**Formação QA Stefanini**

**Fundamentos de Teste de Software**

**Conceitos Básicos**

**O que é e o que faz um testador de software?**

O software tester, termo em inglês para definir testador de software, é o profissional responsável por encontrar erros, falhas, bugs e outros tipos de problemas que não foram detectados durante a confecção (criação) de um software.

**Quais as atividades do Analista de Testes?**

Garantir que as regras de negócio sejam implementadas corretamente;

Ajudar a priorizar os problemas encontrados;

Ajudar a refinar os requisitos (toda a documentação do sistema);

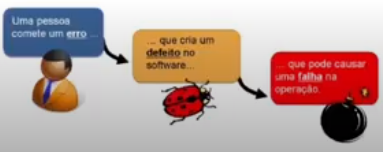
Características Fundamentais: Ser investigativo, ter uma comunicação assertiva e saber trabalhar em grupo.

**Erro x Defeito x Falha**

Erro: é uma ação humana que produz um resultado incorreto;

Defeito: é a manifestação de um erro no software, também conhecido como bug;

Falha: diferença indesejável entre o observado e o esperado (defeito encontrado);



**Quais os tipos de Teste de Software?**

* Teste da caixa branca (Teste Estrutural): Técnica de teste que avalia o comportamento interno do componente de software. Trabalha diretamente sobre o código-fonte do componente de software para avaliar aspectos tais como: teste de condição, teste de fluxo de dados, teste de ciclos e teste de caminhos lógicos. O testador tem acesso ao código fonte da aplicação e pode construir códigos para efetuar a ligação de bibliotecas e componentes. Exemplo: uso da ferramenta livre JUnit para desenvolvimento de casos de teste para avaliar classes ou métodos desenvolvidos na linguagem Java.
* Teste da caixa preta (Teste Funcional): Técnica de teste em que o componente de software a ser testado é abordado como se fosse uma caixa-preta, ou seja, não se considera o comportamento interno do mesmo.  
  Dados de entrada são fornecidos, o teste é executado e o resultado obtido é comparado a um resultado esperado previamente conhecido.  
  Haverá sucesso no teste se o resultado obtido for igual ao resultado esperado.  
  O componente de software a ser testado pode ser um método, uma função interna, um programa, um componente, um conjunto de programas e/ou componentes ou mesmo uma funcionalidade.
* Teste da caixa cinza
* Teste de Regressão: O teste de regressão consiste na aplicação de testes à versão mais recente do software, para verificar que não existem novos defeitos em componentes já testados. Se, ao juntar um novo componente ou suas alterações a outros componentes do sistema surgirem novos defeitos em componentes inalterados, então considera-se que o sistema regrediu. Os testes de regressão, também chamados testes decorrentes de mudanças são os testes realizados novamente em um programa que já foi testado, após sua modificação, para descobrir a existência de algum defeito introduzido ou não coberto originalmente pela mudança.
* Teste de Unidade;
* Teste de Integração: Nesse teste, em vez de se atestar funcionalidades do software, se analisa a integração entre as diferentes unidades que formam o sistema. São averiguados aspectos como a interface e a dependência entre os componentes.
* Teste de Carga;
* Teste de Usabilidade: O teste tem como objetivo verificar a experiência do usuário. Dessa forma, o responsável deve checar a organização dos itens disponíveis na tela, observar se o layout está correto e se os botões se comunicam corretamente entre as diferentes páginas do sistema. A verificação cobre também a performance do programa ao executar uma determinada ação. Afinal, não é difícil nos depararmos com um aplicativo que demora anos para ser carregado, não é mesmo? Dessa forma é possível imaginar o processo ter entrado em loop ou ter executado alguma função inesperada. O processo permite, também, verificar o comportamento da plataforma em diferentes dispositivos. Caso esteja utilizando diferentes navegadores ou dispositivos de tamanhos diferentes, saberemos o quanto o layout é responsivo ou não.
* Teste de Stress: O teste de stress busca rotas imprevisíveis no uso do programa a fim de ver como reagirá aumentando a precaução antes de lançar o produto bem como suas chances de sucesso.

