UNIVERSIDADE SÃO JUDAS TADEU – CAMPUS MOOCA

USJT-2025.1-GQS-Lab03-TesteDeSoftware

Prof. Calvetti

Conceitos e estratégias de testes de softwares

Alex Aldib – RA: 823165544

Leonardo Ribeiro de Almeida – RA: 823213510

Guilherme Figueiredo – RA: 823116166

Conceito de Teste

Um **teste de software** é um processo estruturado para verificar se um sistema ou componente atende aos requisitos especificados e funciona conforme esperado. O objetivo principal do teste é **identificar falhas**, **garantir a qualidade** e **minimizar riscos** antes da entrega do software ao usuário final.

Características Fundamentais de um Teste

1. **Executável:** Deve ser possível rodar o teste e verificar os resultados.
2. **Repetível:** O teste pode ser executado diversas vezes sob as mesmas condições.
3. **Mensurável:** Os resultados devem ser analisáveis para determinar se houve sucesso ou falha.
4. **Objetivo:** Focado em um critério específico, como funcionalidade, desempenho ou segurança.

Objetivo do Teste

- Detectar defeitos no software.
- Verificar se os requisitos do sistema foram atendidos.
- Garantir que novas funcionalidades não quebrem as antigas (testes de regressão).
- Avaliar o desempenho, segurança e usabilidade do sistema.

Os testes podem ser realizados de forma **manual** (por testadores humanos) ou **automatizada** (por scripts e ferramentas especializadas).

Estratégias de teste

As **estratégias de teste de software** são abordagens estruturadas que definem **como**, **quando** e **com quais recursos** os testes serão conduzidos ao longo do desenvolvimento do software. Elas servem como um guia para garantir que os testes sejam **eficazes, organizados e alinhados aos objetivos do projeto**.

Uma boa estratégia de teste deve ser:

- **Bem definida** – Deve especificar os passos necessários para validar o software.
- **Flexível e adaptável** – Deve se ajustar a diferentes tipos de projetos e metodologias.
- **Planejada e sistemática** – Deve seguir um cronograma estruturado.
- **Baseada em riscos** – Foca nas áreas mais críticas do software para evitar falhas graves.

Tipos de Estratégias de Teste

As estratégias podem ser classificadas com base na abordagem adotada.

• Estratégia Preventiva (Testes antecipados)

O foco é prevenir defeitos, e não apenas detectá-los.
Os testes começam desde as fases iniciais do desenvolvimento.
Exemplo: **Revisões de código e análise estática de software**.

• Estratégia Reativa (Testes baseados em falhas)

Os testes são projetados **após** o desenvolvimento do software.
Foca na identificação e correção de defeitos antes da entrega.
Exemplo: **Testes exploratórios e testes de regressão**.

• Estratégia Baseada em Riscos

Prioriza testes nos módulos mais críticos do sistema.
Reduz o impacto de falhas em funcionalidades essenciais.
Exemplo: **Testes de segurança em sistemas bancários**.

• Estratégia Baseada em Modelos

Utiliza modelos matemáticos ou de negócio para definir os testes.
Ajuda a prever cenários de uso do software.
Exemplo: **Testes baseados em Casos de Uso e Modelos de Estado**.

• Estratégia de Automação de Testes

Usa ferramentas para executar testes automaticamente.
Reduz o tempo de execução e melhora a eficiência.
Exemplo: **Testes automatizados com Selenium, JUnit ou Cypress**.

As estratégias de teste são fundamentais para garantir **qualidade, segurança e confiabilidade** no software. Aplicar uma abordagem bem planejada reduz riscos e melhora a entrega final.

Conceitos de Verificação e Validação

No contexto da **Qualidade de Software**, os termos **Verificação** e **Validação** são essenciais para garantir que o software atenda aos requisitos e funcione corretamente. Embora muitas vezes sejam usados como sinônimos, eles têm significados distintos.

