UniSENAI

Trilha 01

Testes de Software

Instruções para a melhor prática de Estudo

1. **Leia atentamente todo o conteúdo:** Antes de iniciar qualquer atividade, faça uma leitura detalhada do material fornecido na trilha, compreendendo os conceitos e os exemplos apresentados.

2. **Não se limite ao material da trilha:** Utilize o material da trilha como base, mas busque outros materiais de apoio, como livros, artigos acadêmicos, vídeos, e blogs especializados. Isso enriquecerá o entendimento sobre o tema.

3. **Explore a literatura:** Consulte livros e publicações reconhecidas na área, buscando expandir seu conhecimento além do que foi apresentado. A literatura acadêmica oferece uma base sólida para a compreensão de temas complexos.

4. **Realize todas as atividades propostas:** Conclua cada uma das atividades práticas e teóricas, garantindo que você esteja aplicando o conhecimento adquirido de maneira ativa.

5. **Evite o uso de Inteligência Artificial para resolução de atividades:** Utilize suas próprias habilidades e conhecimentos para resolver os exercícios. O aprendizado vem do esforço e da prática.

6. **Participe de debates:** Discuta os conteúdos estudados com professores, colegas e profissionais da área. O debate enriquece o entendimento e permite a troca de diferentes pontos de vista.

7. **Pratique regularmente:** Não deixe as atividades para a última hora. Pratique diariamente e revise o conteúdo com frequência para consolidar o aprendizado.

8. **Peça feedback:** Solicite o retorno dos professores sobre suas atividades e participe de discussões sobre os erros e acertos, utilizando o feedback para aprimorar suas habilidades.

Essas instruções são fundamentais para garantir um aprendizado profundo e eficaz ao longo das trilhas.

1. O que são testes de software

Os testes de software são um processo sistemático utilizado para avaliar a qualidade de um software, garantindo que ele atenda aos requisitos especificados e funcione corretamente em diferentes cenários. O objetivo dos testes é assegurar que o software entregue esteja livre de falhas críticas, promovendo confiabilidade, segurança e eficiência. Para isso, são aplicadas diversas metodologias e abordagens, dependendo do contexto do software e das necessidades do usuário final.

Os testes podem ser realizados manualmente ou de forma automatizada, sendo aplicáveis a diferentes estágios do desenvolvimento do software. Eles ajudam a identificar falhas no código antes que o produto seja liberado ao mercado, reduzindo custos de manutenção e evitando impactos negativos para os usuários finais.

Exemplo: Uma empresa desenvolve um aplicativo de banco digital. Para garantir que ele funcione corretamente, os testadores verificam se as transações são processadas corretamente, se a autenticação de dois fatores funciona e se a interface responde como esperado.

1.1. O que são inspeção de software

A inspeção de software é um processo de revisão manual que busca identificar defeitos em código, documentação e requisitos. É uma técnica estática que ajuda a detectar erros antes que o software seja executado.

Técnicas de Inspeção

- **Revisão por Pares**: Desenvolvedores revisam o código uns dos outros.
- **Walkthrough**: O autor do código apresenta sua implementação para revisão.
- **Inspeção Formal**: Processo estruturado conduzido por um moderador.

---

2. Testes de Software e Quality Assurance (QA)

O Quality Assurance (QA) é um conjunto de processos e metodologias que visa garantir a qualidade do software de maneira preventiva. Diferentemente dos testes de software, que se concentram em identificar falhas específicas, o QA atua de forma ampla, garantindo que boas práticas de desenvolvimento sejam seguidas desde o início do ciclo de vida do software.

Uma das principais diferenças entre QA e testes é que o QA define padrões e processos para evitar defeitos, enquanto os testes de software verificam se esses padrões foram cumpridos e identificam possíveis erros no produto final. O QA pode incluir práticas como revisões de código, integração contínua e aplicação de metodologias ágeis para manter a qualidade do software.