**Técnicas de Testes Funcionais**

* Partição de Equivalência: Nesta técnica, os valores de entrada do sistema são divididos em grupos que vão ter um comportamento parecido, para que possam ser processados da mesma maneira. As partições de equivalência podem ser aplicadas para dados válidos e inválidos, bem como a valores de saída, valores internos, valores relacionados a eventos e para os parâmetros recebidos pelas interfaces.
* Análise do valor limite: Este método parte do princípio de que o comportamento na borda de uma partição de dados tem maior probabilidade de apresentar erros. Com isso, esta técnica se torna um complemento para a partição de equivalência. Os valores máximos e mínimos de uma partição são seus valores limites e os testes podem ser feitos para dados válidos como inválidos.
* Tabela de decisão: A tabela de decisão é um método importante para documentar regras de negócios que o sistema deve cumprir. Estas são criadas a partir da análise da especificação funcional e da identificação destas regras de negócios. A tabela de decisão contém as condições de disparo, combinações de verdadeiro e falso para cada entrada de dados, bem como a ação que resulta de cada combinação.
* Transição entre status: Um sistema pode exibir diferentes comportamentos dependendo de seu status atual ou de eventos anteriores. A elaboração de um diagrama permite que o tester visualize os status, transições, entradas de dados, eventos que os acionam e as ações que podem resultar. Esta técnica pode ajudar a identificar possíveis transações inválidas.
* Técnicas baseadas na experiência: As técnicas baseadas na experiência são aqueles em que os testes são derivados da habilidade e intuição do testador, bem como de sua experiência com aplicativos e tecnologias semelhantes. Este método é amplamente utilizado na previsão de erros.

**7 Princípios do Teste**

* Teste demonstra a presença de defeitos;
* Teste exaustivo é impossível;
* Testes devem iniciar o quanto antes e erros encontrados tardiamente custam mais para corrigir;
* Agrupamento de defeitos;
* Paradoxo do pesticida;
* Teste depende do contexto;
* A ilusão da ausência de defeitos;

**Testes na Metodologia Ágil**

* Trabalha na prevenção de bugs;
* Garante que as expectativas do cliente sejam claras;
* Aceita novas ideias;
* O tester é parte do time;
* A qualidade é responsabilidade do time;
* O teste é uma atividade do processo de software.

**A interação com o Time**

O modelo ágil é composto por princípios, os quais são condensados em um manifesto ágil e descrito por meio de quatro valores:

1. Indivíduos e Interações mais que processos e ferramentas, ou seja, valorizam mais as relações entre os indivíduos que os processos burocráticos que contenham várias ferramentas.
2. Software e funcionamento mais que documentação abrangente, ou seja, a preferência é por um software funcional.
3. Colaboração com o cliente mais que a negociação de contratos
4. Responder a mudanças mais que seguir o plano (resiliência)



**Princípios Ágeis**

* Comunicação face a face;
* Feedback contínuo;
* Melhoria contínua (tanto de processos quanto de pessoas);
* Ter coragem;
* Entregar valor para o cliente (estar próximo do cliente para entender suas necessidades e assim transformar isso em cenários de testes);
* Manter Simples;
* Responder a mudanças;
* Auto-organização;
* Foco em pessoas;
* Divertir-se;

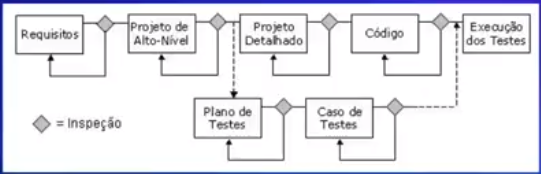
**Ferramentas e Seus Objetivos**

* Categoria de Ferramentas: Estão divididas em Ferramentas de Controle de Versões, de Gestão de Projetos, de Gestão de Testes, de Gestão de Defeitos, de Automação de Testes e Ferramentas de Apoio.



* Gestão de Testes;
* Gestão de Defeitos;
* Testes de Performance;

**Inspeção de Software e Revisões de Artefatos**

Artefatos: Tipos de subprodutos concretos produzidos durante o desenvolvimento de software (toda a documentação produzida durante o desenvolvimento do software).