Verificação

A verificação consiste na avaliação do processo de desenvolvimento para garantir que o software esteja sendo **construído em conformidade com as especificações, normas e diretrizes estabelecidas**. Esse processo tem foco na **qualidade do desenvolvimento** e ocorre ao longo de todas as fases da produção

Características da Verificação:

- Ocorre **durante** o processo de desenvolvimento.
- Assegura que o produto esteja sendo **implementado conforme os requisitos especificados**.
- Utiliza métodos formais, como **revisões técnicas, inspeções e análise estática** do código.

Exemplos de Verificação:

- Revisões e inspeções de documentos de requisitos.
- Análise estática do código-fonte.
- Revisões de arquitetura e design.

Validação

A validação é o processo que garante que o software **atende aos requisitos funcionais e não funcionais esperados pelo usuário**. Esse processo ocorre **após**

a implementação, por meio da execução de testes para verificar se o sistema atende às necessidades do cliente.

Características da Validação:

- Ocorre **após** a fase de desenvolvimento, antes da entrega ao usuário final.
- Garante que o software seja **funcional e atenda às expectativas do usuário**.
- Utiliza **testes funcionais, de aceitação e de usabilidade** para verificar o comportamento do sistema.

Exemplos de Validação:

- Testes de sistema e integração.
- Testes de aceitação do usuário (User Acceptance Testing – UAT).
- Testes de desempenho e usabilidade.

A verificação e a validação são processos complementares que garantem a **qualidade e confiabilidade do software**. Enquanto a **verificação** se concentra na conformidade com os requisitos estabelecidos, a **validação** assegura que o software cumpra seu propósito e atenda às necessidades dos usuários finais.

A correta aplicação dessas práticas reduz falhas, melhora a experiência do usuário e contribui para um desenvolvimento mais eficiente e seguro.

Teste de Software

O teste de software envolve a execução de um programa ou sistema com a intenção de identificar bugs, falhas e defeitos. A principal finalidade dos testes é verificar se o software funciona de acordo com as especificações e requisitos definidos, a**lém de garantir que o sistema atenda aos critérios de qualidade. Testar um software é essencial para validar que ele possui o desempenho esperado em condições normais e extremas de uso**, que ele não contenha falhas críticas e que sua experiência de uso seja satisfatória.

Teste Unitário

O teste unitário é uma técnica fundamental no desenvolvimento de software que se concentra na **verificação de funcionalidades isoladas do código-fonte, geralmente nas menores unidades ou componentes de um sistema**, como funções, métodos ou classes. O objetivo do teste unitário é **garantir que cada unidade do software se comporte como esperado**, antes de ser integrada com outras partes do sistema.

Embora o teste unitário se concentre em componentes pequenos, ele é um pilar importante na construção de software de qualidade, **pois permite detectar falhas cedo no ciclo de desenvolvimento,** o que facilita a correção de erros e a manutenção do código.

1. Teste de Interface

O teste de interface no contexto de testes unitários é focado na interação entre o módulo testado e outras partes do sistema. Embora um teste unitário se concentre em uma unidade isolada, é comum que a unidade interaja com outras partes do código ou com interfaces externas (como bancos de dados ou APIs). Neste caso, o teste verifica se a unidade comunica-se corretamente com esses outros componentes ou interfaces, garantindo que a integração das interfaces seja eficaz.

Exemplo:

- Em uma função que se conecta a um banco de dados, o teste de interface verificaria se a função consegue comunicar-se com a camada de persistência de dados corretamente e retornar os valores esperados.

2. Teste de Estruturas de Dados Locais

O teste de estruturas de dados locais verifica o comportamento das variáveis ou estruturas de dados que são utilizadas dentro da unidade. O objetivo é garantir que a unidade manipule as estruturas de dados de maneira eficiente e correta, considerando todos os tipos de dados e valores possíveis que possam ser manipulados durante a execução da função ou método.

Exemplo:

- Se uma função manipula uma lista, o teste verificaria se a adição, remoção ou alteração de elementos na lista ocorre como esperado, sem causar erros ou inconsistências nos dados.

3. Teste de Condições Limite

O teste de condições limite é utilizado para **verificar como o código lida com valores extremos ou limites dos dados de entrada.** Muitas falhas em software ocorrem quando o sistema é alimentado com valores fora do esperado ou em condições extremas, como valores muito altos ou baixos. O teste de limites garante que o comportamento do sistema seja adequado quando esses valores são processados.

