



# Desenvolvimento de Sistemas

---

## Tipos de teste

Testar um *software* é um processo organizado e planejado. Além disso, não é um processo único, pois tem várias técnicas e características. Com a evolução na engenharia de *software* e do desenvolvimento em si, novos tipos de teste foram sendo pensados de maneira que cubra os mais diversos aspectos de um *software* – desde sua interface e a interação com o usuário até as questões de segurança, passando por questões de desempenho e de garantia da qualidade de *software*.

A seguir, serão explorados vários dos tipos de testes praticados no desenvolvimento de sistema. Vale ressaltar que nem todos os projetos utilizarão todos os tipos de teste; cabe ao time de desenvolvimento e de testes identificar aqueles mais adequados às necessidades do projeto, à realidade da equipe e às exigências do cliente.

## Teste manual e automatizado

Os testes de *software* podem ser realizados de duas maneiras: manualmente ou automaticamente, com o apoio de uma ferramenta de automação.

O teste manual é realizado por pessoas, geralmente um analista ou especialista em testes. A pessoa deverá testar passo a passo os casos de teste, com bastante atenção às condições que o teste impõe, e verificar seus resultados. Há um aspecto subjetivo que pode ser positivo no teste manual, já que são seres humanos avaliando o sistema. Essa subjetividade pode revelar outros problemas com baixa usabilidade ou falhas não observadas anteriormente no sistema.

Geralmente é um teste de baixo custo e que não depende de grandes configurações anteriores. Por outro lado, são testes mais lentos e, pela mesma subjetividade que pode ajudar, há boa chance de algum problema passar despercebido por quem testa.



Figura 1 – Benefícios do teste manual

Fonte: adaptado de Global App Testing (c2023)

Os testes automatizados, por sua vez, são realizados por *softwares* ou bibliotecas de linguagem que exigem uma configuração inicial (*scripts*, códigos ou passos configurados) para a execução de um teste. Após essa configuração, o teste pode ser executado e reexecutado com muita agilidade e sem a necessidade da interferência de uma pessoa, mostrando, na ferramenta, se o resultado esperado para uma funcionalidade testada foi alcançado ou não.

Esses testes são bastante lógicos: se o teste produzir determinado valor ou atingir resultado esperado, então ele passou; se o teste produzir valor ou resultado divergente do esperado, então ele falhou. Por isso, são muito confiáveis, sem risco de interferência em razão

de subjetividades. Podem ser mais caros, pois dependem de ferramentas específicas, e não detectarão problemas que pessoas poderiam observar, como de usabilidade. Também levam mais tempo (e exigem mais *expertise*) para serem configurados ou programados, mas, uma vez que isso é feito, a execução dos testes é muito mais veloz que o teste manual.



Figura 2 – Benefícios do teste automatizado

Fonte: adaptado de Testinium (2021)

É comum (e desejável) que ambos os tipos de teste sejam utilizados no processo de teste de um *software*. Algumas funcionalidades e alguns aspectos do sistema se beneficiarão mais de testes manuais (como nos testes de aceitação, por exemplo, como você verá a seguir), outras são mais beneficiadas com testes automatizados (em testes de regressão, por exemplo). Também se deve levar em consideração a *expertise* da equipe e o tempo disponível para testes. No entanto, em projetos modernos, é recomendável que ao menos um tipo de teste automatizado seja aplicado ao projeto.

## Teste funcional

Testes funcionais são aqueles que verificam a funcionalidade e as características de um sistema sem adentrar em questões técnicas, como código-fonte ou estruturas de banco de dados. Por essa razão, também é denominado teste de caixa-preta (em que se vê o exterior do

programa, mas não o interior). O foco nesse tipo de teste fica nos valores informados como entrada de dados e saídas obtidas pelo programa, comparando-as com o que se esperava obter.

Dentro do teste funcional ainda há técnicas próprias que podem ser aplicadas. Mais detalhes podem ser obtidos no conteúdo **Técnicas de teste** desta unidade curricular.

## Teste de regressão

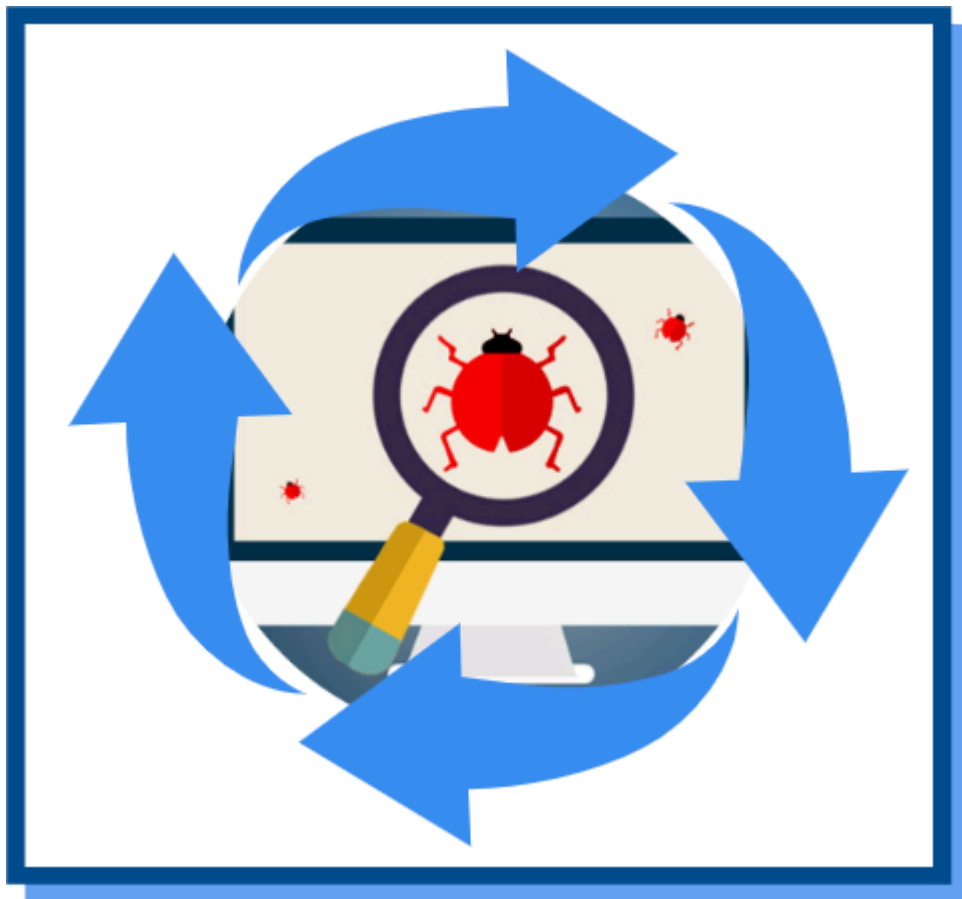


Figura 3 – Testes de regressão

Fonte: Pixabay (2015)

Alterações em código-fonte, correções aplicadas ou funcionalidades novas implementadas em um sistema podem trazer efeitos colaterais no *software*, trazendo falhas ao que já estava funcional anteriormente. Testes de regressão são usados para garantir que alterações em um *software* não afetem negativamente o que já foi testado e está funcionando bem. Procura, portanto, assegurar que o sistema não regrediu em questão de qualidade, mantendo-o confiável.