**Definição dos Conceitos**

O termo defeito muitas vezes é utilizado de forma genérica. No entanto, é importante ter em mente que sua interpretação dependerá do contexto em que ele for utilizado. Defeitos encontrados através de revisões estarão relacionados às faltas no artefato sendo revisado. Quando um defeito se manifesta através de atividades de teste, por sua vez, estaremos lidando com uma falha no software. Estas definições seguem a terminologia padrão para Engenharia de Software do IEEE (IEEE 610.12, 1990):

• Erro: É um defeito cometido por um indivíduo ao tentar entender uma determinada informação, resolver um problema ou utilizar um método ou uma ferramenta.

• Defeito (ou Falta): É uma manifestação concreta de um erro num artefato de software. Um erro pode resultar em diversos defeitos.

• Falha: É o comportamento operacional do software diferente do esperado pelo usuário. Uma falha pode ter sido causada por diversas faltas e algumas faltas podem nunca causar uma falha.

Um padrão IEEE (IEEE 830, 1998), que recomenda práticas para especificação de requisitos de software, define atributos de qualidade que um documento de requisitos deve possuir. Foi considerado que a falta de qualquer um destes atributos constituiria um tipo de defeito. Assim, a seguinte taxonomia foi definida:

• Omissão: (1) Algum requisito importante relacionado à funcionalidade, ao desempenho, às restrições de projeto, ao atributo, ou à interface externa não foi incluído; (2) não está definida a resposta do software para todas as possíveis situações de entrada de dados; (3) faltam seções na especificação de requisitos; (4) faltam referências de figuras, tabelas, e diagramas; (5) falta definição de termos e unidades de medidas.

• Ambiguidade: Um requisito tem várias interpretações devido a diferentes termos utilizados para uma mesma característica ou vários significados de um termo para um contexto em particular.

• Inconsistência: Dois ou mais requisitos são conflitantes.

• Fato Incorreto: Um requisito descreve um fato que não é verdadeiro, considerando as condições solicitas para o sistema.

• Informação Estranha: As informações fornecidas no requisito não são necessárias ou mesmo usadas.

• Outros: Outros defeitos como a inclusão de um requisito numa seção errada do documento.

**Diferenciando as 3 principais formas de documentação**

## ****Scripts de teste****

É a forma mais detalhada de documentar um teste, o script. Quando os scripts de teste são mencionados, geralmente detalham linha por linha cada ação e dados necessários para rodar o teste. Um script tipicamente tem etapas que ajudam a entender como usar o programa, quais botões apertar e em qual ordem, como executar uma ação em particular no programa, etc. Esses scripts também incluem resultados específicos de cada etapa, como a verificação de uma mudança na interface. Em termos práticos, o exemplo seria:

**Ação – Clique no botão X, Resultado – A janela fechou.**

Quando um testador começa um novo trabalho, nem sempre ele sabe muito sobre o produto, a abrangência do negócio ou até mesmo sobre testes de software. Neste momento os scripts entram. Se o testador seguir cuidadosamente as instruções, inserir “abc”, clicar no botão de enviar, garantir que foi enviado e as informações salvas – essas instruções são uma boa base para entender a ideia de teste.

Existem alguns poréns a se considerar antes de dedicar muito tempo nos scripts detalhados. Os projetos de software ativos mudam constantemente, páginas são refeitas, a experiência do usuário muda e novas funcionalidades são adicionadas. Para ser mais produtivo com o tempo, os testadores precisam ter um esforço contínuo para atualizar os scripts. O problema disso é o tempo investido na atualização que poderia ser usado para rodar mais testes. Outro empecilho é que os scripts de testes são feitos para testar uma coisa específica repetidamente, fazendo uso dos mesmos caminhos e dados toda vez que o teste é executado. Isso significa que se tiver algum bug que não esteja no roteiro do script, ele não será encontrado, a não ser que o testador fuja do script. Testes com scripts não incentivam os testadores a serem criativos e nem habilidosos para encontrar bugs.