Exemplo: No desenvolvimento de um sistema de e-commerce, a equipe de QA estabelece diretrizes para evitar falhas, como definir critérios para a aceitação de código e a necessidade de testes automatizados para pagamentos antes da liberação para produção.

---

3. Garantia de Qualidade (QA) e Controle de Qualidade (QC)

Embora estejam relacionados, QA e QC (Controle de Qualidade) possuem diferenças fundamentais:

- **Garantia de Qualidade (QA):** Processo preventivo que visa garantir a conformidade do software com os padrões de qualidade estabelecidos. Atua nos processos antes da entrega do produto.
- **Controle de Qualidade (QC):** Processo corretivo focado na detecção e resolução de defeitos no produto final por meio da aplicação de testes e validações.

O QA atua desde a concepção do projeto até sua manutenção, garantindo que boas práticas sejam seguidas. Já o QC entra em ação no momento de verificar se o software funciona conforme o esperado, realizando testes para validar funcionalidades e identificar falhas.

Exemplo: Em uma empresa de software, o QA define padrões de codificação e práticas ágeis para prevenir defeitos, enquanto o QC executa testes manuais e automatizados para validar que o software atende aos requisitos especificados.

---

4. Erro, Defeito e Falha

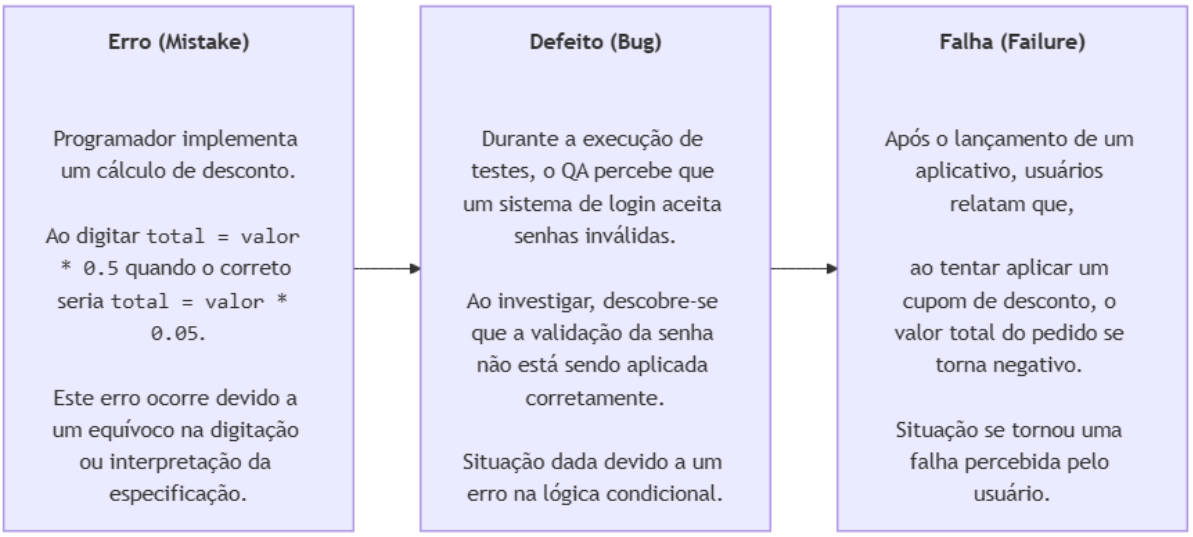
No contexto de Testes de Software e Qualidade (QA), é essencial compreender a diferença entre erro, defeito e falha. Esses termos estão diretamente relacionados à garantia da qualidade do software e à sua confiabilidade.

- **Erro (Mistake):** Uma ação equivocada cometida pelo desenvolvedor ao programar, seja por descuido, falta de conhecimento ou interpretação errada dos requisitos.
- **Defeito (Bug):** Um erro que se manifesta no código e pode comprometer o funcionamento correto do software.
- **Falha (Failure):** Quando um defeito chega à execução do software e gera um comportamento inesperado ou incorreto, afetando o usuário final.