A forma mais simples de teste de regressão é reexecutar uma bateria de testes manuais previstas nos planos de teste do projeto. Entretanto, em razão de restrições de projeto, de equipe, tempo e ferramentas, pode ser que alguns cenários acabem ficando de fora, o que

pode trazer riscos ao projeto. Isso porque quanto mais o *software* se expande, mais casos de teste vão surgindo, tornando o processo de teste mais oneroso.

Outra alternativa, em muitos casos mais eficiente, é a automatização dos testes, ou seja, usar *softwares* ou bibliotecas de linguagem de programação que permitam a criação de casos de teste e a execução automática desses casos dentro do *software*. Entre essas ferramentas está o teste unitário (em Java, utiliza-se JUnit para isso). Uma vez configurada a ferramenta e desenvolvidos os casos de teste, basta acionar sempre que houver uma mudança no código ou agendar a execução periódica dos testes e aguardar os resultados para verificar se houve alguma falha. Esse tema será abordado com mais detalhe no conteúdo **Testes unitários automatizados** desta unidade curricular.

De forma geral, o mais adequado é que alguns testes sigam sendo realizados manualmente enquanto outros são automatizados.

## Passos para realizar testes de regressão

1. Coletar casos de testes que deverão ser reexecutados. Incluir casos de teste que cubram áreas mais propensas a erros no sistema, que sejam mais vulneráveis a mudanças, que testem funções principais do sistema e que sejam mais complexos, como sequências de eventos em interface visual.
2. Estimar o tempo de execução dos testes. É importante levar em consideração os critérios, o plano e a revisão dos casos de teste para calcular o tempo necessário para testes de cada funcionalidade escolhida.
3. Separar os testes que podem ser automatizados. A tendência pode ser de automatizar a maioria dos testes, mas algumas verificações mais complexas, como séries de ações encadeadas em interface visual, podem exigir testes manuais.
4. Priorizar os casos de teste. Recomenda-se que a prioridade sejam funções básicas e centrais do sistema; em seguida, as funções cruciais, mas não centrais, ao *software*; por fim, funções periféricas, mas que possam evitar problemas técnicos e complicações ao desenvolvimento.
5. Aplicar ferramentas para acelerar o processo de teste.

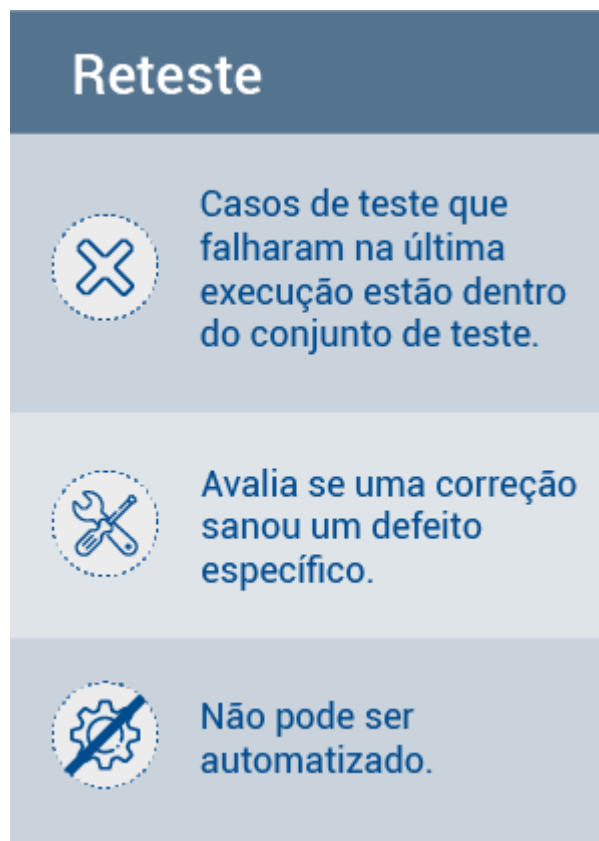
# Teste de confirmação



O teste de confirmação é aquele realizado após a correção de um problema verificado no sistema, possibilitando ao testador confirmar que o problema já não ocorre mais ou para retornar ao time de desenvolvimento caso o problema persista. Também é conhecido como reteste e ocorre regularmente nos *sprints* do projeto usando método ágil.

Não há uma técnica específica. Aplica-se, aqui, teste manual ou automatizado. O único ponto é que o testador precisa executar exatamente o mesmo caso de teste que evidenciou o erro (não é necessário criar um novo caso de teste).

Teste de confirmação não é igual a teste de regressão. Enquanto os testes de confirmação são aplicados especificamente a defeitos recém corrigidos, os testes de regressão são mais abrangentes e acontecem após testes de confirmação.



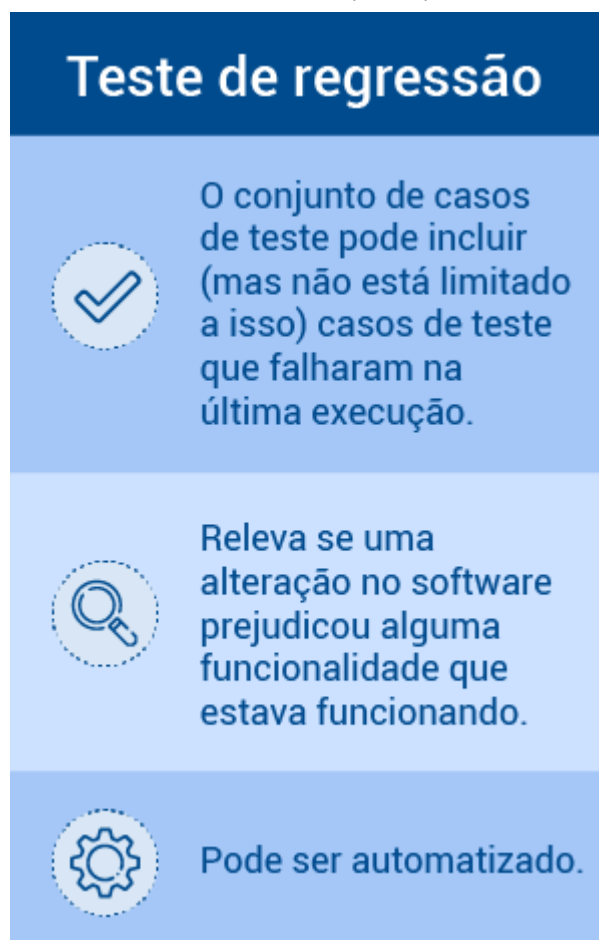


Figura 4 – Reteste (teste de confirmação) *versus* teste de regressão

Fonte: adaptado de Grimms (c2023)

## Teste de aceitação

Conceitualmente próximo dos testes funcionais, os testes de aceitação estão mais ligados aos clientes e envolvem os usuários finais do *software* para a validação das funcionalidades e dos requisitos implementados no sistema.