## ****Casos de teste****

A segunda forma mais detalhada de documentar o trabalho de um teste são os casos. Os casos de teste descrevem uma ideia específica a ser testada, sem detalhar os dados necessários e etapas exatas a serem executadas. Por exemplo, um caso de teste poderia ser “Testar se um código de desconto pode ser aplicado em um produto em promoção”. Isso não descreve quantos vão ser códigos ou como serão utilizados. A forma de testar este caso pode variar de tempos em tempos. O testador pode usar um link para aplicar o desconto, inserir um código, solicitar alguém do suporte ao consumidor para aplicar o desconto ou tentar outras formas criativas de encontrar o resultado. Os casos de teste proporcionam uma flexibilidade para os testadores decidirem de que forma eles pretendem completar o teste.

Esta flexibilidade dos casos é boa e ruim. Flexibilidade é benéfica quando o testador já é familiarizado com testes e com os detalhes do software que está sendo testado. Se o testador entendeu claramente o que já foi testado, o que foi modificado recentemente no programa e quantos usuários geralmente usam o programa, as estratégias de teste pode ser tanto usar os caminhos que os usuários normalmente usam ou formas mais inusitadas, para encontrar bugs escondidos.

Por outro lado, se os testadores não tiverem compreendido como o programa é usado, os riscos recentes e como avaliar esses riscos como um testador, eles podem não ter a informação ou habilidade necessária para realizar ações que podem encontrar bugs importantes.

## ****Cenários de teste****

O tipo menos detalhado de documentação é o cenário de teste. Um cenário de teste é uma descrição de um objetivo que o usuário pode encontrar ao utilizar o programa. Um exemplo seria “Testar se um usuário consegue deslogar do programa ao fechá-lo”. Tipicamente, um cenário de teste vai precisar de diferentes tipos de testes para garantir que o objetivo tenha sido bem testado. Utilizando o exemplo, o testador pode escolher fechar o programa pelo menu, fechá-lo pelo gerenciador de tarefas, desligar o computador ou como o programa reage quando fica sem memória e dá erro. Já que os cenários de testes oferecem pouca informação sobre como completar o teste, os testadores tem uma grande flexibilização para buscar uma solução.

Assim como os casos de teste, a flexibilidade dos cenários oferecem prós e contras similares. O conhecimento de causa e as habilidades em testes podem facilitar os testadores a transformar os cenários em ideias concretas, escolher a abordagem mais lógica e rodar os testes que podem extrair os problemas importantes. Este tipo de trabalho é um desafio para um testador habilidoso, mas pode ser difícil ou até mesmo impossível para um novato. Entretanto, se o novato estiver amparado em uma equipe, ele pode aprender o suficiente para conseguir.

## E como escolher qual tipo usar?

A boa notícia é que não há necessidade de escolher apenas um. Os testadores fazem uso de todos, às vezes simultaneamente. Numa equipe, os testes serão divididos por habilidade e conhecimento. Tem alguma tarefa que precisa ser repetida regularmente e tem um testador disponível? Faça uso do script de teste. Uma ferramenta robusta como o TestComplete pode usar esses scripts para criar testes automatizados em vários dispositivos, plataformas e ambientes de forma rápida. Pensou em várias ideias que precisam ser testadas e tem um testador habilidoso disponível? Faça uso dos casos de teste. Agora, precisa pensar em ações mais abrangentes, tentar pensar em como o usuário vai agir? Crie alguns cenários de teste e coloque seus melhores testadores nesta tarefa. Todas essas formas de documentação são importantes, contanto que você entenda seus benefícios e limitações.

**Exemplificando (Ou Não)**

As etapas de teste são um subconjunto do script de teste e os scripts de teste são um subconjunto dos cenários de teste. Essas são as coisas básicas dos testes. Isto é o que um testador faz em tempo real e, provavelmente, durante toda a vida, se continuar com os testes. O cenário de teste é um esboço aproximado de um módulo perticular que pode ser apresentado em vários scripts de teste. E o script de teste contém etapas de teste. As etapas do teste nada mais são do que a abordagem passo a passo para alcançar o resultado esperado.