Dessa forma, os erros podem ser cometidos na fase de desenvolvimento, gerando defeitos no código-fonte, que podem levar a falhas perceptíveis na utilização do software pelo usuário final. A detecção precoce de erros reduz custos de correção e melhora a confiabilidade do sistema.

Exemplo: Um desenvolvedor esquece de validar a entrada de dados em um formulário. Isso gera um erro no código, que passa despercebido e se torna um defeito no sistema. Quando um usuário insere um valor inválido, o sistema trava, causando uma falha.

Exemplo visual:



---

5. O que é BDD e Gherkin

- **BDD (Behavior Driven Development):** É uma abordagem de desenvolvimento de software baseada em comportamento, onde os requisitos do sistema são descritos em linguagem natural, facilitando o entendimento entre equipes técnicas e não técnicas.
- **Gherkin:** Linguagem estruturada utilizada no BDD para descrever cenários de testes de maneira compreensível para todas as partes envolvidas no desenvolvimento do software.

O BDD busca melhorar a colaboração entre desenvolvedores, testadores e stakeholders, garantindo que o software seja construído de acordo com as expectativas do cliente. A sintaxe do Gherkin permite a escrita de testes de aceitação que podem ser automatizados, auxiliando no processo de validação do software.

BDD utiliza **linguagens específicas de domínio (DSL)**, como Gherkin, para escrever cenários de teste em um formato legível e acessível a todos os envolvidos no projeto.

Estrutura de Cenários BDD

Os cenários BDD são escritos no formato **Gherkin**, que utiliza as palavras-chave:

- **Feature (Funcionalidade):** Descreve o recurso a ser implementado.
- **Scenario (Cenário):** Define um caso de uso específico.
- **Given (Dado):** Estabelece o contexto inicial.
- **When (Quando):** Define a ação realizada.
- **Then (Então):** Explica o resultado esperado.

Exemplo: Para garantir que um sistema de reservas de hotéis funcione corretamente, um analista de QA escreve o seguinte cenário em Gherkin:

```
Scenario: Reserva de hotel bem-sucedida
  Given que o usuário está na página de reservas
  When ele insere as datas e confirma a reserva
  Then ele deve receber uma mensagem de confirmação
```

---

6. Principais tipos de testes

Existem diversos tipos de testes de software, cada um com um propósito específico dentro do ciclo de vida do desenvolvimento:

- **Testes Unitários:** Avaliam o funcionamento de partes individuais do código, como funções e classes.
- **Testes de Integração:** Verificar se diferentes componentes ou módulos de um sistema funcionam corretamente quando integrados. É um passo além do teste unitário, onde se testam interações entre partes do sistema.
- **Testes de Sistema:** Avaliam o software como um todo, verificando se atende aos requisitos definidos.
- **Testes de Regressão:** Verificar se novas alterações no sistema não quebraram funcionalidades já existentes. Esses testes garantem que os bugs corrigidos e os novos recursos não impactem negativamente o sistema.
- **Testes de Aceitação:** Garantir que o sistema atende aos requisitos do cliente ou usuário final. Esses testes geralmente são realizados com base em critérios estabelecidos pela equipe de negócios ou stakeholders.
- **Testes de Carga:** Avaliar como o sistema se comporta sob carga, ou seja, com um grande número de usuários simultâneos, para garantir que ele continue funcionando de maneira eficiente.

Cada tipo de teste desempenha um papel crucial na validação da qualidade do software, garantindo que o produto seja confiável e livre de erros críticos.

Exemplo: Em um aplicativo de delivery, testes unitários verificam se a lógica de cálculo do preço está correta, testes de integração garantem que os pedidos são registrados no banco de dados corretamente e testes de regressão validam que novas funcionalidades não quebram recursos já existentes.

