TESTE DE SOFTWARE

AULA 1

Profª Maristela Regina Weinfurter Teixeira

CONVERSA INICIAL

Vamos fazer um exercício mental sobre a aquisição de um refrigerador. Se estivéssemos numa loja de eletrodomésticos, pensemos no que analisaríamos sobre o que é essencial e o que é nosso desejo de consumo referente a esse refrigerador:

a. Refrigerador na parte superior e um freezer na parte inferior;
b. Manual de instruções;
c. Nenhum arranhão;
d. Na cor cinza;
e. Com o menor consumo de eletricidade entre os modelos similares;
f. Assistência técnica que solucione problemas e tenha um excelente atendimento na nossa cidade.

Enfim, poderíamos listar vários itens importantes para que nossa aquisição seja a melhor possível. Caso uma ou mais requisitos nossos não forem atendidos, naturalmente sairemos frustrados dessa compra.

Esse tipo de expectativa parece não se aplicar a novas instalações de *software*; exemplos de *software* entregue não funcionando como esperado, ou não funcionando, são comuns. Por que isso ocorre? Não existe uma causa única que possa ser corrigida para resolver o problema, mas um importante fator contribuinte é a inadequação dos testes aos quais os aplicativos de *software* são expostos.

O teste de *software* não é complexo nem difícil de implementar, mas é uma disciplina que raramente é aplicada com algo que se aproxime do rigor necessário para fornecer confiança no *software* entregue. O teste de software é uma atividade cara, tanto no nível de esforço dos profissionais de forma manual quanto por conta de ferramentas e tecnologias.

Este conteúdo explora os fundamentos sobre testes de *software* dentro da disciplina de qualidade de *software*, observando técnicas, metodologias, ferramentas e ciclo de vida de *software* e de testes.

As ideias-chave do teste de *software* são aplicáveis independentemente do *software* envolvido e de qualquer metodologia de desenvolvimento específica (clássico, ágil, ou qualquer outro tipo de modelo).

Começamos falando sobre os fundamentos de processos de teste, testes e depuração de código, testes dentro da garantia de qualidade de *software*, erros, defeitos, falhas e o modelo de efeito-causa.

São conteúdos introdutórios para esclarecedores quando falamos da importância dos testes dentro de todo o ciclo de desenvolvimento de *software*, independentemente de sua abordagem.

TEMA 1 – FUNDAMENTOS DE PROCESSOS DE TESTES

1.1 Princípios e fundamentos sobre testes de *software*

O teste de *software* é uma atividade abrangente e complexa e pode ser baseado em princípios gerais oriundos do próprio desenvolvimento de *software*. Esses princípios nos fazem compreender melhor a efetiva necessidade da implementação de testes de *software*.

A execução de um teste por meio de um sistema de *software* pode mostrar apenas a existência de um ou mais defeitos. O teste não pode mostrar que o *software* está livre de erros. Embora possa haver outros objetivos, geralmente o objetivo principal do teste é encontrar defeitos. Portanto, os testes devem ser projetados para encontrar o maior número possível de defeitos.

Uma vez que os testes encontrem problemas, certamente esperamos por mais testes para encontrar problemas adicionais, até que eventualmente encontremos todos eles. Entretanto esse processo de encontrar erros exaustivamente não é possível. Segundo Hambling (2015), o recomendado é realizarmos pequenos testes em pedaços de *software* de forma exaustiva. O teste exaustivo inclui todas as possiblidades de combinações de dados presentes nos dados do *software*. Porém, mesmo aplicando-se o teste exaustivo num pequeno pedaço de *software*, a possibilidade de combinações pode ainda ser impraticável.

Quando iniciamos a discussão sobre o motivo das falhas de *software*, é importante salientar sobre os testes iniciais. Esse princípio é importante porque, à medida que a data de implantação proposta se aproxima, a pressão do tempo pode aumentar drasticamente. Com isso, uma das atividades que são deixadas para trás são os testes, que são efetuados somente após a entrega do projeto. E o perigo encontra-se justamente neste episódio, pois quanto mais cedo a

atividade de teste for iniciada, maior será o tempo decorrido disponível. Os testadores não precisam esperar até que o *software* esteja disponível para teste.

O ideal é a implementação de testes ao longo de todo o ciclo de vida de desenvolvimento de *software*. Os documentos de requisitos são a base para os testes de aceitação, portanto a criação dos testes de aceitação pode começar assim que os documentos de requisitos estiverem disponíveis. À medida que criamos esses testes, ele destacará o conteúdo dos requisitos. Testamos requisitos porque estes podem estar ambíguos, ausentes, incompletos ou mal interpretados. São vários problemas em *software* que, ao rastrearmos, chegamos a requisitos ausentes ou incorretos. O uso de revisões pode quebrar o ciclo *erro-defeito-falha*. Nos testes iniciais, estamos tentando encontrar erros e defeitos antes que eles passem para o próximo estágio do processo de desenvolvimento. As primeiras técnicas de teste estão tentando mostrar o que é produzido como uma especificação do sistema.

Um assunto bastante discutido em engenharia de *software* é como os erros impactam nos custos do desenvolvimento de *software* nos seus vários estágios. A Tabela 1 explora o custo da correção de erros em cada um dos estágios do ciclo de vida do *software*.

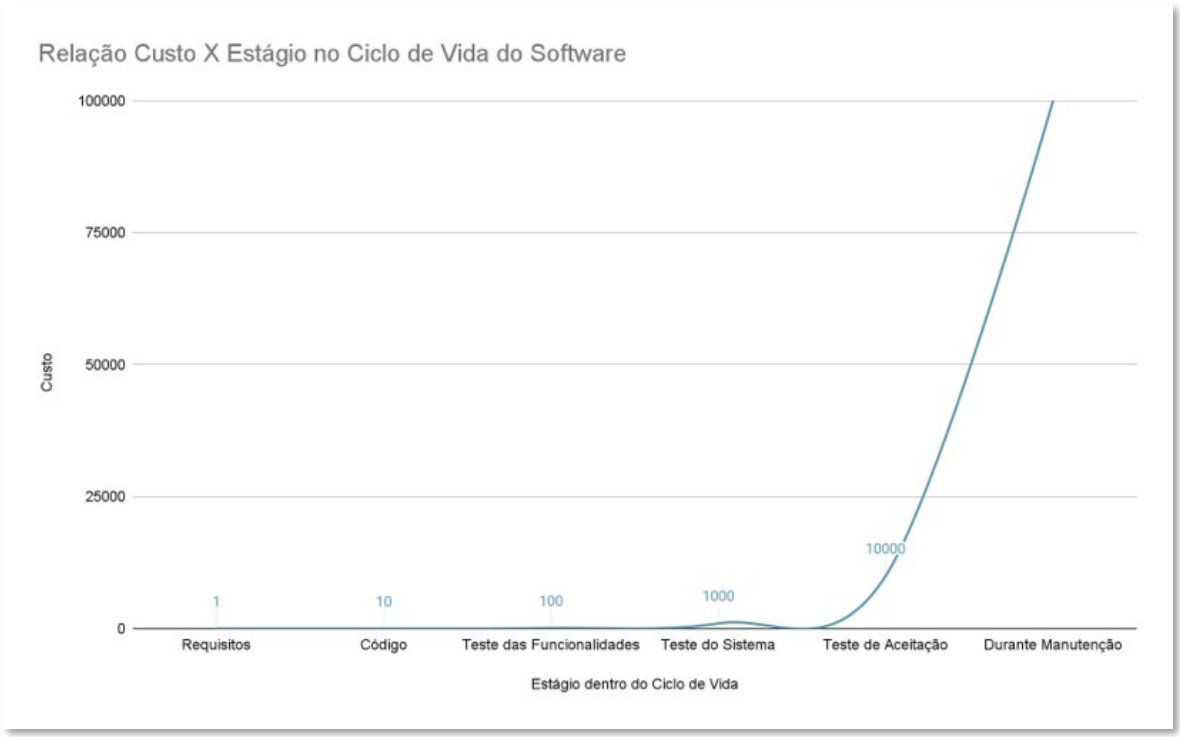
Tabela 1 – Custo comparativo para correção de erros.

