DOCUMENTO DE PLANEJAMENTO DE TESTES

Equipe:

508161 – João Paulo Freitas Queiroz

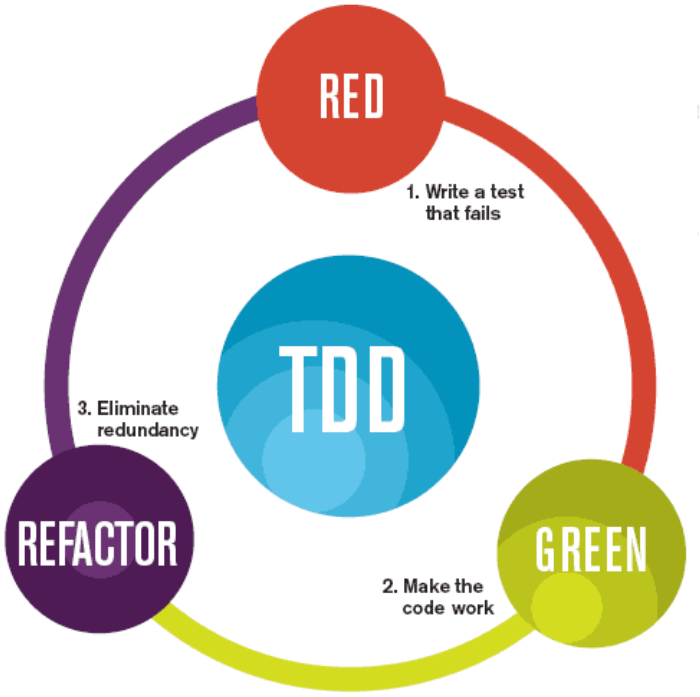
511790 – Matheus Leandro de Melo Silva

473143 - João Lucas de Oliveira Lima

537777 - Eduardo Henrique Brito da Silva

Test Driven Development

Test Driven Development(TDD) tem como objetivo o teste da aplicação durante seu desenvolvimento. Ou seja, no TDD é enfatizado a criação de testes automatizados antes de escrever o código de produção. Vale ressaltar que, no TDD os testes unitários são feitos sobre métodos e/ou classes individuais. Por fim, o processo TDD segue um ciclo iterativo curto e consiste em três etapas principais.



The mantra of Test-Driven Development (TDD) is "red, green, refactor."

Fases:

1. Red (Vermelho):

Na fase inicial, será escrito um teste automatizado que descreve uma funcionalidade específica que desejamos implementar no nosso software. Porém, o teste é projetado para falhar inicialmente, porque o código de produção correspondente ainda não existe.

2. Green (Verde):

Agora, escreveremos o código de produção necessário para fazer o teste passar. Pois, o objetivo é que o teste automatizado mude de "falha" (vermelho) para "sucesso" (verde), indicando que o código atende aos critérios definidos no teste.

3. Refactor (Refatoração):

Por fim, podemos refatorar o código, se necessário. Porque, por meio da refatoração, melhoramos a estrutura da qualidade do código, tornando-o mais legível, eficiente e sustentável.

Benefícios Motivadores:

1. Maior Confiabilidade:

TDD ajuda a garantir que o código seja testado automaticamente e frequentemente, o que reduz significativamente a probabilidade de introduzir erros.

2. Documentação Automática:

Os testes servem como documentação atualizada do comportamento do software, tornando mais fácil para os desenvolvedores entenderem como as diferentes partes do sistema devem funcionar.

3. Design Melhorado:

O TDD incentiva o desenvolvimento de código modular e coeso, pois você deve pensar em como testar cada parte do seu código antes de escrevê-lo. Isso geralmente leva a um design de software mais limpo.

4. Refatoração Segura:

Com testes automatizados em vigor, você pode fazer refatorações com confiança, sabendo que, se algo der errado, os testes identificaram os problemas.

5. Detectar Problemas:

O TDD ajuda a identificar problemas de lógica e integração no início do ciclo de desenvolvimento, quando são mais fáceis e menos dispendiosos de corrigir.

Conclusão:

Portanto, escolhemos o TDD por ser uma prática de desenvolvimento que promove a qualidade do código, a confiabilidade do software e a documentação eficaz, ao mesmo tempo em que melhora o processo de desenvolvimento ao permitir uma detecção precoce e correção de problemas. Ele é amplamente adotado em equipes de desenvolvimento de software que buscam criar software de alta qualidade de forma consistente.

Teste de Integração

Diferente do TDD, o teste de integração é a fase do teste de software em que módulos são integrados e testados em grupo. Como por exemplo, o software acessando um banco de dados ou fazendo uma chamada externa a outros sistemas. O objetivo principal do teste de integração é garantir que as diversas partes do sistema funcionem juntas de forma eficaz e que os dados sejam transferidos e processados corretamente entre essas partes.

Benefícios Motivadores:

1. Integração de Componentes:

Em um sistema de software complexo, o código é frequentemente dividido em componentes, módulos ou serviços menores para facilitar o desenvolvimento e a manutenção. O teste de integração concentra-se em verificar como esses componentes interagem e cooperam entre si.