---

7. Pirâmide de Testes

A Pirâmide de Testes é um conceito que orienta a melhor distribuição dos testes dentro do ciclo de desenvolvimento do software. Ela visa estabelecer um equilíbrio entre rapidez, eficiência e cobertura de testes, garantindo um fluxo de desenvolvimento mais robusto e confiável. Essa abordagem é representada por uma estrutura piramidal que se divide em três camadas principais:

7.1. Base: Testes Unitários

O que são: Testam componentes ou unidades individuais do código, como funções ou métodos, de forma isolada.

Objetivo: Verificar se cada unidade de código funciona corretamente.

Características:

- Alta velocidade de execução.
- Fácil automação.
- Maior cobertura do código.

Exemplo: Testar se uma função de cálculo de imposto retorna o valor correto para diferentes entradas.

Quantidade: Deve ser a camada mais ampla da pirâmide, com muitos testes unitários cobrindo grande parte do código.

7.2. Camada do Meio: Testes de Integração

O que são: Verificam a interação entre diferentes componentes ou módulos do sistema.

Objetivo: Garantir que os componentes funcionem corretamente juntos, incluindo conexões com APIs, bancos de dados ou serviços externos.

Características:

- Média velocidade de execução.
- Foco na comunicação e nos dados trocados entre partes do sistema.

Exemplo: Testar a integração entre um sistema de login e um banco de dados.

Quantidade: Deve haver uma quantidade moderada de testes de integração.

7.3. Topo: Testes de Interface ou de Aceitação (End-to-End)

O que são: Testam o sistema como um todo, simulando o comportamento do usuário final.

Objetivo: Validar o funcionamento completo do sistema, desde a interface até o back-end.

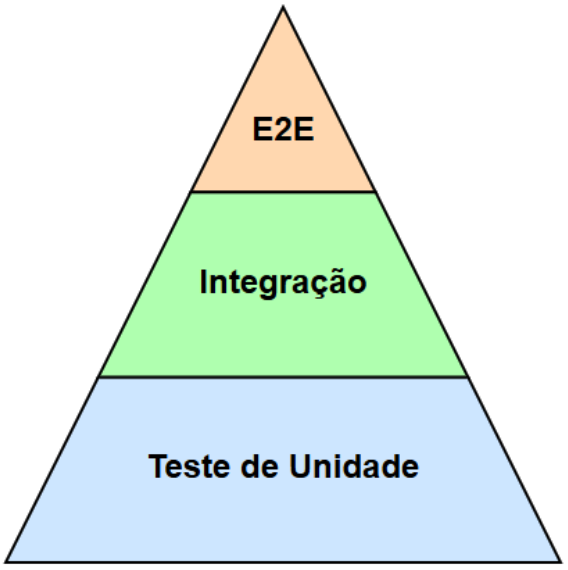
Características:

- Alta complexidade e custo de manutenção.
- Maior tempo de execução.
- Mais suscetíveis a falhas intermitentes.

Exemplo: Verificar se o fluxo de compra em um e-commerce, desde a seleção de produtos até o pagamento, está funcionando corretamente.

Quantidade: Deve haver poucos testes end-to-end, focando nos fluxos mais críticos.

Exemplo visual:



A distribuição equilibrada desses testes é essencial para evitar dependência excessiva de testes end-to-end, que são mais demorados e caros, enquanto garante a cobertura eficiente de todas as partes do software.

Exemplo: Em um aplicativo de redes sociais, a empresa prioriza testes unitários para funções básicas, testes de integração para verificar o funcionamento entre módulos (ex: upload de imagens e notificações) e testes end-to-end para validar a experiência do usuário ao postar conteúdos.

---

8. Ciclo de Vida de Testes

O ciclo de vida dos testes de software segue um conjunto de etapas bem definidas para assegurar que todas as funcionalidades sejam validadas antes da entrega ao cliente. Esse ciclo compreende:

1. **Planejamento e Controle:** Nesta fase, são definidos os objetivos dos testes, os recursos necessários, os cronogramas e os critérios de entrada e saída. O controle ocorre ao longo do processo para garantir que os testes sigam conforme o planejado.
2. **Análise e Design:** Aqui são analisados os requisitos do software para identificar quais cenários precisam ser testados. São criados casos de teste, roteiros e estratégias de teste.
3. **Implementação e Execução:** Nesta fase, os casos de teste são preparados e executados, seja manualmente ou de forma automatizada. Os resultados são registrados para avaliação posterior.
4. **Registro e Relato de Defeitos:** Documentação dos defeitos encontrados e análise de suas causas.