Estágio no qual o erro foi encontrado	Comparativo de custos
1. Requisitos	$1
2. Código	$10
3. Teste das funcionalidades	$100
4. Teste do sistema	$1.000
5. Teste de aceitação	$10.000
6. Durante manutenção	$100.000

Fonte: Hambling, 2015.

Com base nos dados da Tabela 1, podemos visualizar de forma gráfica na Figura 1 o real impacto de tais custos de correções dos erros de *software* por meio de testes feitos nos primeiros estágios do ciclo de desenvolvimento de *software*.

Figura 1 – Custo comparativo para correção de erros



Fonte: Hambling, 2015.

Os objetivos de vários estágios de teste podem ser diferentes. Nos processos de revisão, podemos nos concentrar em se os documentos são consistentes e se nenhum erro foi introduzido quando os documentos foram produzidos. Outras etapas do teste podem ter outros objetivos. O ponto importante é que o teste tenha objetivos definidos. As metodologias ágeis impulsionaram a necessidade dos testes em quaisquer etapas do ciclo de vida de *software*, tendo seu ponto de partida o teste inicial.

1.2 Princípios e fundamentos sobre testes de *software*

Considerando que o teste é um processo, ele deve ser detalhado como um processo de teste fundamental aplicável em todas as etapas de teste. A parte mais visível do teste é o ato da própria execução do teste, mas precisamos preparar a execução de testes, analisar os que foram executados e verificar se o teste foi concluído. Tanto o planejamento quanto a análise são atividades muito necessárias que potencializam e ampliam os benefícios da própria execução do teste. Não adianta testar sem decidir *como*, *quando* e *o que* testar. O

planejamento também é necessário para as abordagens de teste menos formais, como testes exploratórios.

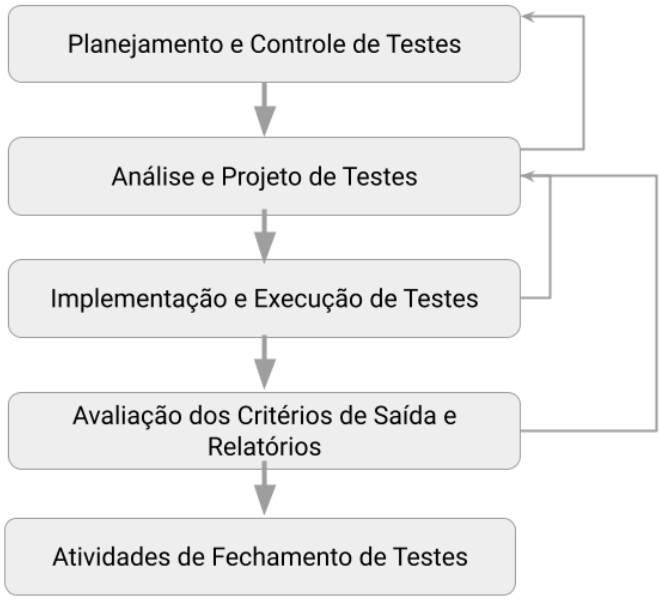
Os principais testes de *software* são divididos cinco partes e abrangem todos os aspectos do teste (Figura 2):

Figura 2 – Processos fundamentais de teste

Fonte: Hambling, 2015.

Embora as principais atividades sejam em uma sequência ampla, elas não são realizadas de forma rígida. Uma atividade anterior pode precisar ser revisitada. Às vezes, um defeito encontrado na execução do teste pode ser resolvido adicionando uma funcionalidade que originalmente não estava presente (falta por erro ou os novos recursos são necessários para corrigir a outra parte). Os próprios novos recursos precisam ser testados, portanto, embora a implementação e a execução estejam em andamento, a atividade anterior de análise e *design* deve ser realizada para os novos recursos (Figura 3).

Figura 3 – Iteração de atividades de testes



Fonte: Hambling, 2015.

Às vezes precisamos fazer duas ou mais atividades principais em paralelo. Vale lembrar que a pressão do tempo pode trazer como consequências o início dos testes sem os seus devidos planejamentos e análises.

Planejar é determinar o que será testado e como isso será alcançado. É quando desenhamos um mapa: como as atividades serão realizadas e quem as executará. O planejamento de teste também é o momento em que definimos os critérios de conclusão do teste. Os critérios de conclusão são a forma como sabemos quando o teste é concluído. O controle, por outro lado, é o que fazemos quando as atividades não condizem com os planos. É a atividade contínua em que comparamos o progresso com o plano. À medida que o progresso ocorre, podemos precisar ajustar os planos para atingir as metas, se isso for possível. Portanto, precisamos realizar o planejamento e o controle ao longo das atividades de teste. Planejamos no início, mas à medida que os testes avançam, realizamos atividades de monitoramento e controle (monitoramento para medir o que aconteceu, controle para ajustar atividades futuras à luz da experiência). O monitoramento e o controle retroalimentam a atividade contínua de planejamento.

A análise e o projeto estão preocupados com os detalhes do que testar (condições de teste) e como combinar as condições de teste em casos de teste, de modo que um pequeno número de casos de teste possa cobrir o maior

número possível de condições de teste. A fase de análise e projeto é a ponte entre o planejamento e a execução do teste. Olhamos para trás para o planejamento (horários, pessoas, o que vai ser testado) e para frente para a atividade de execução (resultados esperados do teste, qual ambiente será necessário). Uma parte do processo de projeto precisa considerar quais dados de teste serão necessários para as condições de teste e casos de teste que foram elaborados.

O projeto de teste ainda se preocupa com o comportamento do teste para um determinado conjunto de circunstâncias. Às vezes, o resultado esperado de um teste é trivial: ao encomendar livros de uma livraria *online*, por exemplo, sob nenhuma circunstância o dinheiro deve ser devolvido no cartão do cliente sem a intervenção de um supervisor. Se não detalharmos os resultados esperados antes de iniciar a execução do teste, há um perigo real de perdermos um item de detalhe que é extremamente importante.