Figura 5 – Teste de aceitação verifica o que está de acordo ou não com o que o usuário final necessita

Fonte: Pixabay (2014)

Geralmente são testes elaborados com base em histórias do usuário. O teste de aceitação, que parte da visão do cliente sobre o sistema, diferencia-se, assim, dos demais testes de sistema, que partem da visão dos desenvolvedores sobre os requisitos.

Em metodologias mais antigas de desenvolvimento *software*, geralmente seriam testes realizados apenas no final do desenvolvimento e antes da implantação. Com os métodos ágeis e uma proximidade maior do cliente, geralmente acontecem em ciclos (no Scrum, testes de aceitação podem ser realizados nas reuniões de revisão dos *sprints*).

Há três estratégias principais para teste de aceitação. Veja, a seguir, quais são elas.

## Testes formais



São elaborados com base em planos de testes pensados com o mesmo detalhamento dos testes de sistema. Podem ser automatizados, executados pelo time de teste ou executados por representantes do cliente.

## Testes informais ou testes alfa

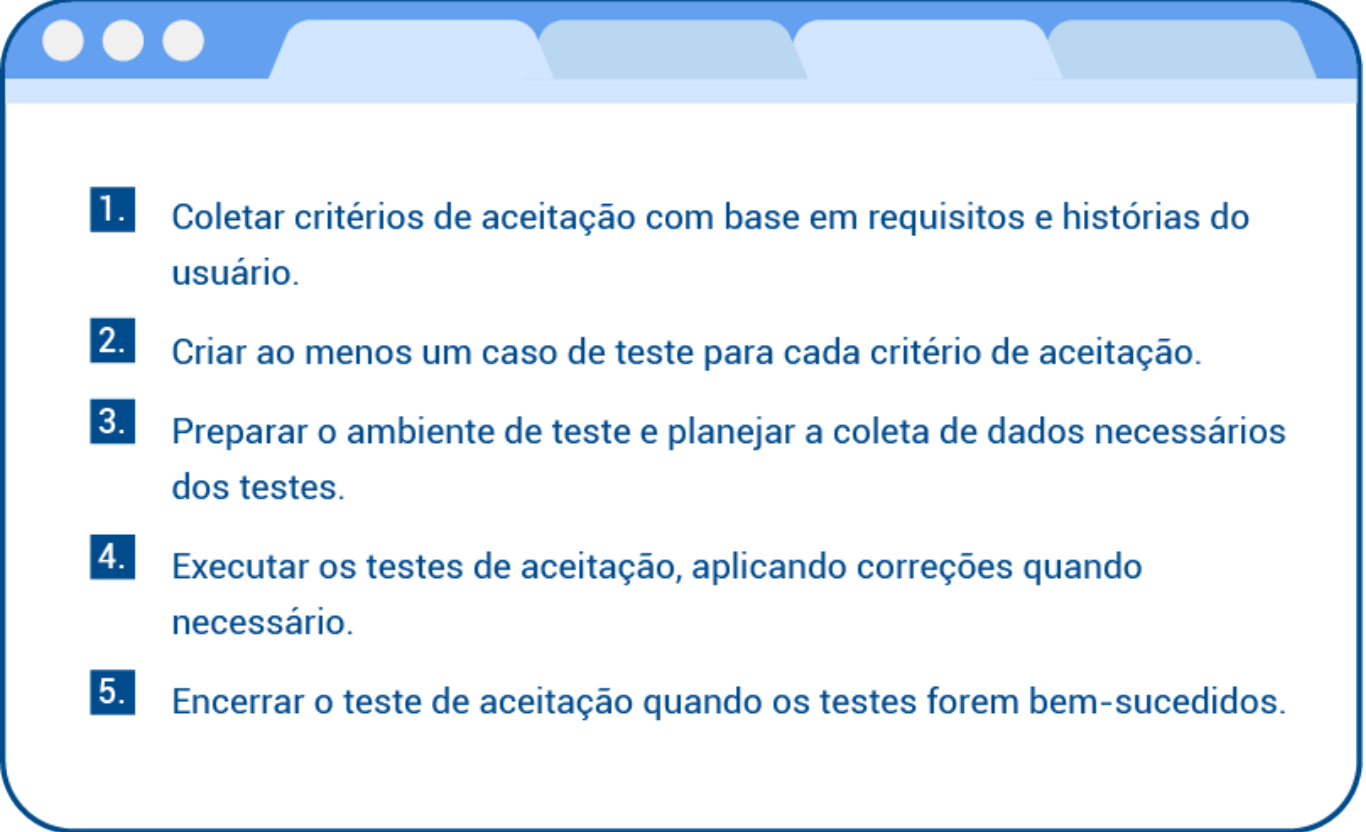
Geralmente são executados pelo cliente. São testes sem um roteiro definido (as funcionalidades do sistema a serem exploradas são identificadas e documentadas, mas não são desenvolvidos casos de testes para serem seguidos). Os usuários testadores exploram o sistema, executando tarefas cotidianas e observando inconsistências.

## Testes beta

Nesta modalidade, um número maior de potenciais usuários (não necessariamente ligados ao cliente) fazem testes livremente. É um tipo de teste bastante aplicado em *softwares* como o WhatsApp, que disponibiliza versões exclusivas do aplicativo com novas funcionalidades a usuários cadastrados. Com base nos dados coletados do uso desses testadores, a equipe pode ajustar ou remover funcionalidades do sistema.