5. **Reexecução e Validação:** Realização de retestes e testes de regressão após a correção dos defeitos.
6. **Encerramento:** Os testes são finalizados, e um relatório final é gerado contendo as lições aprendidas e sugestões para melhorias futuras.

Esse fluxo permite uma abordagem sistemática para garantir qualidade e confiabilidade no software antes de sua liberação para os usuários.

Exemplo: Em um sistema bancário, a equipe de testes planeja os casos de teste, analisa requisitos, executa os testes, documenta os defeitos, executar testes após correções e finaliza o ciclo validando a estabilidade da aplicação antes do lançamento.

---

9. Roteiro de Testes

O roteiro de testes é um documento fundamental para guiar a equipe de testes na execução das validações necessárias. Ele contém informações detalhadas sobre como os testes devem ser conduzidos e quais critérios são usados para avaliar a conformidade do software.

Os principais elementos de um roteiro de testes incluem:

- **Objetivo:** O que se pretende verificar com os testes.
- **Escopo:** Definição das funcionalidades que serão testadas.
- **Critérios de entrada e saída:** Requisitos necessários para iniciar e finalizar os testes.
- **Passos para execução:** Sequência de ações para realizar cada teste.
- **Critérios de aceitação:** Definição do que caracteriza um teste bem-sucedido.

Seguir um roteiro de testes bem estruturado melhora a organização e eficiência dos testes, reduzindo falhas e retrabalho no desenvolvimento.

Exemplo: Para um app de check-in em aeroportos, o roteiro de testes define que os testadores devem validar o login, selecionar um voo, adicionar bagagem e concluir o check-in, verificando se todos os fluxos funcionam corretamente.

---

10. Relatórios de Defeitos

Os relatórios de defeitos documentam as falhas encontradas durante a execução dos testes, fornecendo informações essenciais para a equipe de desenvolvimento corrigir os problemas. Um relatório eficaz deve conter:

- **Identificação do defeito:** Número ou código único para rastreamento.
- **Descrição do problema:** Explicação detalhada da falha observada.
- **Passos para reprodução:** Instruções claras sobre como reproduzir o erro.
- **Severidade e prioridade:** Classificação do impacto da falha no sistema.
- **Evidências:** Capturas de tela, logs de erro ou vídeos demonstrando o problema.

Relatórios bem documentados agilizam o processo de correção e evitam que falhas conhecidas sejam ignoradas, contribuindo para a qualidade do software.

Exemplo: Um testador encontra um bug em um site de compras e preenche um relatório descrevendo o problema: “Ao adicionar um item ao carrinho e tentar removê-lo, o site exibe um erro 500”. O relatório inclui passos para reproduzir, evidências visuais e a prioridade de correção.

---

11. Automação de Testes

A automação de testes consiste na execução de testes por meio de ferramentas especializadas, eliminando a necessidade de testes manuais repetitivos. Essa abordagem traz diversos benefícios:

- **Eficiência:** Redução do tempo de execução dos testes.
- **Maior cobertura:** Possibilita testar um grande número de cenários automaticamente.
- **Consistência:** Reduz a chance de erros humanos ao repetir testes.
- **Rapidez na detecção de falhas:** Identificar rapidamente problemas em ciclos contínuos de desenvolvimento.

As ferramentas mais populares de automação incluem Selenium (para testes web), Appium (para testes móveis), JUnit (para testes unitários em Java) e Cypress (para aplicações front-end). A implementação da automação de testes é essencial para processos ágeis e entrega contínua de software de alta qualidade.