2. Verificação de Fluxo de Dados:

Garantir que os dados sejam transmitidos de um componente para outro sem perda ou corrupção e que sejam transformados e processados de acordo com as especificações.

3. Identificação de Problemas de Interface:

Durante o teste de integração, os problemas relacionados às interfaces entre os componentes são identificados e resolvidos. Podemos incluir problemas de compatibilidade de tipos de dados, inconsistências na comunicação ou na ordem de chamadas de funções, entre outros.

4. Teste de Fluxo de Controle:

Além do fluxo de dados, o teste de integração verifica o fluxo de controle do programa, ou seja, a sequência em que as funções ou componentes são chamados e como o controle do programa é transferido entre eles.

5. Detecção de Problemas de Interoperabilidade:

Em sistemas que dependem de componentes de terceiros ou de sistemas externos, o teste de integração também pode detectar problemas de interoperabilidade, como incompatibilidades com APIs (interfaces de programação de aplicativos) ou protocolos de comunicação.

8. Integração Contínua:

Muitas equipes incorporam testes de integração em seus processos de integração contínua, o que significa que os testes são executados automaticamente sempre que uma alteração no código é integrada ao sistema, garantindo que a integração entre componentes seja verificada constantemente.

9. Testes de Ponta a Ponta:

Os testes de integração podem ser estendidos para abranger toda a aplicação, verificando a integração entre todas as partes do sistema, incluindo a interface do usuário.

Conclusão:

Assim, o teste de integração é essencial para garantir que todas as partes de um sistema trabalhem em harmonia e que o software funcione como um todo coeso. É particularmente importante em sistemas complexos, onde a interação entre os componentes desempenha um papel crítico no funcionamento geral do software. Estes testes ajudam a identificar problemas de integração, interoperabilidade e fluxo de dados antes que o software seja implantado em um ambiente de produção.

Teste de sistema

O teste de sistema é uma etapa fundamental no ciclo de teste de software e se concentra em verificar se o sistema como um todo atende aos requisitos e especificações definidos para ele.

Benefícios Motivadores:

1. Verificação de Requisitos:

O objetivo principal dos testes de sistema é garantir que o software atenda aos requisitos funcionais e não funcionais definidos no início do projeto. Isso envolve a verificação de todas as funcionalidades, recursos e comportamentos esperados do sistema.

3. Ambiente Realista:

Os testes de sistema são normalmente conduzidos em um ambiente que se assemelha ao ambiente de produção o mais próximo possível. Isso ajuda a simular as condições reais em que o software será usado.

4. Teste de Casos de Uso:

Os casos de teste de sistema são frequentemente baseados em cenários de uso realistas, que representam como os usuários finais interagem com o sistema. Isso inclui fluxos de trabalho típicos, cenários de erro e casos excepcionais.

5. Teste de Desempenho e Escalabilidade:

Os testes de sistema podem incluir testes de desempenho para verificar como o sistema se comporta sob carga e se atende às métricas de desempenho definidas. A escalabilidade também pode ser testada para verificar se o sistema pode lidar com um aumento significativo na carga.

6. Teste de Segurança:

A segurança do sistema é uma preocupação crítica, e os testes de sistema frequentemente incluem verificações de segurança para identificar vulnerabilidades e ameaças potenciais.

7. Teste de Usabilidade:

Os aspectos de usabilidade do sistema são testados para garantir que a interface do usuário seja intuitiva e eficiente.

Conclusão:

Logo, os testes de sistema são uma etapa crucial no processo de garantia de qualidade de software. Eles verificam se o sistema atende aos requisitos, funciona como esperado em condições reais e é adequado para implantação. É a última etapa antes da entrega do software aos usuários finais ou clientes.

Teste do layout do compose

A avaliação das Interfaces de Usuário (UIs), das telas ou do layout desempenha um papel crucial na verificação do comportamento adequado do código no Compose, contribuindo para aprimorar a qualidade do aplicativo ao detectar erros no estágio inicial do processo de desenvolvimento. O Compose disponibiliza um conjunto de APIs de teste que facilitam a localização de elementos, a verificação de atributos e a execução de ações do usuário. Essas APIs também incluem funcionalidades avançadas, como a manipulação do tempo.

Os testes de UI no Compose fazem uso da semântica para interagir com a hierarquia da interface. A semântica, como o próprio nome sugere, confere significado a partes da UI. Nesse contexto, uma "parte da UI" (ou elemento) pode representar qualquer coisa, desde um único elemento combinável até uma tela completa. A árvore semântica é automaticamente gerada juntamente com a hierarquia da UI e a descreve em detalhes.

