicados serão caixa preta, já que não objetivam análise e verificação do funcionamento das linhas de código criadas.

Checklists para verificação das linhas de código produzidas serão aplicados. Cada classe deve estar de acordo com os padrões de codificação definidos, bem como com os subprogramas e demais implementações.

A cada *sprint* um documento “Plano de testes - Sprint x” deverá ser criado para especificação dos testes e dos respectivos casos de teste aplicados. Assim como a criação de um “Relatório de testes - Sprint x”, contendo a descrição de *bugs* e o *code review*, bem como os resultados dos testes aplicados.

Testes Unitários

O objetivo de um teste unitário é verificar se unidades individuais de código, como métodos ou funções, funcionam conforme o esperado. Um teste unitário tem como foco isolar e testar pequenas partes do código de maneira independente para garantir que elas produzam os resultados desejados. Isso ajuda a identificar e corrigir erros ou defeitos no código de forma precoce, facilitando a manutenção e melhorando a qualidade do software como um todo. Os testes unitários são uma prática fundamental em desenvolvimento de software e fazem parte das metodologias ágeis, como é o caso desse projeto.

Os teste unitários serão aplicados em todas as *sprints*, ou seja, cada *sprint* vai possuir casos de testes unitários específicos, tanto para implementações dos *devs Seniors*, quanto para os *devs juniors*.

A biblioteca a ser utilizada para aplicação dos testes unitários será a JUnit. A JUnit é um framework de código aberto amplamente utilizado para escrever e executar testes unitários em Java. Ela fornece um ambiente que permite aos desenvolvedores criar testes para suas classes e métodos, verificar se o código produz os resultados esperados e automatizar o processo de execução dos testes.

Testes funcionais

São uma categoria de testes de software que se concentram em avaliar se um sistema ou um componente de software está funcionando conforme as especificações e requisitos funcionais. Eles se concentram em verificar se as funções ou recursos do software atendem às expectativas do usuário. O principal objetivo dos testes funcionais é garantir que o software funcione como deveria do ponto de vista do usuário final.

São caracterizados pela presença de cenários de uso do software, ou seja, tentam simular situações do mundo real em que o software será usado para garantir que ele funcione corretamente em situações reais.

Os teste funcionais serão aplicados em todas as *sprints*, ou seja, cada *sprint* vai possuir um documento que contenha os teste funcionais realizados, além dos testes unitários. Somente serão realizados testes funcionais para o conjunto de implementações dos *devs* (seniores e juniores).

Passo a passo para criação de teste funcional:

- **Identifique os Casos de Uso:** Comece identificando os casos de uso do sistema. Os casos de uso descrevem interações específicas entre os usuários e o sistema. Certifique-se de entender completamente cada caso de uso.
- **Crie Cenários de Teste:** Com base nos casos de uso, crie cenários de teste que refletem as interações dos usuários com o sistema. Isso pode incluir a descrição das etapas que um usuário seguiria para realizar uma determinada tarefa.
- **Defina Dados de Teste:** Prepare os dados necessários para cada cenário de teste. Isso pode incluir entradas, como formulários preenchidos, e resultados esperados.
- **Desenvolva os Testes:** Escreva scripts de teste Java que executem os cenários de teste. Isso pode envolver a automação de interações com a interface do usuário (por exemplo, por meio de ferramentas como o Selenium) ou chamadas diretas às APIs do sistema.
- **Execute os Testes:** Execute os testes, observando se o sistema se comporta conforme o esperado em cada cenário de teste. Registre os resultados, identificando quaisquer problemas ou discrepâncias.
- **Relate Problemas:** Se um cenário de teste revelar problemas ou falhas no sistema, registre essas questões em um sistema de rastreamento de problemas ou em uma ferramenta de gerenciamento de testes.
- **Itere e Repita:** À medida que os problemas são corrigidos, repita os testes para garantir que as correções não afetam outras partes do sistema.
- **Automatize se Possível:** Se você tiver muitos casos de uso e cenários de teste, considere a automação dos testes funcionais. Isso pode economizar tempo e garantir que os testes sejam executados de maneira consistente.
- **Integre com seu Ciclo de Desenvolvimento:** Integre os testes funcionais com seu ciclo de desenvolvimento, garantindo que eles sejam executados regularmente, especialmente após alterações no sistema.