A atividade de implementação e execução de teste envolve a execução de testes, bem como quaisquer atividades de configuração/desmontagem para o teste. Também envolverá a verificação do ambiente de teste antes do início do teste. A execução do teste é a parte mais visível do teste, mas não é possível sem outras partes do processo de teste fundamental. Não se trata apenas de fazer testes. Como já mencionamos, os mais importantes precisam ser executados primeiro. Um aspecto importante realizado neste estágio é combinar os casos de teste em um procedimento geral de execução para que o tempo de teste possa ser utilizado de forma eficiente. Aqui a ordenação lógica dos testes é fundamental para que, sempre que possível, o resultado de um teste crie as pré-condições para um ou mais testes que estão posteriormente na sequência de execução.

À medida que os testes são executados, seus resultados precisam ser registrados e haver uma comparação feita entre os resultados esperados e os resultados reais. Sempre que houver uma discrepância entre os resultados esperados e os reais, isso precisa ser investigado. Se necessário, um incidente de teste deve ser levantado. Cada incidente requer investigação, embora a ação corretiva não seja necessária em todos os casos.

Quando algo muda (*software*, dados, procedimentos de instalação, documentação do usuário etc.), precisamos fazer dois tipos de teste no *software*. Em primeiro lugar, os testes devem ser executados para garantir que o problema

foi corrigido. Também precisamos ter certeza de que as alterações não danificaram o *software* em outro lugar. Esses dois tipos geralmente são chamados de *reteste* e *teste de regressão*, respectivamente. No reteste, estamos analisando em detalhes a alteração de uma área de funcionalidade, enquanto o teste de regressão deve abranger todas as funções principais para garantir que não ocorram alterações não intencionais. Em um sistema financeiro, devemos incluir o processamento de final de dia/final de mês/final de ano, por exemplo, em um pacote de teste de regressão.

A implementação e a execução do teste são geralmente as atividades mais visíveis e compostas pelas seguintes partes:

- Desenvolver e priorizar casos de teste, criar dados de teste, escrever procedimentos de teste e, opcionalmente, preparar equipamentos de teste e escrever *scripts* de teste automatizados;
- Coletar casos de teste em suítes de teste, onde os testes podem ser executados um após o outro para maior eficiência;
- Verificar se a configuração do ambiente de teste está correta;
- Executar casos de teste na ordem determinada. Isso pode ser feito manualmente ou usando ferramentas de execução de teste;
- Manter um registro das atividades de teste, incluindo o resultado (aprovado/reprovado) e as versões de *software*, dados, ferramentas e *testware* (*scripts* etc.);
- Comparar os resultados reais com os resultados esperados;
- Relatar discrepâncias como incidentes com o máximo de informações possível, incluindo, se possível, análise causal (defeito de código, especificação de teste incorreta, erro de dados de teste ou erro de execução de teste);
- Sempre que necessário, repetir as atividades de teste quando as alterações forem feitas após os incidentes levantados. Isso inclui a reexecução de um teste que falhou anteriormente para confirmar uma correção (reteste), a execução de um teste corrigido e a execução de testes aprovados anteriormente para verificar se os defeitos não foram introduzidos (teste de regressão).

Os critérios de saída devem ser definidos durante o planejamento do teste e antes do início da execução do teste. No final da execução do teste, o gerente

de teste verifica se eles foram atendidos. Se o critério fosse que 85% de cobertura de instrução (ou seja, 85% de todas as instruções executáveis foram executadas), e como resultado da execução o número é de 75%, existem duas ações possíveis: *alterar os critérios de saída* ou *executar mais testes*. É possível que, mesmo que os critérios predefinidos fossem atendidos, mais testes seriam necessários. Além disso, escrever um resumo de teste para as partes interessadas diria o que foi planejado, o que foi alcançado, destacar quaisquer diferenças e, em particular, coisas que não foram testadas.

A quarta etapa do processo de teste, avaliando os critérios de saída, compreende o seguinte:

- Verificar se os critérios de saída previamente determinados foram atendidos;
- Determinar se são necessários mais testes ou se os critérios de saída especificados precisam ser alterados;
- Redigir o resultado das atividades de teste para os patrocinadores do negócio e outras partes interessadas.

Finalmente, os testes nessa fase foram concluídos. As atividades de encerramento de teste concentram-se em garantir que tudo esteja arrumado, relatórios escritos, defeitos fechados e esses defeitos adiados para outra fase claramente visto como tal. Ao final do teste, o estágio de encerramento do teste é composto das seguintes atividades:

- Garantir que a documentação esteja em ordem; é definido o que foi entregue (pode ser mais ou menos do que o planejado originalmente), fechando os incidentes e levantando mudanças para entregas futuras, documentando que o sistema foi aceito;
- Fechar e arquivar o ambiente de teste, infraestrutura e ambiente de testes usados;
- Passar o ambiente de teste para a equipe de manutenção;
- Anotar as lições aprendidas com este projeto de teste para o futuro e incorporar lições para melhorar o processo de teste (*maturidade de teste*).

Para o desenvolvimento que utiliza metodologias ágeis, os processos fundamentais de testes são muito parecidos. O que pode variar são algumas formas de como utilizamos determinadas atividades de teste, utilização maior de ferramentas automatizadas para auxiliar nos testes e no estágio final do

processo de teste, agendar uma última *sprint* para efetuar os devidos fechamentos dos testes. Além claro, de que nas entregas, o conjunto de definição de pronto permite que o gráfico de *burn-down* seja elaborado permitindo o acompanhamento do progresso dos testes.

TEMA 2 – TESTE E DEPURAÇÃO

Teste e depuração são tipos de atividades diferentes, porém ambas importantes e complementares. Depuração é o processo pelo qual os desenvolvedores passam para identificar a causa de *bugs* ou defeitos no código e realizar correções. Idealmente, é feita alguma verificação da correção, mas isso pode não se estender à verificação de outras partes do sistema que não foram afetadas inadvertidamente pela correção. O teste, por outro lado, é uma exploração sistemática de um componente ou sistema com o objetivo principal de encontrar e relatar defeitos. O teste não inclui a correção de defeitos – estes são repassados ao desenvolvedor para correção. O teste, no entanto, garante que as alterações e correções sejam verificadas quanto ao seu efeito em outras partes do componente ou sistema (Brown, 2014).

A depuração eficaz é essencial antes que o teste comece a elevar o nível de qualidade do componente ou sistema para um nível que valha a pena testar, ou seja, um nível suficientemente robusto para permitir a execução de testes rigorosos. A depuração não dá confiança de que o componente ou sistema atenda completamente aos seus requisitos. O teste faz um exame rigoroso do comportamento de um componente ou sistema e relata todos os defeitos encontrados para a equipe de desenvolvimento corrigir. O teste então repete testes suficientes para garantir que as correções de defeitos tenham sido eficazes. Portanto, ambos são necessários para alcançar um resultado de qualidade.