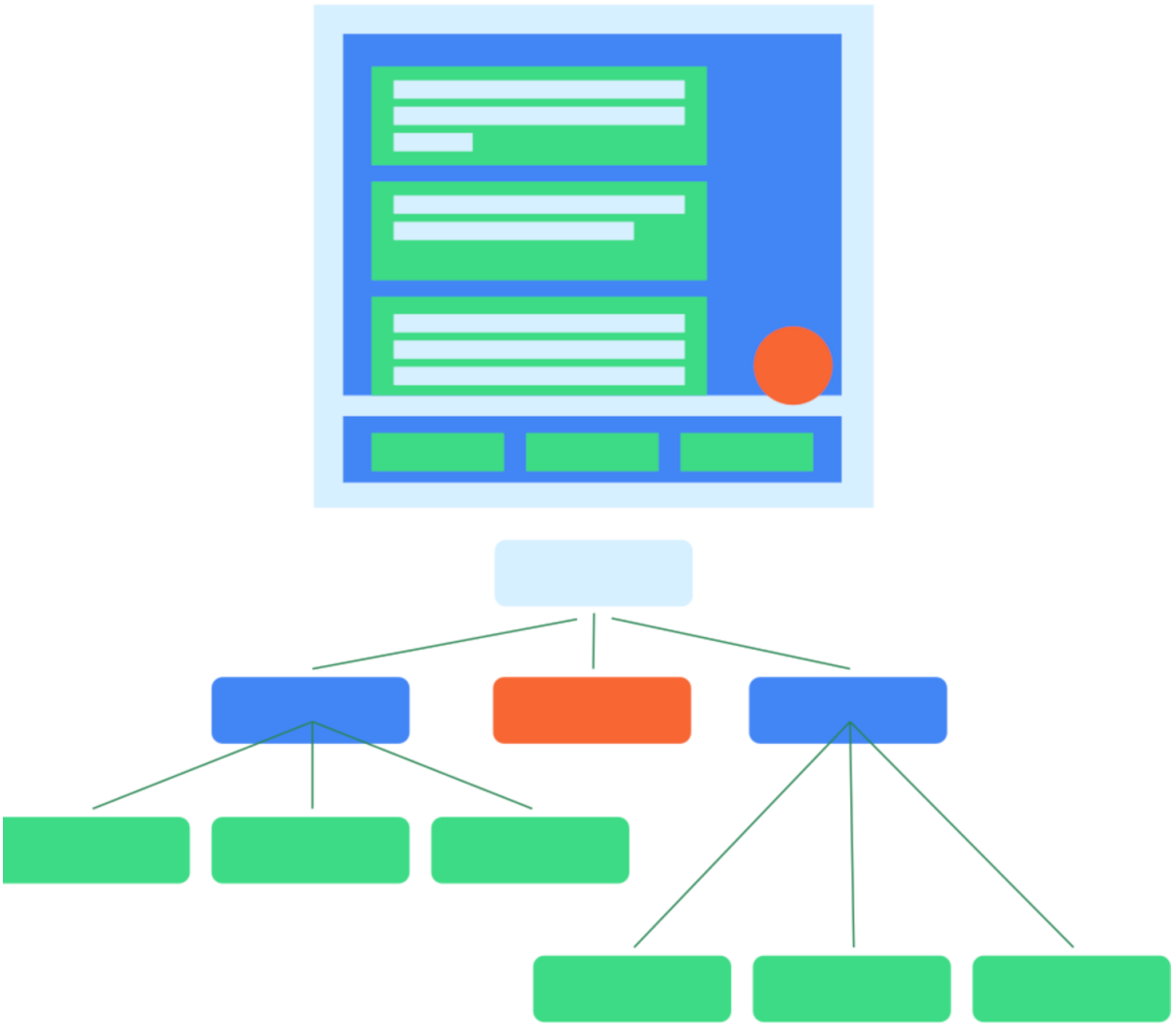


Diagrama mostrando um layout típico de IU e a maneira como esse layout seria mapeado para uma árvore semântica correspondente.

O framework semântico é principalmente utilizado para fins de acessibilidade, e, portanto, os testes se baseiam nas informações fornecidas pela semântica em relação à estrutura da interface de usuário. A decisão sobre o que e em que extensão deve ser exposto fica a critério dos desenvolvedores. Por exemplo, ao considerar um botão, composto por um ícone e um elemento de texto, a árvore semântica padrão contém apenas o rótulo de texto. Isso ocorre porque alguns elementos que podem ser compostos, como o componente "Text", já incluem automaticamente algumas propriedades na árvore semântica, sendo possível adicionar mais propriedades à árvore semântica utilizando um "Modifier".

Por fim, ao integrar eficazmente os testes de UI e a semântica em projetos Compose, nós poderemos aprimorar a qualidade da aplicação, detectando e corrigindo erros desde o início do processo de desenvolvimento, ao mesmo tempo em que garantimos uma experiência acessível e de alta qualidade para todos os usuários.

Acompanhamento dos testes

- TDD
    - Foram iniciados
    - Atualmente todos estão com falhas, pois focamos primeiro na interface
- Testes de layout
    - Foram testados os composables de títulos, imagens e botões usados atualmente na aplicação.
- Testes de Interface
    - Ainda não foram iniciados
- Testes Integrados
    - Ainda não foram iniciados