## Como realizar teste de aceitação

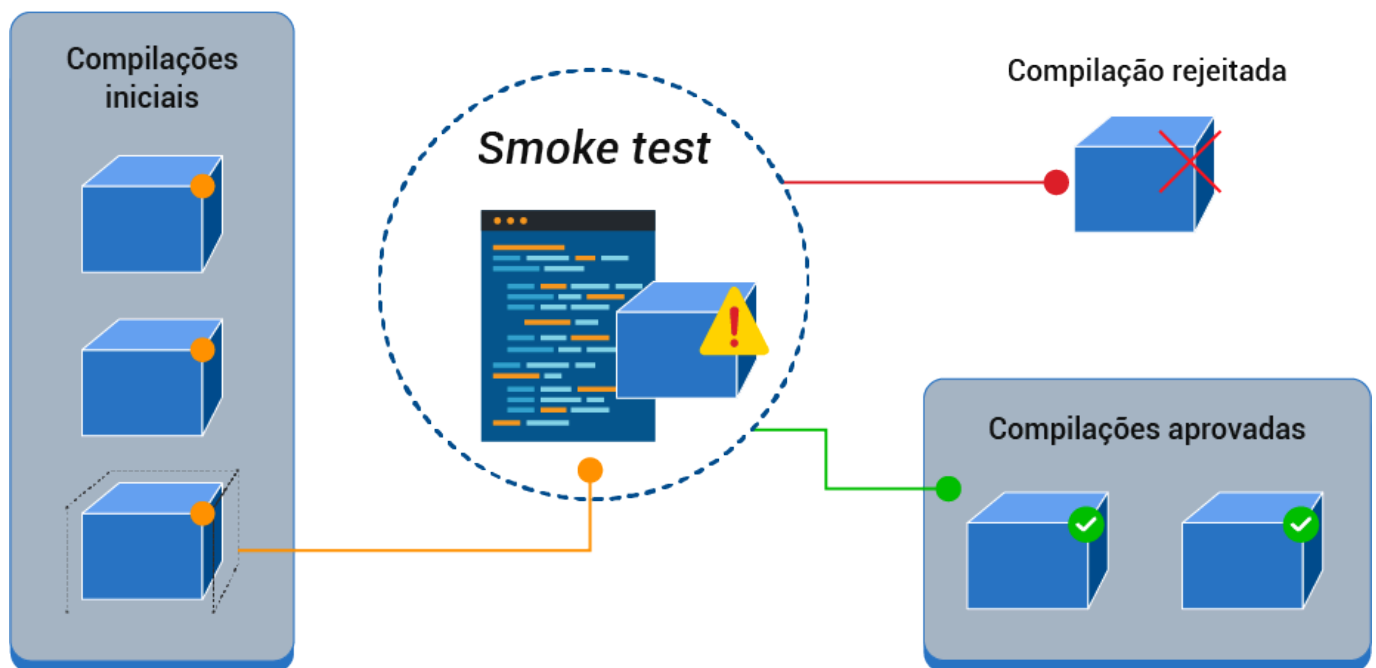
Os passos gerais para realizar atividades de teste de aceitação são:

- 
1. Coletar critérios de aceitação com base em requisitos e histórias do usuário.
  2. Criar ao menos um caso de teste para cada critério de aceitação.
  3. Preparar o ambiente de teste e planejar a coleta de dados necessários dos testes.
  4. Executar os testes de aceitação, aplicando correções quando necessário.
  5. Encerrar o teste de aceitação quando os testes forem bem-sucedidos.

## Smoke test

Os *smoke tests*, ou testes de fumaça, são testes rápidos que verificam as funcionalidades básicas de um sistema. Esse teste deve determinar se o *build* (a compilação) do sistema está gerando um *software* estável ou não. É semelhante a um teste de aceitação, mas de maneira bastante resumida, devendo se ater aos recursos mais importantes da aplicação, ou seja, é um teste mínimo executado rapidamente após um *build*.

Sempre que novas funcionalidades são desenvolvidas e integradas ao sistema, recomenda-se a aplicação do *smoke test*. O objetivo é informar problemas com agilidade e bloquear a versão, caso necessário.

Figura 6 – Fluxo do *smoke test*

Fonte: adaptado de Martins (2021)

Os testes podem ser manuais ou automatizados. No caso do teste manual, o time de testadores recebe a nova versão compilada do *software* e executa os casos de teste de alta prioridade. Se eles falharem, a versão é rejeitada e retorna ao time de desenvolvimento para correções. Se os testes passarem, os demais testes funcionais são executados de acordo com o previsto no projeto.

Os testes automatizados podem agilizar bastante o *smoke test*, podendo executar imediatamente antes ou depois da compilação do sistema. Os desenvolvedores podem rapidamente reter a versão para correções, não precisando passar pelo time de testadores.

## Teste exploratório

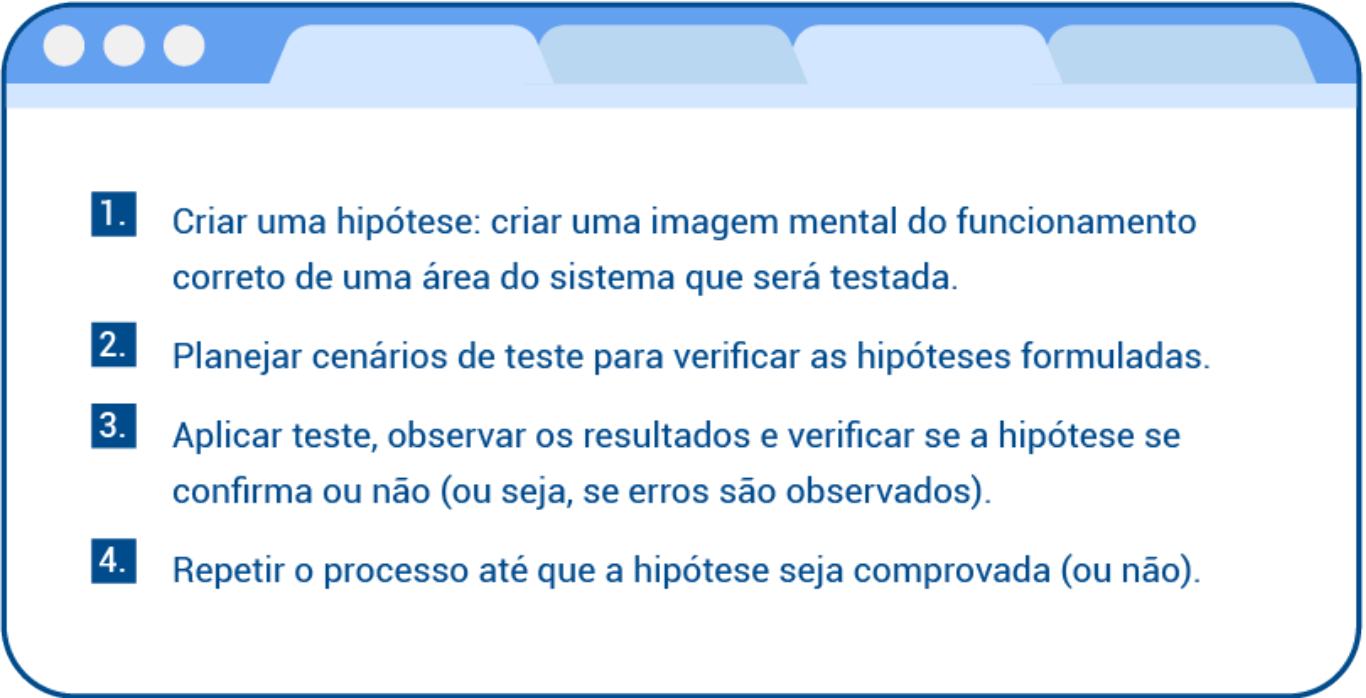
Por definição, nos testes exploratórios, a criação e a execução dos testes acontecem ao mesmo tempo. Ou seja, normalmente são testes que não seguem uma definição formal de planos de testes, mas, sim, a experiência e as observações do testador. Daí o nome do teste: o testador responsável fará uma exploração mais ou menos livre pelo *software* em busca de inconsistências.