Há quem diga que *depuração* é um termo infeliz, com sua associação com defeitos. O fato é que o que chamamos de *depuração* é uma atividade que faremos o tempo todo, seja implementando um novo recurso, aprendendo como algo funciona ou corrigindo um *bug*. Um termo melhor pode ser *explorar*, mas ficaremos com *depuração*, já que a atividade a que se refere é inequívoca, independentemente da motivação (Brown, 2014). A depuração é uma habilidade frequentemente negligenciada: parece que a maioria dos programadores não encontram base em obras, livros e periódicos. O fato é que a depuração é uma

habilidade que pode ser ensinada e é uma maneira importante pela qual os programadores entendem não apenas a estrutura em que estão trabalhando, mas também seu próprio código e o de sua equipe.

Algumas ferramentas e técnicas são utilizadas para auxiliar no processo de depuração de código. Mas para usarmos de forma adequada tais ferramentas e técnicas, é importante observarmos alguns princípios, segundo (Brown, 2014].

O primeiro Princípio da Depuração é o processo de eliminação. Os sistemas de computador modernos são incrivelmente complicados e, se tivéssemos que manter todo o sistema em nossa memória, provavelmente nem saberíamos por onde começar. Sempre nos deparamos com um problema que não é imediatamente óbvio, nosso primeiro pensamento deve ser “O que posso eliminar como fonte do problema?”. Quanto mais eliminarmos, menos lugares teremos para procurar.

A eliminação pode assumir muitas formas, tais como alguns exemplos comuns:

- Comentar ou desabilitar sistematicamente blocos de código;
- Escrever o código que pode ser coberto por testes de unidade; os próprios testes de unidade fornecem uma estrutura para eliminação;
- Analisar o tráfego de rede para determinar se o problema está no lado do cliente ou do servidor;
- Testar uma parte diferente do sistema que tenha semelhanças com a primeira;
- Usar a entrada que funcionou antes e alterar essa entrada com uma parte de cada vez até que o problema seja exibido;
- Usar o controle de versão para voltar no tempo, um passo de cada vez, até que o problema desapareça;
- Funcionalidade de simulação para eliminar subsistemas complexos.

Normalmente a ferramenta utilizada para depuração de código é justamente a mesma IDE utilizada para codificação. Os procedimentos são muito parecidos entre as várias IDEs e para as várias linguagens de programação.

Alguns procedimentos mais utilizados durante o processo de depuração de código:

1. Criação de *breakpoints* em partes do código para que possamos ir passo a passo pelo código entre estes pontos de atenção;

2. Uma vez colocados os *breakpoints*, podemos executar a depuração sobre o código parando exatamente nestes *breakpoints*;
3. Acionamos variáveis, estruturas e outros elementos que possamos identificar o exato conteúdo no momento no qual o código passa pelo ponto que estamos observando;
4. Podemos executar a depuração linha a linha de código ou de ponto a ponto de parada (*breakpoints*), observando-se os conteúdos transformados dentro da funcionalidade que estamos depurando.

Muitas vezes, os problemas são devidos a interações complexas entre dois ou mais componentes: eliminemos (ou simulemos) qualquer um dos componentes, e o problema pode desaparecer, mas o problema não pode ser isolado a nenhum componente único. Mesmo nessa situação, porém, a eliminação pode ajudar a diminuir o problema. A eliminação é mais bem-sucedida quando é cuidadosa e metódica. É muito fácil perder coisas quando eliminamos componentes sem considerar como esses componentes afetam o todo.

TEMA 3 – GARANTIA DA QUALIDADE E TESTES

As ferramentas de teste modernas estão se tornando cada vez mais avançadas e fáceis de usar. O seguinte descreve como a atividade de teste de *software* evoluiu e está evoluindo ao longo do tempo. Isso define a perspectiva de onde as ferramentas de teste automatizadas estão indo.

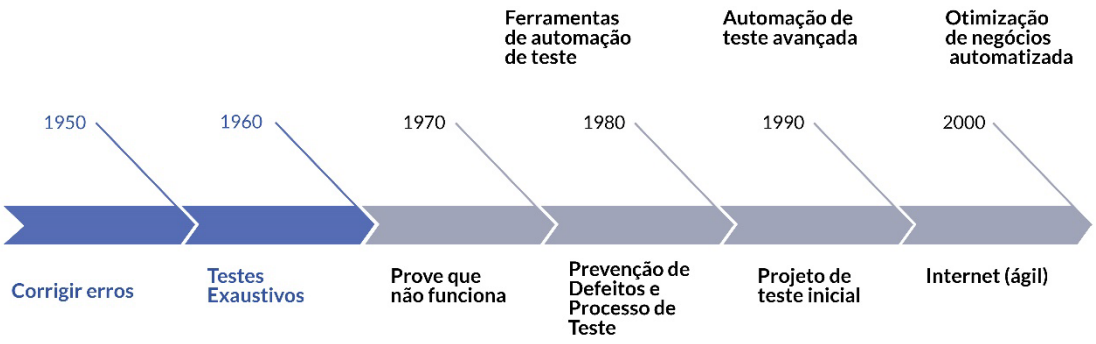
Teste de *software* é a atividade de executar uma série de execuções dinâmicas de programas de *software* após o desenvolvimento do código-fonte do *software*. É realizado para descobrir e corrigir o maior número possível de erros potenciais antes da entrega ao cliente. Como apontado anteriormente, o teste de *software* ainda é uma arte. Pode ser considerada uma técnica de gestão de risco; a técnica de garantia de qualidade, por exemplo, representa a última defesa para corrigir desvios de erros na especificação, projeto ou código (Lewis, 2009).

Ao longo da história, houve muitas definições e avanços no teste de *software*. A Figura 4 ilustra graficamente essas evoluções. Na década de 1950, o teste de *software* era definido como “o que os programadores faziam para encontrar *bugs* em seus programas”. No início da década de 1960, a definição

de teste passou por uma revisão. Foi considerado o teste exaustivo do *software* em termos dos possíveis caminhos ao longo do código, ou enumeração total das possíveis variações dos dados de entrada. Observou-se que era impossível testar completamente um aplicativo porque (1) o domínio das entradas do programa é muito grande, (2) há muitos caminhos de entrada possíveis e (3) problemas de *design* e especificação são difíceis de testar. Por causa dos pontos anteriores, testes exaustivos foram descontados e considerados teoricamente impossíveis.

À medida que o desenvolvimento de *software* amadureceu ao longo das décadas de 1960 e 1970, a atividade de desenvolvimento foi chamada de *ciência da computação*. O teste de *software* foi definido como “o que é feito para demonstrar a correção de um programa” ou como “o processo de estabelecer a confiança de que um programa ou sistema faz o que deveria fazer” no início dos anos 1970. Uma técnica de ciência da computação de curta duração que foi proposta durante a especificação, projeto e implementação de um sistema de *software* foi a verificação de *software* por meio de “prova de correção”. Embora esse conceito fosse teoricamente promissor, na prática era muito demorado e insuficiente. Para testes simples, foi fácil mostrar que o *software* funciona e provar que teoricamente funcionará. No entanto, como a maior parte do *software* não foi testada usando essa abordagem, um grande número de defeitos permaneceu a ser descoberto durante a implementação real. Logo se concluiu que a *prova de correção* era um método ineficiente de teste de *software*. No entanto, ainda hoje há necessidade de demonstrações de correção, como testes de aceitação, conforme descrito em várias seções deste livro.