→ Tudo começa com um cenário de teste. O cenário de teste é quando você avalia os cenários de um aplicativo para testar todas as funcionalidades pretendidas. Por exemplo, para uma ferramenta de teste entre navegadores, que fornece uma enorme biblioteca de navegadores que consiste em milhares ou mais navegadores, deve ter um cenário de teste no qual pode verificar se um navegador específico diz Chrome, quando hospedado na VM deve fornecer todas as funcionalidades pretendidas, como registro de bugs, navegação, captura de tela e assim por diante. Depois que um cenário de teste é decidido, é a vez dos scripts de teste.

Portanto, sabemos agora que precisamos testar diferentes versões do Chrome. Agora precisamos escrever os scripts de teste. Os scripts de teste descrevem como uma funcionalidade deve ser testada!

→ Um testador irá pensar em todas as maneiras possíveis de encontrar erros em um cenário de teste. Essas maneiras são anotadas e denominadas como scripts de teste. Por exemplo, se quisermos verificar a funcionalidade de uma captura de tela, escreveremos scripts que garantirão que uma captura de tela seja anotada com texto, uma captura de tela será destacada no editor de imagens, se um editor de imagens no aplicativo for fornecido. Verificaremos se a captura de tela está sendo salva e em breve.

→ Após o teste, os scripts virão casos de teste. Um script de teste é composto de vários casos de teste. Por exemplo - uma captura de tela está sendo salva é um caso de teste e essa captura de tela está sendo anotada é outro caso de teste. Tratavam-se de casos de teste positivos; casos de teste negativos envolveriam uma captura de tela que não seria salva quando anotada com texto e em breve

→ Finalmente, anotamos as etapas do teste. São etapas simples que explicam como um teste foi realizado através de um processo passo a passo. Eles são críticos na reprodução de bugs, pois diferentes usuários usarão a funcionalidade do produto através de diferentes abordagens. Por exemplo: se um usuário captura uma captura de tela e clica em salvar instantaneamente, o produto está funcionando bem, mas se um usuário captura uma captura de tela e deixa o sistema ocioso por alguns minutos e depois pressiona salvar, o produto trava! Este é apenas um exemplo para um entendimento em nível abstrato.

**Como Identificar o que Testar?**

* Teste baseado em documentação
* Técnicas de Testes
* Curiosidade
* Saber Ouvir
* Ser um bom Team Player

**Casos de Teste, Relatos de Incidente e Prioridades**

**Tópicos**

* Como escrever casos de testes a partir de uma historia;
* Como fazer um relato de um incidente (Boas Praticas)
* Classificando a prioridade e severidade de um incidente

**Como escrever casos de testes a partir de uma historia**

* O que são histórias de usuários: Uma história do usuário é uma explicação informal e geral sobre um recurso de software escrita a partir da perspectiva do usuário final.

Histórias de usuários são um dos componentes principais de um programa ágil. Elas possibilitam uma estrutura centrada no usuário para o trabalho diário, o que impulsiona a colaboração, a criatividade e um produto melhor em geral.

Uma história de usuário é a menor unidade de trabalho em uma estrutura ágil. É um objetivo final, não um recurso, expresso da perspectiva do usuário do software.

Uma história do usuário é uma explicação informal e geral sobre um recurso de software escrita a partir da perspectiva do usuário final ou cliente.

* Objetivos das histórias de usuários: articular como um recurso de software pode gerar valor para o cliente.

O objetivo de uma história de usuário é articular como uma única tarefa pode oferecer um determinado valor ao cliente. Observe que “clientes” não precisam ser usuários finais externos no sentido tradicional; também podem ser clientes internos ou colegas na empresa que dependem da sua equipe.

**Especificação dos Casos de Teste**

* Título: Sucinto, simples e autoexplicativo, de modo que o analista de testes saiba qual a validação que o teste propõe.
* Objetivo: Descrever o que será executado, fornecendo uma visão geral do teste a ser realizado.
* Pré-Condição: Condições necessárias para que o caso de teste seja executado.
* Passos: Todas as ações que o analista deve seguir para chegar no resultado esperado, e sua nomenclatura deve iniciar com um verbo no infinitivo (clicar, preencher, etc.). ou imperativo (acesse, preencha, etc).
* Resultados Esperados;