Exemplo:

- Para uma função que calcula o índice de massa corporal (IMC), um teste de condição limite pode verificar se o cálculo é correto quando o peso ou a altura está em seus valores máximos ou mínimos possíveis.

4. Teste de Caminhos Independentes

O teste de caminhos independentes assegura que diferentes caminhos de execução dentro de uma unidade de código sejam testados de forma isolada, sem dependências de outros caminhos. O objetivo é **garantir que todas as possíveis ramificações de um código sejam verificadas**, incluindo as que não são comumente usadas. Isso ajuda a identificar problemas que podem surgir em diferentes condições de execução.

Exemplo:

- Em uma função que usa uma série de instruções condicionais (if, else, switch), o teste de caminhos independentes verificaria cada ramificação dessas instruções, garantindo que todas as condições sejam testadas.

5. Teste de Caminhos de Manipulação de Erro

O teste de caminhos de manipulação de erro **foca na forma como a unidade de software lida com situações excepcionais ou falhas**. Em muitos casos, é importante verificar se o sistema reage corretamente quando ocorrem erros, como falhas de rede, dados incorretos ou outros problemas imprevistos. Este tipo de teste garante que os erros sejam tratados de maneira apropriada, sem causar falhas catastróficas ou **comportamentos inesperados no sistema.**

Exemplo:

- Se uma função faz chamadas para um serviço externo, o **teste de manipulação de erro verificaria se a função lida corretamente com falhas**

na comunicação com o serviço, como ao retornar uma mensagem de erro amigável ou ao tentar uma nova tentativa de conexão.

Teste de Integração

O Teste de Integração é uma fase do processo de teste de software que **verifica a interação entre diferentes módulos ou componentes do sistema**. O objetivo é identificar defeitos nas interfaces e garantir que os módulos desenvolvidos individualmente funcionem corretamente quando combinados.

Esse tipo de teste é fundamental para **evitar problemas de comunicação entre os módulos**, que podem causar falhas inesperadas no funcionamento do software.

Abordagens do Teste de Integração

Existem diversas formas de conduzir testes de integração, cada uma adequada a diferentes contextos de desenvolvimento. As principais abordagens são:

1. Teste de Integração Não Incremental (Big Bang)

- **Todos os módulos do software são integrados simultaneamente e testados como um todo.**

- Essa abordagem pode ser eficiente para sistemas pequenos, mas tem o risco de dificultar a localização de falhas, pois tudo é testado de uma vez.

- Exemplo: Em um sistema de e-commerce, **testar o carrinho de compras, pagamento e envio ao mesmo tempo pode dificultar a identificação de**

um erro específico.

2. Teste de Integração Incremental

- Os **módulos são integrados e testados progressivamente**, o que permite uma identificação mais precisa dos erros.

- Pode ser feito de três maneiras principais:

a) Teste de Integração Ascendente (Bottom-Up)

- Começa pelos módulos de nível inferior (mais básicos) e gradualmente adiciona os de nível superior.

- Utiliza drivers (módulos temporários que simulam as partes superiores que ainda não foram desenvolvidas).

- Exemplo: Em um sistema bancário, **primeiro testam-se os módulos de cálculo de juros antes de integrá-los com a interface do usuário.**

b) Teste de Integração Descendente (Top-Down)

- Começa pelos módulos de nível superior e gradualmente adiciona os módulos inferiores.

- Utiliza stubs (códigos temporários que simulam funcionalidades dos módulos inferiores ainda não implementados).

- Exemplo: Em um aplicativo de pedidos de comida, **primeiro testa-se a interface de usuário antes de integrar funcionalidades** como cálculo de preços e geração de pedidos.

c) Teste de Fumaça (Smoke Test)

- Uma abordagem rápida para verificar se as **principais funcionalidades do sistema estão operacionais.**

- Não realiza **testes detalhados, mas confirma se o software está estável** o suficiente para testes mais aprofundados.

- Exemplo: Em um sistema de login, um teste de fumaça verificaria apenas se um usuário pode acessar a conta sem testar todas as variações possíveis.