Figura 4 – Histórico de testes de *software*



No final da década de 1970, afirmou-se que o teste é um processo de execução de um programa com a intenção de encontrar um erro; não provar que

ele funciona. A nova definição enfatizou que um bom caso de teste é aquele que tem uma alta probabilidade de encontrar um erro ainda não descoberto. Um teste bem-sucedido é aquele que descobre um erro ainda não descoberto. Esta abordagem foi exatamente o oposto da seguida até este ponto.

Os seguintes bons princípios de teste foram propostos (Lewis, 2009):

- Uma parte necessária de um caso de teste é uma definição da saída ou resultado esperado;
- Os programadores devem evitar tentar testar seus próprios programas;
- Uma organização de programação não deve testar seus próprios programas.
- Inspecione cuidadosamente os resultados de cada teste;
- Os casos de teste devem ser escritos para condições de entrada inválidas e inesperadas, bem como válidas e esperadas;
- Examinar um programa para ver se ele não faz o que deveria fazer é apenas metade da batalha. A outra metade é ver se o programa faz o que não deveria fazer;
- Evite casos de teste descartáveis, a menos que o programa seja realmente um programa descartável;
- Não planeje um esforço de teste sob a suposição tácita de que nenhum erro será encontrado;
- A probabilidade da existência de mais erros em uma seção de um programa é proporcional ao número de erros já encontrados nessa seção.

A década de 1980 viu a definição de teste estendida para incluir a prevenção de defeitos. Projetar testes é uma das técnicas de prevenção de *bugs* mais eficazes conhecidas. Foi sugerido que fosse necessária uma metodologia de teste, especificamente, que o teste deve incluir revisões ao longo de todo o ciclo de vida de desenvolvimento de *software* e que deve ser um processo gerenciado. Foi promovida a importância de testar não apenas um programa, mas os requisitos, projeto, código, testes em si e o programa. Ainda nesta mesma década, surgiram ferramentas de teste automatizadas para automatizar o esforço de teste manual para melhorar a eficiência e a qualidade do aplicativo de destino.

Até o início da década de 1980, segundo Lewis (2009), o teste referia-se ao que era feito em um sistema depois que o código de trabalho era entregue,

no entanto o teste hoje é um *teste maior*, no qual um testador deve estar envolvido em quase todos os aspectos do ciclo de vida de desenvolvimento de *software*. Uma vez que o código é entregue para teste, ele pode ser testado e verificado, mas se algo estiver errado, as fases de desenvolvimento anteriores devem ser investigadas. Se o erro foi causado por uma ambiguidade de projeto ou um descuido do programador, é mais simples tentar encontrar os problemas assim que eles ocorrem, não esperar até que um produto realmente funcional seja produzido. Estudos mostraram que cerca de 50% dos *bugs* são criados nos requisitos ou estágios de projeto, e isso pode ter um efeito composto e criar mais bugs durante a codificação. Quanto mais cedo um *bug* ou problema for encontrado no ciclo de vida, mais barato será a correção. Em vez de testar um programa e procurar bugs nele, os requisitos ou projetos podem ser rigorosamente revisados. Infelizmente, ainda hoje, muitas organizações de desenvolvimento de *software* acreditam que o teste de *software* é uma atividade de *back-end*.

No início da década de 1990, o poder do projeto de teste inicial foi reconhecido. O teste foi redefinido para “planejar, projetar, construir, manter e executar testes e ambientes de teste”. Essa era uma perspectiva de garantia de qualidade de teste que assumia que um bom teste é um processo gerenciado, uma preocupação total do ciclo de vida com a testabilidade. Além disso, no início da década de 1990, ferramentas de teste de captura/reprodução mais avançadas ofereciam linguagens de *script* ricas e recursos de geração de relatórios. As ferramentas de gerenciamento de teste ajudaram a gerenciar todos os artefatos, desde requisitos e *design* de teste, até *scripts* de teste e defeitos de teste. Além disso, ferramentas de desempenho comercialmente disponíveis chegaram para testar o desempenho do sistema. Essas ferramentas testaram o estresse e testaram a carga do sistema de destino para determinar seus pontos de ruptura. Isso foi facilitado pelo planejamento de capacidade. Neste período surgiram as primeiras abordagens ágeis com técnicas de teste, tais como testes exploratórios, testes rápidos e testes baseados em risco.

No início dos anos 2000, com a ideia de governança de TI somada à nova forma de metodologia ágil, com a introdução de várias ferramentas de automação de testes, bem como com o advento da internet, agora com *software* orientado para *web* e mais adiante para equipamentos móveis, faz-se necessário que o teste de *software* não olhe apenas para o *backend*, mas também para

outras áreas, tais como UI/UX, infraestrutura, segurança, dados, comunicação e todo o contexto tecnológico necessário para que o *software* funcione com qualidade.

Dentre as metodologias ágeis, a programação extrema (XP) é um exemplo de tal tendência. XP é uma abordagem pouco ortodoxa para o desenvolvimento de *software*, e tem sido argumentado que não tem aspectos de design. A metodologia de programação extrema propõe um afastamento radical dos processos de desenvolvimento de *software* comumente aceitos. Existem realmente duas regras de XP:

1. Faça um pequeno *design*; e
2. Sem requisitos, apenas histórias de usuários.

Os discípulos da programação extrema insistem:

– Realmente não existem regras, apenas sugestões. A metodologia XP exige pequenas unidades de projeto, de dez minutos a meia hora, feitas periodicamente de um dia entre as sessões até uma semana inteira entre as sessões. Efetivamente, nada é projetado até que seja hora de programá-lo.

Embora a maioria das pessoas no negócio de desenvolvimento de *software* considerem a documentação de requisitos vital, a XP recomenda a criação da menor documentação possível. Nenhuma documentação de requisito inicial é criada no XP e muito pouco é criado no processo de desenvolvimento de *software*.

No modelo XP, o desenvolvedor cria cenários de teste antes de fazer qualquer outra coisa. A premissa básica por trás do *design test-first* é que a classe de teste é escrita antes da classe real; assim, o propósito final da classe real não é simplesmente cumprir um requisito, mas simplesmente passar em todos os testes que estão na classe de teste. O problema com essa abordagem é que são necessários testes independentes para descobrir coisas sobre o produto que o desenvolvedor não pensou ou não conseguiu descobrir durante seus próprios testes.

Com o aumento da complexidade no desenvolvimento de *software*, os testes também tomaram outra proporção e importância dentro do contexto da qualidade de *software*.

TEMA 4 – ERROS, DEFEITOS E FALHAS

Diariamente ouvimos, após algumas reuniões de trabalho ou até mesmo quando ligamos para tentar resolver algum problema em bancos, comércio eletrônico, ou quaisquer outros serviços digitais dos quais somos usuários diretos ou indiretos, que houve algum problema no sistema.

