

Sistemas Embarcados

Aula 4 - Testes de software



O que é teste de software e por que testar?

O teste nada mais é que executar trechos do código desenvolvido para verificar se ele se comporta como esperado. Criar rotinas de teste é super importante para garantir a qualidade do software e verificar se não existem erros durante a execução. Existem diferentes tipos de teste e alguns deles estão listados abaixo:

- Teste de unidade
- Teste de componentes (no caso de frontends)
- Teste de Integração
- Teste de end-to-end (e2e)
- Teste de fumaça
- Teste de caixa preta e caixa branca

Nesta disciplina vamos focar nos testes de unidade e de integração

Na documentação do Dart é possível encontrar mais informações sobre como executar os testes nesta linguagem.

Teste de Unidade

O teste de unidade serve para verificar o comportamento de pequenas partes de código dentro de um sistema maior. Nesse caso, as pequenas partes que podemos testar são funções, classes e bibliotecas. Vamos voltar ao exemplo das figuras geométricas:

```
class Quadrado extends FigurasGeometricas{
  int _lado;

  Quadrado(this._lado);

  @override
  int calculaArea(){
    return _lado * _lado;
  }

  @override
  int calculaComprimento(){
    return 4 * _lado;
  }
}
```

```
import 'package:test/test.dart';

void main() {
  test('calcula a area', () {
    var q = Quadrado(5);
    expect(q.calculaArea(), 25);
  });

  test('calcula comprimento', () {
    var q = Quadrado(5);
    expect(q.calculaComprimento(), 20);
  });
}
```

Importante: para executar os testes no Android Studio é importante adicionar o pacote de testes nas dependências do projeto, dentro do arquivo `pubspec.yaml`.

```
dev_dependencies:
  test: ^1.5.1
```

Cobertura de código

A cobertura de código é uma maneira de identificar o quanto seus testes estão efetivamente executando trechos do código fonte da sua aplicação. Quanto maior a cobertura, mais o software está sendo testado. Porém, alcançar 100% de cobertura de código pode não ser viável.

A seguir estão alguns comandos para verificar a cobertura de código em um projeto com Dart/Flutter

```
## Executa os testes Dart e coloca os resultados no diretório `./coverage`:  
dart run test --coverage=./coverage  
  
## Ativa o pacote `coverage` (se necessário):  
dart pub global activate coverage  
  
## Formata os resultados de cobertura para LCOV (apenas arquivos no diretório "lib")  
dart pub global run coverage:format_coverage --packages=.dart_tool/package_config.json --report-on=lib --lcov -o ./coverage/lcov.info  
  
## Gera o relatório LCOV:  
genhtml -o ./coverage/report ./coverage/lcov.info  
  
## Abre o HTML com o relatório de cobertura:  
open ./coverage/report/index.html
```

Mock

O mock é uma técnica usada para simular a chamada de uma função ou método de objeto sem de fato executar seu respectivo código. Com isso podemos simular a chamada de bancos de dados e APIs sem de fato consumir o serviço, tornando os testes mais ágeis. Para criar mocks nos testes em Dart devemos usar o pacote `mockito` ou usar as classes de teste disponíveis no pacotes adicionados ao projeto, como no caso do `http/testing`.

```
Future<bool> getPokemonListFromAPI() async {
  bool apiCallStatus = false;
  try {
    var url = Uri.parse(
      "https://pokeapi.co/api/v2/pokemon/?offset=0&limit=10"
    );
    var response = await _client.get(url);
    _listaDePokemons = json.decode(response.body) as List;
    apiCallStatus = response.statusCode == 200 ? true : false;
  } catch (e) {
    // Handle error
  }
}
```

Aqui temos um teste que realiza o mock do `Client` da biblioteca `http`. [Link pro Replit](#)

```
test("busca de catalogo de pokemons com mock", () async {
  final mockClient = MockClient(
    (request) async {
      return Response(jsonEncode(responseMap), 200);
    }
  );

  Catalog pokemonCatalog = Catalog(mockClient);
  await pokemonCatalog.getPokemonListFromAPI();
  const expectedPokeNames = [
    "bulbasaur",
    "ivysaur",
    "venusaur",
    "charmander",
    "charmeleon"
  ];

  expect(pokemonCatalog.listaDeNomes, expectedPokeNames);
});
```

Teste de Integração

Neste teste nós verificamos o comportamento do software quando conectado com o outros serviços, como WEB APIs, bancos de dados e outros dispositivos (no caso de software que envolvem conexões Bluetooth, NFC e etc).

Aqui é importante definirmos Ambientes diferentes para rodar os testes:

- **Dev:** Ambiente local de desenvolvimento do software (desktop, notebook, smartphone, simuladores e todos os dispositivos necessários para o desenvolvimento do software). Aqui devemos rodar os testes de unidade e testes e2e que não necessitam de interação pelo usuário, ou que a interação possa ser simulada;
- **Staging:** Ambiente para testes que funciona de maneira muito semelhante ao ambiente de produção. Aqui serão realizados testes de integração, porém o banco de dados e os endereços de APIs deverão ser bancos e endereços de teste.
- **Prod:** Ambiente de produção é aquele em que o usuário vai atuar e inserir dados no sistemas. Nesse ambiente não devemos realizar testes e devemos usar os bancos de dados e endereços de APIs definitivos.

Do and Don't do

Don't do

- Utilizar o ambiente de produção para testes;
- Usar banco de dados e APIs de produção para teste;
- Desenvolver classes com métodos, construtores e propriedades privadas sem testar. (Problema com encapsulamento).

Do

- Manter coesão e encapsulamento das classes (manter responsabilidade única para cada classe);
- Utilizar técnicas como o TDD (Test Driven Development). (#TODO: compartilhar livro sobre TDD)