Esse tipo de teste pode ser interessante para casos em que não se tem um conhecimento prévio sobre o sistema; em que não há requisitos formais documentados; em que há pouco tempo disponível; em que seja necessário identificar passos de um defeito aleatório do sistema,

em projetos que aplicam metodologias ágeis, que dispensam formalidades e documentações. Também pode ser aplicado como apoio à definição formal de casos de teste no sistema, podendo auxiliar a criação ou expansão de casos de teste do sistema.

Apesar de ser um processo com mais liberdade, a exploração não acontece de maneira aleatória. Geralmente, esses testes iniciam pelo questionamento sobre qual é o teste mais importante que se deve realizar no momento e o planejamento do teste desdobra-se disso.

Uma proposição para o processo de teste exploratório é a seguinte:

- 
1. Criar uma hipótese: criar uma imagem mental do funcionamento correto de uma área do sistema que será testada.
  2. Planejar cenários de teste para verificar as hipóteses formuladas.
  3. Aplicar teste, observar os resultados e verificar se a hipótese se confirma ou não (ou seja, se erros são observados).
  4. Repetir o processo até que a hipótese seja comprovada (ou não).

## Teste de desempenho

Um teste geralmente relegado ao final do processo de desenvolvimento de *software*, mas que deveria estar presente desde o início do projeto, é o de desempenho, uma modalidade que avalia tempo de processamento, velocidade de transferência de dados, eficácia de consulta em banco de dados, número máximo de usuários conectados, uso de processador e memória da máquina, uso de rede, tempo de resposta a comandos do usuário, entre outros aspectos não funcionais do sistema de *software*.



Figura 7 – Testes de *performance* verificam a eficiência do programa

Fonte: Okeke (2022)

Veja algumas razões para realizar testes de desempenho:

- ◆ Verificar se a aplicação satisfaz requisitos não funcionais definidos no projeto.
- ◆ Encontrar gargalos no sistema, ou seja, momentos específicos em que o sistema demora mais que o normal para responder.
- ◆ Verificar a estabilidade do sistema sob pico de acessos.

No contexto de desenvolvimento ágil, o teste de desempenho pode ser aplicado regularmente a cada *sprint* ou em momentos adequados para isso.

O processo de teste de desempenho variará bastante de empresa para empresa e de projeto para projeto. Algumas técnicas como teste de estresse, carga e volume, abordadas a seguir, podem fazer parte do processo caso a intenção seja testar, por exemplo, a resiliência de um sistema a acessos simultâneos.

De maneira geral, recomenda-se que primeiro sejam identificados os indicadores não funcionais a serem testados. Com base nisso, descobrem-se quais são as ferramentas adequadas para esse teste (*softwares* de simulação de acesso, por exemplo, poderiam ser usados para verificar número de usuários máximo executando o sistema ao mesmo tempo). Também é importante definir o ambiente (rede, banco de dados, condições de Internet que sejam próximas do cliente, entre outros) onde os testes executarão.

Definem-se, então, os critérios de aceite para os testes (por exemplo, qual é um bom número de usuários conectados simultaneamente) e, com base nisso, estabelecem-se os planos de teste. Com as ferramentas preparadas, então, executam-se os testes, verificando os problemas e realizando os ajustes necessários. É importante que os testes sejam executados repetidas vezes, para confirmar os problemas de desempenho.

## Teste de carga

Esse é um tipo de teste não funcional especialmente interessante para aplicações *web*. O teste de carga estabelece e avalia a capacidade de processamento de um *software* dada uma carga crescente de dados, geralmente conexões de usuários.

Em uma aplicação *web*, o teste de carga pode ser avaliado simulando um número X de conexões inicialmente e verificando a disponibilidade do sistema. Depois, aumenta-se esse número de conexões, revendo como o sistema está operando, e assim sucessivamente até que se alcance um número próximo ao que seria alcançado no dia a dia do sistema ou até que o sistema apresente falha.

Analogamente, seria possível verificar carga de processos de banco de dados, aumentando gradativamente o número de consultas geradas pelo sistema e verificando a estabilidade de sua conexão com o banco de dados.

Com base nos testes de carga, é possível estabelecer a capacidade da aplicação e identificar fatores limitantes, que podem ser de *hardware*, de codificação, de rede, entre outros.

Os resultados desse teste geralmente são representados pelo número de transações simultâneas e pela quantidade de usuários conectados ao mesmo tempo no sistema. Podem indicar problemas não só em código como em infraestrutura (a necessidade de um servidor com maior capacidade de processamento, por exemplo, ou uma conexão à Internet com maior banda).

# Teste de estresse



Enquanto os testes de carga geralmente simulam uma alta demanda cotidiana de um sistema, o teste de estresse (ou teste de esforço) simula uma demanda extraordinária, que poderá acontecer pontualmente, mas que poderá levar à falha do sistema. Esses testes são, portanto, usados para garantir a estabilidade e a confiabilidade de um sistema, simulando uma alta taxa de tráfego de dados e registrando a resposta do *software* a essa demanda. É um teste indicado, por exemplo, para sistemas *web* que, em razão de uma data específica ou um evento, precisam testar seus limites.

Como exemplo, é possível citar um *site* de vendas em uma data de liquidação ou um *site* de inscrições limitadas para um evento. Ambos os casos apresentarão, em um período, uma situação de acesso muito acima do que acontece diariamente. Assim, um teste de estresse pode prevenir que o sistema caia por excesso de acessos.