São comentários conforme a lista a seguir:

- O sistema travou durante a produção;
- A área de TI cometeu um erro;
- Após uma revisão, encontramos um defeito no plano de teste;
- Encontrei um *bug* em um aplicativo hoje;
- O sistema quebrou.
- Uma falha foi relatada no subsistema de monitoramento.

Para identificação e correção ou melhoria em relação ao sistema, aplicativo, entre outros vocábulos que no final referem-se a um *software* a área de gestão da garantia da qualidade possui vários tipos de testes, verificações, validações, inspeções e auditorias, de acordo com a necessidade para cada situação.

Esses testes e outras formas de validações existem porque de fato podemos cometer enganos, erros, e podem ocorrer defeitos e falhas no *software* (Pressman, 2011).

São vocábulos que a princípio parecem dizer a mesma coisa, porém cada um deles tem um significado diferente quando nos referimos a testes de *software*.

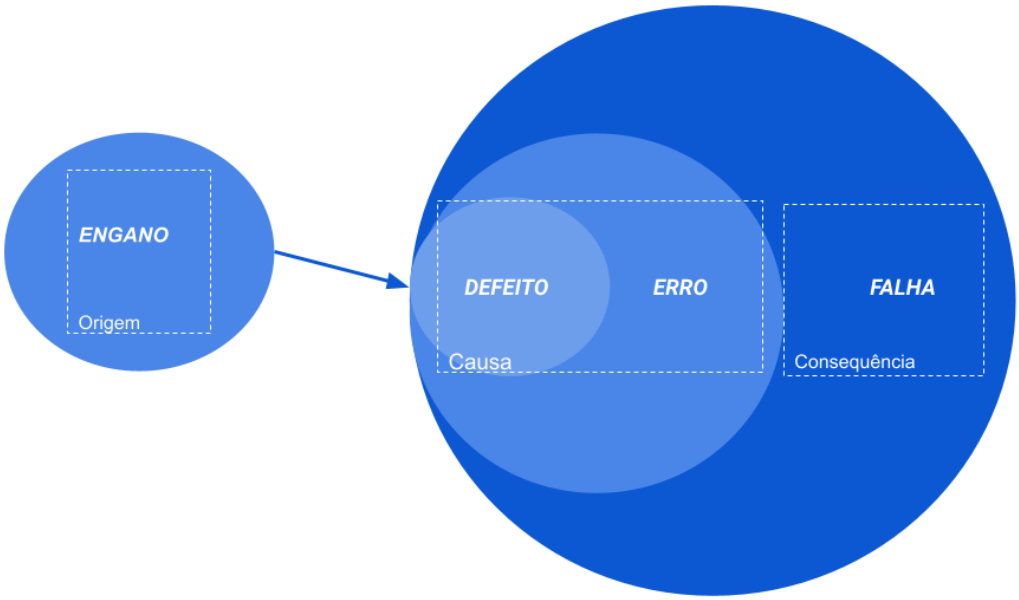
Vamos compreender então essa diferenciação:

1. **Engano:** o engano é uma ação humana acidentalmente inclusa dentro de uma classe, uma função, ou outro elemento qualquer de um código de programa. Por exemplo, numa fórmula de cálculo de impostos sobre vendas, podemos incluir a fórmula de forma errada ou até mesmo a fórmula errada;
2. **Defeito**: o defeito é a consequência de um engano cometido em um trecho de código de programa, o qual resulta em saídas inesperadas ou inconsistentes;

3. **Erro**: o erro então se forma devido a um defeito causado por sua vez por um engano. E neste caso, o resultado do *software* é diferente do esperado;
4. **Falha**: a falha é uma consequência de um erro. Por exemplo, voltando ao cálculo dos impostos sobre vendas, que teve um defeito em decorrência da escrita de uma fórmula errada, gerou um erro que causou a falha na nota fiscal.

A Figura 5 demonstra tais conceitos de uma forma mais clara:

Figura 5 – Demonstração da relação entre enganos, defeitos, erros e falhas.



No entanto a ISO 24765 diz que *erro* é uma ação humana que produz um resultado incorreto. *Defeito* é um problema ou falha que, se não for corrigido, pode fazer com que um aplicativo falhe ou produza resultados incorretos. Uma imperfeição ou deficiência em um *software* ou componente do sistema que pode fazer com que o componente não desempenhe sua função, por exemplo, uma definição de dados incorreta ou instrução de código-fonte, segundo a ISTQB 2011 (ISTQB, 2022). Um defeito, se executado, pode causar a falha de um *software* ou componente do sistema. E *falha* é o término da capacidade de um produto de desempenhar uma função requerida ou sua incapacidade de funcionar dentro de limites previamente especificados, conforme a ISO 25010 (ISO, 2011).

Segundo Pressman (2011), *erros* são as causas de problemas de qualidade de *software* e é necessário investigarmos suas causas, com a

finalidade de prevenirmos e não vivermos continuamente corrigindo-os após as ocorrências.

Os erros podem ser divididos em nove tipos principais:

1. **Definição dos requisitos**: as definições incorretas acabam por gerar grande parte dos erros de *software*. Uma definição errada ou incompleta ocorre por falta de requisitos essenciais para construção do *software*;
2. **Falhas de comunicação:** a comunicação é algo essencial em qualquer processo empresarial, e no desenvolvimento de *software* não é diferente. Usuários e equipe técnica precisam construir pontes de comunicação para que os requisitos sejam elaborados levando em consideração as reais necessidades do *software*;
3. **Desvios nos requisitos de *software***: quando os desenvolvedores fogem do contexto dos requisitos, podem gerar problemas no *software*. Por exemplo, a omissão de funcionalidades em decorrência de prazos e orçamentos podem causar a aprovação de funcionalidades incompletas;
4. **Erros de projeto lógico**: definições no uso de bibliotecas erradas podem causar problemas no funcionamento de determinado *software*, e tais erros são introduzidos no *software* que acaba gerando uma sequência de erros;
5. **Erros de codificação**: este é o tipo de erro relacionado à lógica de programação e utilização de estruturas equivocadamente no *software*, podendo provocar a inclusão de erros durante a codificação;
6. **Não conformidade com documentação**: documentação com problemas e não em conformidade dificulta o êxito das equipes de testes, pois geram massas de testes equivocadas;
7. **Falhas no processo de testes**: Planos de testes precisam estar completos, bem como documentações e relatos de erros e falhas. Quando da ocorrência e detecção de falhas, as correções devem ser feitas o mais rápido possível;
8. **Erros de UI (*User Interface*)**: Interfaces mal projetadas, principalmente pela falta de observância de técnicas e heurísticas que conduzam uma interação centrada no usuário, também são causas de problemas no *software*;
9. **Erros de documentação**: erros encontrados na documentação, em manuais do *software*, ou em documentos integrados ao corpo do código são algumas situações desse tipo de erro.