Teste de Validação

O Teste de Validação é uma etapa do processo de testes de software que verifica se o sistema **atende aos requisitos especificados e se funciona conforme esperado pelo usuário final.**

Esse teste começa após a conclusão do **teste de integração, garantindo que o software não apenas funcione corretamente**, mas também satisfaça as necessidades do cliente.

Os **critérios de validação** são parâmetros usados para determinar se o software está pronto para ser entregue. Alguns dos principais critérios incluem:

- **Conformidade com os requisitos:** O sistema deve atender às especificações definidas.

- **Funcionalidade esperada:** Todos os recursos devem operar corretamente.

- **Experiência do usuário:** A usabilidade deve ser intuitiva e eficiente.

- **Desempenho adequado:** O software deve responder bem sob diferentes condições de uso.

- **Segurança:** Deve proteger os dados e operações contra acessos não autorizados.

Revisão de Configuração

A revisão de configuração é um processo realizado **antes da validação para garantir que o sistema está na versão correta** e configurado corretamente para os testes. Isso envolve:

- **Verificação de versões do software** e componentes atualizados.
- Conferência da documentação técnica para **assegurar conformidade.**
- Testes preliminares para validar a estabilidade do ambiente de teste.

Testes Alfa e Beta

Os testes de validação incluem duas fases principais:

1. Teste Alfa

- Realizado internamente, geralmente por testadores da equipe de desenvolvimento.
- O objetivo é **identificar problemas críticos** antes de liberar o software para usuários externos.
- Normalmente é feito em um **ambiente controlado.**
- Pode envolver **simulações de uso real,** mas com supervisão dos desenvolvedores.

Exemplo: Em um novo aplicativo de compras online, um teste alfa pode verificar se todas as funções básicas (login, compra, pagamento) funcionam sem erros dentro da empresa.

2. Teste Beta

- Realizado por **usuários reais**, fora do ambiente de desenvolvimento.
- Testa o **software** em condições reais de uso, **identificando** problemas que podem não ter surgido nos **testes anteriores**.
- Os usuários **relatam bugs**, falhas e sugestões para **melhorias** antes do lançamento oficial.

Exemplo: Antes de lançar um novo jogo, a empresa libera um beta para um grupo de jogadores testarem, coletando feedback sobre desempenho e jogabilidade.

Teste de Sistema

O Teste de Sistema é uma fase do processo de testes de software que **avalia o comportamento do sistema como um todo,** garantindo que todos os componentes funcionem corretamente dentro do ambiente operacional previsto.

Diferente dos testes anteriores, que focam em módulos individuais (teste unitário) ou na integração entre eles (teste de integração), **o teste de sistema verifica o software em conjunto com outros elementos do ambiente computacional, como hardware, redes, bancos de dados e outros sistemas externos.**

Características do Teste de Sistema

- O software é apenas um dos elementos do sistema computacional, sendo necessário **verificar sua interação com outros componentes**, como servidores, dispositivos e infraestrutura de rede.
- Os testes de integração de sistema e validação não fazem parte diretamente do processo de desenvolvimento do software, mas são fundamentais para garantir sua funcionalidade dentro de um sistema maior.

- As etapas de desenvolvimento e teste bem planejadas **aumentam a probabilidade de sucesso na integração do software** ao ambiente de produção.

Desafios do Teste de Sistema

1. **Complexidade do Ambiente**

   - Como o software precisa interagir com diversos componentes, é **necessário testar diferentes configurações de hardware,** versões de sistemas operacionais e compatibilidades com outras aplicações.

2. **Problema da "Procura do Culpado"**

   - Quando um erro ocorre, pode ser difícil determinar a origem do problema.

   - O erro pode estar no software, no banco de dados, na configuração do servidor ou em outro componente do sistema.

   - Esse problema g**era desafios na identificação e correção dos defeitos**, exigindo uma abordagem estruturada para análise e depuração.

Etapas do Teste de Sistema

1. **Planejamento do Teste**: Define-se o escopo, os critérios de aceitação e as configurações a serem testadas.