Exemplo: Em uma startup de fintech, testes automatizados com Selenium são usados para validar a interface do usuário, garantindo que botões, formulários e pagamentos funcionem corretamente em diferentes navegadores.

---

12. Device Farms

Device Farms são infraestruturas remotas que permitem a execução de testes em uma ampla variedade de dispositivos físicos e virtuais, garantindo que o software seja compatível com diferentes ambientes. Essas plataformas são essenciais para testar aplicações móveis e web em múltiplas configurações.

As principais vantagens do uso de Device Farms incluem:

- **Acesso a diversos dispositivos:** Possibilita testar o software em vários modelos de smartphones, tablets e desktops.
- **Redução de custos:** Elimina a necessidade de manter uma grande quantidade de dispositivos físicos para testes.
- **Testes em condições reais:** Simular diferentes cenários, como variações de rede, versões de sistemas operacionais e características de hardware.

Plataformas como AWS Device Farm, Firebase Test Lab e Sauce Labs são amplamente utilizadas para validar a usabilidade e estabilidade de aplicativos antes de sua liberação no mercado, garantindo uma experiência consistente para os usuários finais.

Exemplo: Uma empresa de desenvolvimento de aplicativos móveis utiliza AWS Device Farm para testar seu app em diversos dispositivos Android e iOS, garantindo que funcione corretamente em diferentes telas, resoluções e versões do sistema operacional.

---

13. Ciclo PDCA

O Ciclo PDCA (Plan-Do-Check-Act) é uma metodologia utilizada para a melhoria contínua de processos dentro de uma organização. Ele é um ciclo iterativo que ajuda a identificar problemas, implementar soluções, monitorar resultados e realizar ajustes para otimização contínua.

Objetivo do PDCA

O principal objetivo do PDCA é aprimorar processos de forma contínua, garantindo eficiência, qualidade e redução de falhas. Ele é amplamente utilizado em áreas como gestão de qualidade, segurança da informação, manufatura, desenvolvimento de software e TI.

As 4 etapas do Ciclo PDCA

Plan (Planejar)

- Identificar problemas e oportunidades de melhoria.
- Definir metas e estratégias para solução.
- Elaborar um plano de ação.

Do (Executar)

- Implementar as ações planejadas.
- Testar novas abordagens em pequena escala.
- Registrar dados e observações.

Check (Verificar)

- Analisar os resultados obtidos.
- Comparar com os objetivos definidos na fase de planejamento.
- Identificar possíveis ajustes.

Act (Agir)

- Aplicar melhorias com base nos resultados analisados.
- Padronizar processos bem-sucedidos.
- Repetir o ciclo para melhorias contínuas.

Lista de Exercícios de Fixação

Questões Dissertativas

1. Explique o conceito de testes de software e como ele contribui para a entrega de produtos de qualidade.
2. Diferencie Quality Assurance (QA) e Quality Control (QC), dando exemplos práticos da aplicação de cada um.
3. Descreva o ciclo de vida dos testes de software, explicando a importância de cada etapa.
4. Explique a relação entre erro, defeito e falha no desenvolvimento de software e como esses conceitos impactam a experiência do usuário, dando exemplos práticos da aplicação de cada um.
5. O que é Behavior Driven Development (BDD) e como ele pode facilitar a comunicação entre diferentes equipes no desenvolvimento de software?
6. Com base na pirâmide de testes, explique a importância de priorizar testes unitários e testes de integração antes de realizar testes end-to-end.
7. Qual a importância da automação de testes no desenvolvimento de software e quais desafios podem ser encontrados na sua implementação?
8. Como um roteiro de testes pode melhorar a eficiência do processo de teste em um projeto de software?
9. Qual a importância dos relatórios de defeitos no processo de desenvolvimento de software e quais informações essenciais devem conter?
10. Discorra sobre os benefícios do uso de Device Farms para a validação de aplicativos móveis e web, mencionando como isso pode impactar a qualidade do produto final.