Além de servir para verificar quanto o sistema suporta de tráfego de dados, o teste de estresse também serve para avaliar como o sistema se comporta após uma falha.

Com base no teste de estresse, assim como no teste de carga, é possível definir ajustes em infraestrutura e no *software* para garantir mais robustez ao sistema.

Algumas métricas podem ser avaliadas baseando-se no teste de estresse:

- ◆ Quantidade de páginas requisitadas por segundo (em sistemas *web*)
- ◆ Taxas de transferência (tamanho dos dados de resposta em sistema *web*)
- ◆ Tempo de carregamento de uma tela ou página
- ◆ Número de falhas na conexão

Uma das ferramentas mais usados para testes de carga e de estresse é o JMeter, em que o desenvolvedor pode configurar cenários de teste e o *software* fará simulação de acessos ao sistema.

## Teste de volume

Mais um teste de desempenho, o teste de volume verifica o comportamento do sistema em situações de grande carga de dados. Diferente dos testes de carga e de estresse, que se preocupam essencialmente com o tráfego de informações e conexões estabelecidas, o teste de

volume trabalha com tamanho de arquivos e bancos de dados usados pelo sistema testado.



Para o teste, estende-se o tamanho do banco de dados ou arquivo até um limite estabelecido e, então, o desempenho do sistema é testado, observando-se o tempo de resposta para as ações do usuário e o comportamento do sistema (se erros acontecem, se as funcionalidades são executadas corretamente).

O teste de volume é interessante porque todo sistema, com o passar do tempo, recebe quantidades de dados que podem diminuir o desempenho do *software*. No tempo de desenvolvimento, os testes funcionais geralmente acontecem sobre pequenos volumes de dados. Assim, o teste de volume pode detectar problemas não observados nos testes comuns. Os dados gerados para o teste geralmente vêm de um *software* desenvolvido para povoar com dados válidos o banco ou os arquivos usados pelo sistema.

Com base no teste de volume, pode-se identificar, então, o ponto em que a estabilidade do sistema diminui e a capacidade de dados da aplicação. Alguns dados podem ser levados em consideração na análise desses testes:

- ◆ Tempo de resposta do sistema (o sistema demora mais, ou não, quando está sob grande volume de dados)
- ◆ Perda de dados (o sistema passa a perder informações quando se trabalha com banco de dados volumosos)
- ◆ Armazenamento e sobrescrita de dados (os dados continuam sendo gravados corretamente, ou não, com um banco de dados muito povoado)

Entre as vantagens do teste de volume, estão a capacidade de detectar melhorias necessárias em infraestrutura e em código para o sistema, a identificação precoce de gargalos de desempenho e a garantia de que o sistema poderá ser usado no mundo real.

## Teste de recuperação

É denominado teste de recuperação o processo de teste não funcional realizado para verificar a resiliência do sistema, se ele consegue se recuperar de falhas ou não. No teste, força-se a ocorrência de uma falha e verifica-se como o sistema se comporta a seguir.

É importante que o sistema seja desenvolvido de tal maneira que, após uma falha, ele possa continuar sua execução tratando essa falha e retornando a um estado consistente anterior do sistema. Essa recuperação deve acontecer em um tempo específico e é essencial



em sistemas críticos como de defesa, dispositivos médicos e outros.



Como exemplo de recuperação, pode-se imaginar uma aplicação baseada em rede, processando envios e recebimentos de dados. Se desconectar o cabo de rede e algum tempo depois restabelecer a conexão, então o *software* deve ser capaz de retomar a transmissão de dados do exato ponto onde parou quando perdeu conexão.

Algumas situações previsíveis podem ser testadas para recuperação:

- ◆ Falha na alimentação de energia na máquina onde o programa está executando (ao retornar, terá salvado as últimas alterações?)
- ◆ O servidor em uma aplicação de cliente-servidor não está acessível (o programa no cliente permitirá executar outras ações localmente?)
- ◆ Dispositivo externo não está respondendo (o sistema trava por causa disso?)

É importante que o testador se prepare para o processo de teste de recuperação:

- ◆ Assegurando-se do ambiente de execução do *software* e da capacidade de expansão (processador, memória etc.)
- ◆ Estudando o impacto e a gravidade de possíveis falhas
- ◆ Observando que o plano de teste contemple testes de recuperação
- ◆ Realizando a manutenção de *backups* do sistema com vários estados do *software* e do banco de dados, para o caso de a falha ser severa a tal ponto que se percam dados
- ◆ Documentando todas as etapas do teste de recuperação

Quando se observa mais de uma falha, ao invés de se ocupar com todas as falhas de uma vez, os testadores devem estruturar a situação, escolhendo e priorizando um segmento de cada vez.

Como vantagens do teste de recuperação, é possível citar: a melhoria geral na qualidade do sistema, a elaboração de planos de recuperação de desastres para o sistema, a eliminação de riscos e o aprimoramento do desempenho geral do sistema, tornando-o mais confiável.

O processo também conta com desvantagens: pode ser demorado e custoso por envolver várias etapas e preparações antes e durante os testes; precisa contar com profissionais treinados e capazes de observar de maneira abrangente as falhas e a recuperação; pode apresentar algumas falhas imprevisíveis ou difíceis de alcançar.

## Teste de segurança

Se um sistema armazena dados sensíveis sobre seus usuários ou sobre dados de negócio do cliente, então a segurança desses dados é um requisito, pois é um potencial alvo de acesso impróprio ou ilegal. Para isso, é importante que o sistema tenha um processo de testes de segurança, que buscam brechas ou vulnerabilidades que podem ser exploradas por *hackers* que tentam invadir o sistema e acessar e/ou corromper dados sensíveis.

O objetivo dos testes de segurança é, portanto, identificar todas as lacunas e as fraquezas de um sistema que possam resultar em perda de informações, prejuízos financeiros e prejuízos na reputação do cliente ou das pessoas envolvidas por dados que param em mãos de terceiros. Tais problemas podem ser solucionados com ajustes no código do sistema ou na infraestrutura.

Vale observar que são testes aplicados para aplicações que estejam conectadas a uma rede (seja interna, seja externa, seja Internet). É um tipo de teste especialmente importante para sistemas *web*, mas que pode (e deve) ser aplicado para sistemas que funcionam em rede local.