**Classificação de bugs: Severidade e Prioridade**

**Severidade**

* S1 – Crítica / Show Stopper: Bloqueia o teste de uma função ou feature, o release e causa crash na aplicação nos principais casos de uso.
* Botão não funciona, impedindo o uso da feature;
* Segurança;
* Feature esperada não aparece na build;
* Perda grave de dados;
* S2 – Grave: Feature funciona pobremente
* Input esperados causam crash ou efeitos indesejados;
* Input incomum causa efeitos irreversíveis;
* S3 – Moderada: Feature não atinge certos critérios de aceitação, mas sua funcionalidade em geral não é afetada.
* Mensagem de erro ou sucesso não é exibida
* Input incomum causa efeitos indesejados, mas contornáveis;
* S4 – Pequena: Quase nenhum impacto na funcionalidade, mas ainda é um erro válido.
* Erro ortográfico
* Pequenos erros de UI

**Prioridade**

* P1 – Crítico: Tem que ser solucionado imediatamente
* Erros de severidade S1 encontrados produção
* Erros de UI que afetam feeling/UX
* P2 – Alta: Feature não é usável como deveria ser, por problema no código
* Configuração de dependências
* Detalhes da feature não levantados durante seu desenvolvimento
* Deadlines do projeto
* P3 – Média: Problemas que podem ser avaliados para entrar no ciclo de desenvolvimento no futuro; dependendo dos recursos disponíveis
* Efeitos indesejados quando o sistema é submetido a inputs incomuns ou não-existentes no ambiente do usuário
* P4 – Baixa: Erros que não afetam funcionalidade
* Pequenas melhorias de UI e UX
* Erros de digitação

**Automação de Testes**

**Desenvolvendo Scripts de Testes Automatizados**

Conteúdo: Configuração do ambiente; Padrão Page Objects; Conceitos básicos da linguagem Java; Cucumber; Abordagem Page Factory; Selenium; Validação com JUnit; Gerando Evidências de Teste;

**Configurações do Ambiente de Testes**

pageObjects: pacote responsável por representar as páginas do sistema.

stepsDefinitions: pacote responsável pelas classes que representarão as funções que serão automatizadas pelo sistema.

Runners: pacote responsável pela classe que executa os testes.

Utils: Neste pacote, teremos uma classe com vários métodos que serão úteis para a automação (não precisaremos duplicar métodos em outros lugares do projeto).

**Padrão PageFactory**

* Mapeamento de elemento mais simples;
* Leitura melhor de código;
* Melhor abstração da página do sistema;

**Selenium: Mapeamento de Elementos**

* Id;
* Name;
* Linktext;
* Xpath;
* cssSelector;

**Selenium: Mapeamento de Elementos**

* click;
* sendkeys;
* clear;
* getText;
* isDisplayed;
* isSelected;

**Selenium: Actions**

* clickAndHold;
* click com botão direito;
* Mover para elemento;

**Desenvolvendo Scripts de Teste Automatizados para Mobile**

**Conteúdo**

* Configuração do ambiente;
* Conceitos básicos da linguagem Java;
* Cucumber;
* Appium;
* Padrão Page Objects;
* Abordagem Page Factory;
* Validação com Junit;
* Gerando evidências de teste;

**Testes Unitários**

* O que são? São testes implementados a nível de componente, ou seja, testes feitos na menor unidade do código, sejam eles métodos ou funções.
* Quais as vantagens? Encontrar defeitos durante o desenvolvimento da aplicação.
* Características: Os testes unitários devem ser rápidos (FAST), uma vez que os projetos podem conter milhares de testes unitários. Além disso, os testes unitários devem ser independentes (INDEPENDENT), ou seja, eles não podem ser aninhados (um teste não pode estar contido em outro).

Os testes também devem ser capazes de rodar várias vezes sem problemas (REPEATABLE). Os testes também precisam ser capazes de conhecer seu resultado (SELF-VERIFYNG), ou seja, eles precisam ser capazes de detectar se passaram ou não nos testes, para isso, o teste não pode estar viciado. Por fim, os testes devem ser atemporais (TIMELY), pois eles podem ser executados após um longo tempo do seu desenvolvimento, por exemplo, serem desenvolvidos e 5 anos depois, serem testados.