2. **Preparação do Ambiente:** Configuração de servidores, redes e dispositivos para simular o ambiente real de operação.

3. **Execução dos Testes:** Avaliação do comportamento do software sob diferentes cenários e condições de uso.

4. **Análise dos Resultados:** Identificação e correção de falhas encontradas.

5. **Reteste e Validação:** Garante que os erros corrigidos não impactaram outras funcionalidades.

Tipos de Testes de Sistema

Para cobrir diferentes aspectos do funcionamento do software dentro do ambiente computacional, são aplicados diversos tipos de testes:

- **Teste Funcional:** Avalia se o sistema atende aos requisitos especificados.
- **Teste de Desempenho:** Mede tempos de resposta e capacidade de processamento sob carga.
- **Teste de Segurança**: Verifica vulnerabilidades e riscos de acesso não autorizado.
- **Teste de Compatibilidade**: Garante que o software funciona em diferentes dispositivos, navegadores e sistemas operacionais.
- **Teste de Recuperação:** Avalia a capacidade do sistema de se recuperar de falhas.
- **Teste de Usabilidade**: Analisa a experiência do usuário ao interagir com o sistema.

Depuração

A **depuração** é o processo de identificação, análise e correção de erros (bugs) em um software. Ela ocorre após a execução de testes que revelam falhas no sistema, permitindo que os desenvolvedores encontrem e resolvam os problemas antes da entrega final do produto.

O Processo de Depuração

O processo de depuração pode ser dividido em várias etapas:

1. **Identificação do erro**: O erro pode ser detectado durante testes automatizados, manuais ou por meio de relatórios de usuários.
2. **Reprodução do erro**: O desenvolvedor tenta recriar as condições exatas que causam a falha, facilitando a análise do problema.
3. **Diagnóstico**: O código é analisado para determinar a causa raiz do problema, geralmente utilizando ferramentas de depuração e logs.
4. **Correção**: Após identificar a causa, o código é modificado para resolver o erro sem afetar funcionalidades existentes.
5. **Reteste**: O software é testado novamente para garantir que a correção foi bem-sucedida e não introduziu novos erros.

Considerações Psicológicas

A depuração não é apenas um processo técnico, mas também envolve aspectos psicológicos, pois exige concentração, paciência e habilidades analíticas. Alguns fatores psicológicos importantes incluem:

- **Viés do desenvolvedor**: Muitas vezes, os programadores tendem a acreditar que seu código está correto, dificultando a identificação de erros.
- **Frustração**: Encontrar bugs complexos pode ser frustrante, exigindo persistência e pensamento lógico para resolvê-los.
- **Colaboração**: Em muitos casos, a ajuda de outro programador ou a revisão coletiva do código pode facilitar a descoberta do problema.

Estratégias de Depuração

Para tornar a depuração mais eficiente, algumas estratégias podem ser utilizadas:

- **Depuração com Logs**: Adicionar mensagens de saída no código para registrar informações importantes sobre a execução do programa.
- **Uso de Breakpoints**: Ferramentas de depuração permitem pausar a execução do código em pontos específicos para análise do estado do sistema.
- **Divisão e Conquista**: Dividir o código em pequenas partes e testar cada uma separadamente para isolar o erro.
- **Revisão de Código**: Um segundo desenvolvedor revisa o código em busca de erros que podem ter passado despercebidos.
- **Teste de Regressão**: Após a correção, executar novamente os testes para garantir que nenhuma outra funcionalidade foi afetada.

Correção do Erro

Após identificar a origem do erro e aplicar uma correção, algumas práticas devem ser seguidas para evitar a reincidência do problema:

- **Documentação**: Registrar a causa e a solução do erro para consultas futuras.
- **Testes automatizados**: Criar testes que verifiquem se o erro não reaparece após futuras modificações no código.
- **Refatoração**: Melhorar a estrutura do código para evitar problemas semelhantes no futuro.
- **Monitoramento contínuo**: Implementar ferramentas que alertem sobre possíveis falhas em tempo real.

A depuração é um **processo essencial no desenvolvimento de software**, garantindo a qualidade e estabilidade do sistema antes de sua implantação.