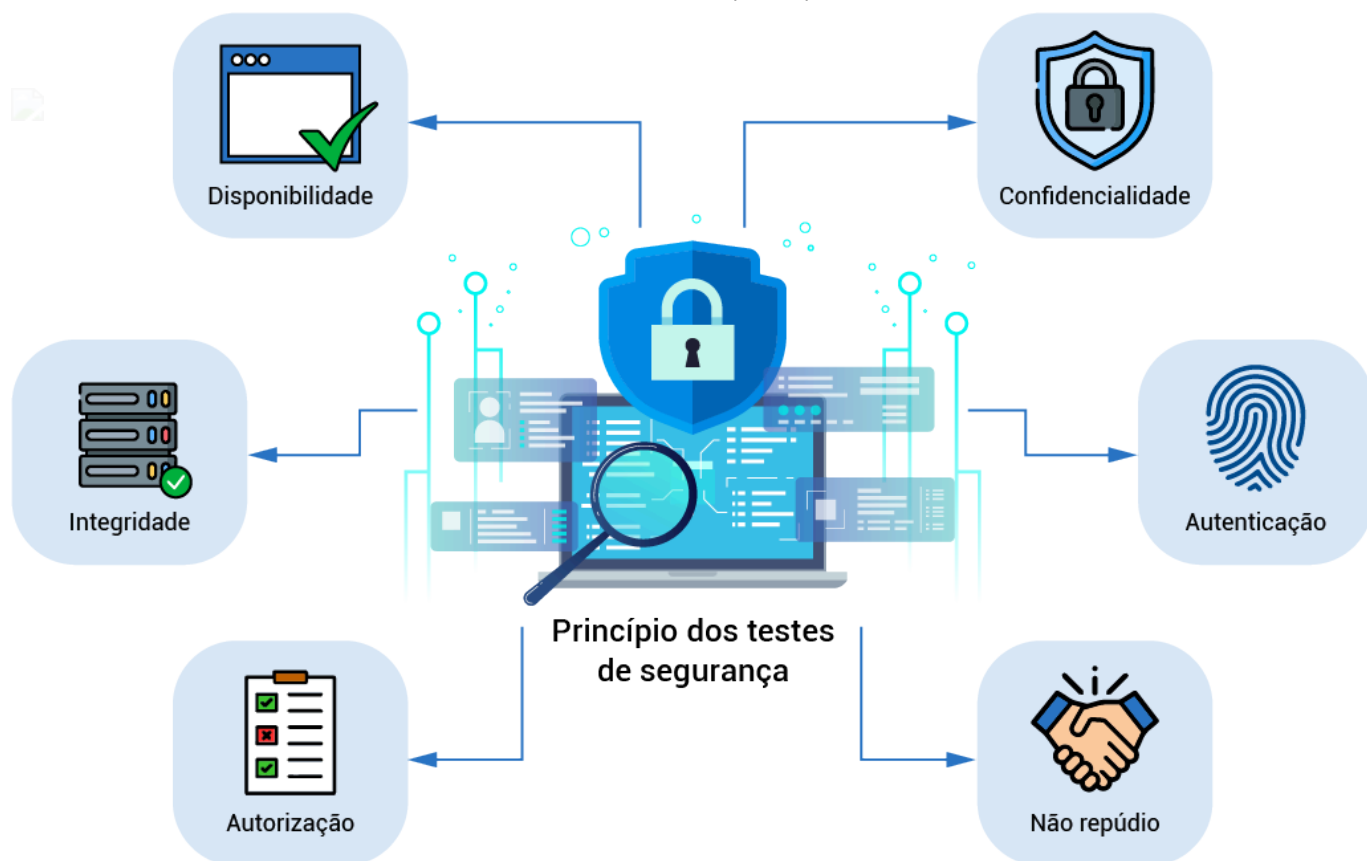


Figura 8 – Princípios dos testes de segurança

Fonte: Senac EAD (2023)

É possível classificar sete tipos de teste de segurança. Veja, a seguir, quais são eles.

## Varredura de vulnerabilidade

Um teste automatizado em que um *software* varre o sistema buscando vulnerabilidades conhecidas. Alguns exemplos de ferramentas usadas são o SecPod SanerNow, o Invicti (ambos pagos) e o Nikto2 (gratuito).

## Varredura de segurança

Pode ser manual ou automatizada e tem por objetivo buscar pontos fracos na rede e no sistema, produzindo soluções a esses riscos.

## Teste de penetração

Busca por vulnerabilidade de acesso externo, simulando ataques *hackers* no sistema. Também é conhecido como *pentest*.

## Avaliação de risco

É uma análise sobre os riscos de segurança observados na organização, classificando-os como baixo, médio ou alto e prevendo medidas para reduzi-los.

## Auditoria de segurança

É uma inspeção interna dos aplicativos, dos sistemas operacionais e das linhas de código do sistema em busca de falhas de segurança.

## *Hacker ético*

Com a ajuda de um hacker contratado para esses testes, realizam-se tentativas sucessivas de invasão ao sistema, expondo falhas de segurança. Também é conhecido como teste *red team*.

## Avaliação de postura

É uma classificação da postura geral da empresa com relação à segurança obtida baseando-se na combinação da varredura de segurança, no hackeamento ético e nas avaliações de risco.

Outra classificação se dá de acordo com o momento em que o teste de segurança ocorre e com a origem das ações. A seguir, conheça o SAST (*static application security test*), o DAST (*dynamic application security testing*) e o IAST (*interactive application security testing*).

### **SAST**

Testes de segurança de aplicativo estático, que acontecem nas etapas de desenvolvimento e, ao encontrarem falhas e vulnerabilidades, passam rapidamente para correção. São testes internos, realizados com base no próprio ambiente de desenvolvimento.

## DAST

Testes de segurança de aplicativo dinâmico, que são realizados antes da liberação do aplicativo ao cliente (seja a entrega de um sistema *desktop*, seja a disponibilização de um sistema *web* ao público). São realizados da parte externa para a interna, ou seja, testa-se em um ambiente externo ao de desenvolvimento, possibilitando a identificação de vulnerabilidades mais visíveis. Também analisam questões de desempenho, como tempo de execução.

## IAST

Testes de segurança de aplicativo dinâmico, que são internos e concedem ao testador ou à ferramenta automatizada de teste um acesso maior aos dados e ao código, permitindo detecções mais precisas de vulnerabilidades. Geralmente acontecem no início do ciclo de vida do projeto.

Veja, a seguir, as técnicas que podem ser aplicadas no processo de teste.

## Acessos à aplicação

Testar as permissões dos diferentes usuários do sistema para verificar se algum perfil está acessando indevidamente recursos que deveriam estar bloqueados a ele. Por exemplo: um vendedor não deve acessar os dados de outros vendedores e informações financeiras em um sistema para uma loja. Para cada permissão, o testador deverá verificar todos os menus, as telas e as funções para detectar se algum recurso está indevidamente acessível.