Segundo a classificação anterior dos focos sobre inclusão de erros num *software*. Para que um requisito seja considerado com boa qualidade, ele deveria atender as seguintes características (Laporte, 2018]:

- Estar correto;
- Estar completo;
- Ser claro para cada grupo de partes interessadas (por exemplo, o cliente, o arquiteto do sistema, testadores e aqueles que irão manter o sistema);
- Não ter ambiguidade, ou seja, mesma interpretação do requisito por parte de todas as partes interessadas;
- Ser conciso (simples, preciso);
- Estar consistente;
- Ser viável (realista, possível);
- Ser necessário (responde à necessidade do cliente);
- Ser independente do projeto;
- Ser independente da técnica de implementação;
- Ser verificável e testável;
- Poder ser rastreado até uma necessidade de negócios;
- Ser único.

Os erros também podem ocorrer devido a problemas na comunicação entre o pessoal de *software* e os usuários. Alguns exemplos de má compreensão entre usuários e desenvolvedores podem ser, por exemplo:

- Má compreensão das instruções do cliente;
- O cliente quer resultados imediatos;
- O cliente ou o usuário não se dedique a ler a documentação que lhe foi enviada;
- Má compreensão das mudanças solicitadas aos desenvolvedores durante o projeto;
- O analista deixa de aceitar mudanças durante a fase de definição e desenho dos requisitos, uma vez que para determinados projetos 25% das especificações terão sido alteradas antes do final do projeto.

Outro aspecto de erros também pode ocorrer quando o desenvolvedor interpreta incorretamente os requisitos e desenvolve com base em sua própria interpretação. Desse tipo de situação, os desvios podem ser por:

- reutilizar o código existente sem fazer os ajustes adequados para atender aos novos requisitos;
- decidir abandonar parte dos requisitos devido a pressões orçamentárias ou de tempo;
- iniciativas e melhorias introduzidas por desenvolvedores sem verificação com os clientes.

A arquitetura do *software* também pode inserir problemas aos requisitos serem traduzidos de forma equivocada. Os erros de projeto típicos são:

- Uma visão incompleta do *software* a ser desenvolvido;
- Papel pouco claro para cada componente da arquitetura de *software* (responsabilidade, comunicação);
- Dados primários não especificados e classes de processamento de dados;
- Um projeto que não usa os algoritmos corretos para atender aos requisitos;
- Sequência incorreta de processos de negócios ou técnicos;
- Desenho deficiente de critérios de regras de negócios ou processos;
- Um projeto que não rastreie os requisitos;
- Omissão de *status* de transação que representem corretamente o processo do cliente;
- Falha no processamento de erros e operações ilegais, o que permite que o *software* processe casos que não existiriam no setor de negócios do cliente – estima-se que até 80% do código do programa processe exceções ou erros.

Muitos erros podem ocorrer na construção do *software*. Erros podem ser por ineficiências comuns de programação, como:

- Escolha inadequada de linguagem de programação e convenções;
- Não abordando como gerenciar a complexidade desde o início;
- Má compreensão/interpretação de documentos de projeto;
- Abstrações incoerentes;
- Erros de *loop* e condição;
- Erros de processamento de dados;
- Erros de sequência de processamento;

- Falta ou validação deficiente dos dados na entrada;
- Desenho deficiente de critérios de regras de negócios;
- Omissão de status de transação que são necessários para representar verdadeiramente o processo do cliente;
- Falha no processamento de erros e operações ilegais, o que permite ao *software* processar casos que não existiriam no setor de negócios do cliente;
- Má atribuição ou processamento do tipo de dados;
- Erro no *loop* ou interferência no índice do *loop*;
- Falta de habilidade para lidar com ninhos extremamente complexos;
- Problema de divisão de inteiros;
- Má inicialização de uma variável ou ponteiro;
- Código-fonte que não remonta ao *design*;
- Confusão sobre um alias para dados globais (variável global passada para um subprograma).

A falta de documentação do *software* também pode ser causa de problemas, tais como:

- Quando os membros da equipe de *software* precisam coordenar seu trabalho, eles terão dificuldade em entender e testar software mal documentado ou não documentado;
- A pessoa que substituir ou manter o *software* terá apenas o código-fonte como referência;
- A garantia da qualidade encontrará um grande número de não conformidades (no que diz respeito à metodologia interna) em relação a esse *software*;
- A equipe de teste terá problemas para desenvolver planos e cenários de teste, principalmente porque as especificações não estão disponíveis.

Observamos que há várias origens que por algum tipo de engano pode acarretar defeitos contendo erros e que fazem com que o *software* falhe. Não é tarefa fácil a garantia de que tais problemas possam ser prevenidos, porém, com a inclusão de atividades relacionadas à verificação (durante o processo de desenvolvimento) e validações (pós desenvolvimento de *software*) podem gerar bons resultados tanto a nível de *software* quanto de processo de desenvolvimento.

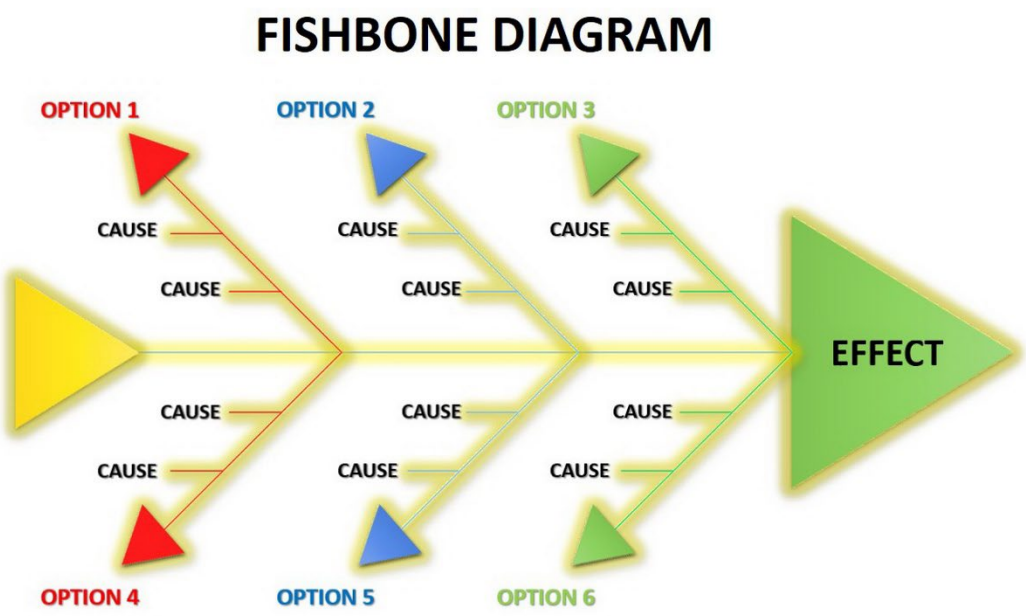
TEMA 5 – CAUSA RAIZ E SEUS EFEITOS

A RCA (*Root Cause Analysis*), segundo Latino (2011), é uma das ferramentas mais valiosas para qualquer organização. Isso é especialmente verdadeiro para grandes empresas com uso intensivo de ativos. Existem muitos problemas que surgem e, se não houver um plano para lidar com eles, a instalação pode se tornar muito reativa.