Teste de usabilidade

Testes de usabilidade são uma técnica de avaliação que envolve observar usuários reais enquanto eles interagem com um sistema de software para identificar problemas de usabilidade. Os testes de usabilidade têm o objetivo de garantir que um sistema seja fácil de usar, eficaz e forneça uma experiência positiva ao usuário. Esses testes serão aplicados em laboratório no campus da universidade e os participantes serão alunos da graduação. As máquinas utilizadas também serão as presentes no respectivo laboratório.

Para realização destes testes os seguintes passos devem ser realizados:



- Defina Objetivos: Identifique os objetivos do teste de usabilidade. O que você deseja avaliar? Quais são as principais áreas de preocupação em relação à usabilidade?
- Recrute Participantes: Selecione um grupo de participantes representativos do público-alvo do sistema. O tamanho do grupo pode variar, mas geralmente envolve de 5 a 10 participantes.
- Desenvolva Cenários e Tarefas: Crie cenários e tarefas que os participantes devem realizar durante o teste. Essas tarefas devem refletir ações típicas que os usuários realizariam no sistema.
- Conduza os Testes: Realize as sessões de teste com os participantes, observando suas interações com o sistema. Registre observações e colete feedback dos participantes.
- Analise os Resultados: Após a conclusão dos testes, analise os dados coletados. Identifique problemas de usabilidade e avalie o quão bem o sistema atendeu aos objetivos do teste.
- Documente Resultados: Prepare um relatório que descreva os problemas encontrados, suas gravidades e recomendações para melhorias.

Planos e relatórios de testes

Em cada *sprint* um relatório e um plano de testes deverão ser criados, a partir dos "Template Relatório de testes - Sprint x" e "Template Plano de testes - Sprint x", respectivamente. Cada *sprint* diferente possuirá um diferente relatório e um diferente plano de teste, contendo os detalhes, aplicações e os casos de teste.

Por exemplo, na *sprint* 1 as implementações de *devs juniors* e de *devs seniors* deverão ser testadas de acordo com o(s) caso(s) de uso aplicados, portanto, no início da *sprint* será criado um plano de teste a ser aplicado, assim como ao final da *sprint* um relatório deverá ser entregue. Dessa forma, cada implementação e cada etapa do desenvolvimento será testada e documentada individualmente.

Referências

- [1] Plano de Teste - Um Mapa Essencial para Teste de Software. Pode ser acessado em: <<https://www.devmedia.com.br/plano-de-teste-um-mapa-essencial-para-teste-de-software/13824>>. Último acesso em: 26/10/2023.
- [2] ChatGPT - Referências citadas: "Documentação do JUnit: <https://junit.org/junit5/docs/current/user-guide/>. Beck, K. (2003)"; "Test-Driven Development: By Example. Addison-Wesley. Freeman, S., & Pryce, N. (2009)"; "Growing Object-Oriented Software, Guided by Tests. Addison-Wesley"; "Best Practices for Peer Code Review" (SmartBear); "Code Review Best Practices" (Atlassian); "The Ultimate Guide to Code Review" (GitPrime). Pode ser acessado em: <<https://chat.openai.com>>. Último acesso em: 06/11/2023.
- [3] Template Plano de Testes. Pode ser acessado em: <  Plano de Teste >. Último acesso em: 26/10/2023.
- [4] Template Plano de Testes: Teach2Learn. Pode ser acessado em: <  Modelo_Plano_Testes >. Último acesso em: 26/10/2023.
- [5] JUnit. Pode ser acessado em: <<https://junit.org/junit5/>>. Último acesso em: 26/10/2023.
- [6] Material didático ESOF 2 - Slides. Pode ser acessado em: <<https://ava.ufv.br/>>. Último acesso em: 26/10/2023.