## Proteção de dados

Além de recursos do sistema, o acesso a determinados dados pode ser restrito a alguns usuários. No exemplo anterior, os dados de salários e faturamento da loja não devem estar ao alcance dos vendedores. Isso pode ser resolvido por meio de permissões no sistema,

possivelmente em nível de banco de dados, mas também com encriptação de dados (para senhas e dados de cartões de crédito, por exemplo).

## Ataques de força bruta

Usar um *software* para, com base em um usuário conhecido, testar senhas uma após a outra até conseguir acesso ao sistema. Geralmente sistemas aplicam uma restrição de quantidade de tentativas de autenticação, inibindo esse tipo de ataque (ao tentar acessar uma rede social, por exemplo, após um número de tentativas malsucedidas, o acesso pode ficar bloqueado por algumas horas e um *captcha* ou uma confirmação de identidade por outro meio pode ser requisitada).

## SQL Injection e XSS (*cross-site scripting*)

Um *script* é usado maliciosamente para tentar invadir ou prejudicar dados do sistema. No caso de SQL Injection, aproveitam-se as vulnerabilidades na implementação de banco de dados, como consultas não protegidas, e usam-se comandos SQL para a invasão. Já o XSS trata de códigos de *script* de uma página que apontem para outra página, com a intenção de obter dados ou acesso a esta.

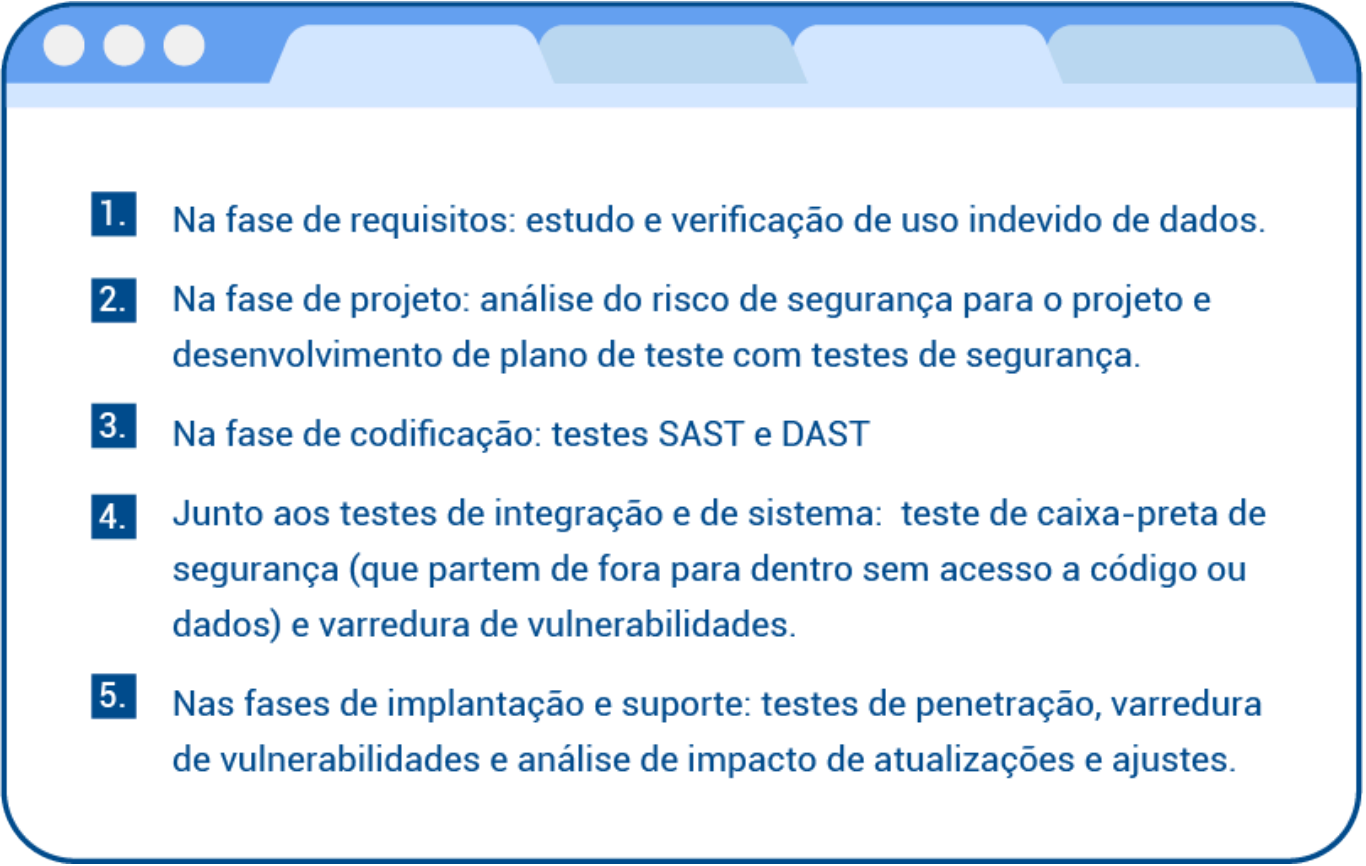
## Pontos de acesso a serviços

Aplicações *web* atuais muitas vezes expõem serviços para que outras aplicações possam acessar. Nesse contexto, é importante testar se apenas as aplicações autorizadas estão tendo acesso aos serviços autorizados e se, com base nesse serviço, não se obtém acesso a outras funcionalidades ou dados que deveriam estar protegidos.

## Manejo de erros

É importante informar ao usuário quando um erro acontece, mas nunca devem aparecer dados detalhados da falha, que podem explicitar detalhes do sistema e podem ser usados por invasores para obter acesso. Assim, uma exceção (e principalmente seu *stack-trace*) não deve ser exposta ao usuário. Ao invés disso, esses detalhes podem ser registrados em arquivo interno de *log*, que deve estar protegido de acesso externo, mas acessível aos desenvolvedores para investigar problemas.

Considerando o processo tradicional de desenvolvimento de *software*, de maneira geral pode-se dizer que os testes de segurança devem percorrer todo o processo de desenvolvimento do sistema.

- 
1. Na fase de requisitos: estudo e verificação de uso indevido de dados.
  2. Na fase de projeto: análise do risco de segurança para o projeto e desenvolvimento de plano de teste com testes de segurança.
  3. Na fase de codificação: testes SAST e DAST
  4. Junto aos testes de integração e de sistema: teste de caixa-preta de segurança (que partem de fora para dentro sem acesso a código ou dados) e varredura de vulnerabilidades.
  5. Nas fases de implantação e suporte: testes de penetração, varredura de vulnerabilidades e análise de impacto de atualizações e ajustes.

No caso de métodos ágeis, os testes de segurança deverão estar presentes em todos os *sprints*, em seu planejamento, na implementação e na finalização do incremento de *software*.