O desafio de uma RCA eficaz é: quando aplicamos os recursos para identificar as causas-raiz de um problema? Há simplesmente muitos problemas que surgem para resolver todos de forma eficaz. Portanto, uma abordagem mais inteligente deve ser adotada para selecionar os problemas certos a serem resolvidos. Muitas vezes vemos organizações lutando com quais falhas analisar usando RCA. Outras vezes, o trabalho de análise é limitado a questões regulatórias como segurança e eventos ambientais. Em outro momento, os problemas relacionados a equipamentos ou processos são simplesmente corrigidos e o processo é reiniciado sem saber a causa. Sem identificar e abordar as várias causas-raiz, o problema provavelmente se repetirá. Parece que sem algum tipo de pressão externa para realizar uma análise isso simplesmente não acontece. Portanto, uma estratégia deve ser empregada para direcionar o pessoal sobre o que e quando fazer RCA. Quando esses problemas ocorrem, eles são considerados muito importantes e devem ser abordados. Precisamos de alguma maneira de separar a emoção do fracasso do dia do que é realmente importante para o sucesso da instalação. Portanto, precisamos determinar quais são as perspectivas, objetivos e medidas para a organização.

Até aqui falamos de forma genérica, utilizando RCA, ou Diagrama Espinha de Peixe (Figura 6) para quaisquer tipos de situações. No entanto, pensando em metodologias ágeis no gerenciamento de desenvolvimento de software, podemos utilizar essa técnica para questões relacionadas à prática de DevOps, por exemplo. Mitigar os riscos do gerenciamento do ciclo de vida da aplicação, na gestão de configuração, integração contínua, estratégia de testes, *build* e *release* de *pipelines*.

Figura 6 – Exemplo da estrutura de um Diagrama RCA/Espinha de Peixe



Crédito: Mindroom/Shutterstock.

Neste momento, podemos utilizar o uso do 5-Why. Com apoio do Diagrama de Ishikawa, ou diagrama de causa e efeito, podemos descobrir a causa raiz do *bug* apresentado no *software* em produção.

Uma vez coletados todos os erros do *software* em produção, analisamos a causa raiz do problema, ao agruparmos as causas em grupos. Os erros, uma vez listados e relacionados, atuam na causa e na resolução do problema em definitivo.

Podemos então passar para o desenvolvimento de métricas para uma iniciativa de manutenção e confiabilidade. Frequentemente ouvimos falar de tempo médio entre falhas (MTBF), tempo médio de restauração (MTTR) e muitas outras medições. Medir o desempenho por medir não é especialmente útil, a menos que as medições estejam diretamente relacionadas ao desempenho da organização e ações sejam tomadas para fazer as melhorias necessárias quando as medidas estiverem indo em uma direção negativa. Quais metas ou objetivos estamos tentando atingir antes de podermos determinar quais medidas precisamos monitorar. Uma metodologia eficaz para determinar os objetivos da sua empresa é criar um *mapa estratégico*.

Um mapa estratégico pega todos os objetivos da empresa e os coloca em várias perspectivas. Trata-se de uma introdução ao processo de trabalho de

análise de causa raiz (RCA). É sempre importante lembrar que não podemos melhorar aquilo que não conseguimos medir. Com base na lista de defeitos, erros e problemas, podemos criar KPis para auxiliar na medição desses problemas.

Algumas questões podem ser levadas em consideração para criação do diagrama de causa e efeito:

1. Listar os defeitos encontrados no ambiente;
2. Listar os problemas dos testes;
3. Planejar os próximos passos para realização de testes funcionais ou de regressão;
4. Listar *Stories* que não estavam como *ready*;
5. Listar ambiguidade em relação às regras de negócio;
6. Listar problemas com cronograma;
7. Listar pontos de questões sobre problemas na concepção ou nos requisitos.

Todos os planos de ação propostos após a implementação do diagrama de causa e efeito devem ser revistos e avaliados por todos os integrantes do time.

FINALIZANDO

A garantia da qualidade de *software* está intimamente relacionada a atividades de testes. Podem ser testes, depurações, validações, verificações, auditorias ou inspeções. O gerenciamento da qualidade de *software* preocupa-se em garantir que o número de erros caia sensivelmente e que a carga de trabalho em manutenções corretivas também tenha um declínio considerável. Eliminarmos problemas nas etapas de requisitos, análise e projetos podem diminuir em muito os problemas que possam ocorrer nas fases finais do ciclo de vida.

Não podemos esquecer que testes envolvem pessoas com muitos papéis: desenvolvedores, engenheiros de *software*, analistas de sistema, analistas de testes, gerentes de projetos, pessoal de dados e, principalmente, os clientes ou amostragem dos clientes.

As falhas de *software*, dependendo da aplicação deste, pode levar uma empresa a perder dinheiro, tempo, reputação empresarial, prejuízo e até mortes,

se o *software* for direcionado para produtos que interajam diretamente com os clientes.

O teste insuficiente ou o tipo errado de teste feito pode custar muito mais cara para uma empresa do que todo o planejamento e implantação de garantia da qualidade com o envolvimento de testes de *software*.

REFERÊNCIAS

BROWN, E. **Web development with Node and Express**. Sebastopol, CA: O'Reilly Media, Inc., 2014.

HAMBLING, B. et al. **Software testing**: an ISTQB-BCS certified tester foundation guide. 3. ed. Swindon, UK: BCS Learning & Development Ltd., 2015.

ISO. ISO 8402. Quality management and quality assurance — Vocabulary. **ISO**, 1994. Disponível em: <https://www.iso.org/standard/20115.html>. Acesso em: 7 out. 2022.

_____. ISO 9000 family. Quality management. **ISO**, 2015. Disponível em: <https://www.iso.org/iso-9001-quality-management.html>. Acesso em: 7 out. 2022.

_____. ISO/IEC/IEEE 24765. Systems and software engineering – Vocabulary. **ISO**, 2017. Disponível em: <https://www.iso.org/standard/71952.html>. Acesso em: 7 out. 2022.

_____. ISO/IEC 25010. Systems and software engineering – Systems and software Quality Requirements and Evaluation (SQuaRE) – System and software quality models. **ISO**, 2011. Disponível em: <https://www.iso.org/standard/35733.html>. Acesso em: 7 out. 2022.

ISTQB. Certified Tester Foundation Level (CTFL). **ISTQB**, 2022. Disponível em: <https://www.istqb.org/certifications/certified-tester-foundation-level>. Acesso em: 7 out. 2022.

LAPORTE, C; APRIL, A. **Software quality assurance**. Wiley: IEEE Press, 2018.

LATINO, R. **Root cause analysis**: improving performance for bottom-line results. CRC Press, 2011.

LEWIS, W. E. **Software testing and continuous quality improvement**. 3. ed. Boca Raton, FL: Taylor & Francis Group, LLC, 2009.

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

SOMMERVILLE, I. **Engenharia de *software***. São Paulo. Pearson Education do Brasil, 2018.

WOOD, B.; AIKEN, H.; BROOKS JR., F. P. The origens of the computer science. **IBM**, S.d. Disponível em: <https://www.ibm.com/ibm/history/ibm100/us/en/icons/compsci/>. Acesso em; 7 out. 2022.