Testes em aplicativos móveis

Apresentação

A dependência das pessoas em aplicações computacionais, sobretudo as voltadas para computação móvel, vem aumentando também a exigência por qualidade dessas aplicações. Para garantir um nível mínimo de qualidade, diversos padrões de testes de *software* devem ser aplicados. Os usuários estão mais exigentes e a concorrência está maior. Além do fato de algumas aplicações lidarem com dados críticos, nos quais os erros podem ter consequências graves.

Nesta Unidade de Aprendizagem você vai estudar os princípios de testes de *software* e as principais ferramentas utilizadas para testar as aplicações, além disso, vai entender como elas são aplicadas.

Bons estudos.

Ao final desta Unidade de Aprendizagem, você deve apresentar os seguintes aprendizados:

- Definir os princípios básicos de testes em aplicativos móveis.
- Identificar as principais ferramentas utilizadas para o teste em aplicativos móveis.
- Aplicar teste em aplicativos móveis.

Desafio

Na medida em que um *software* se desenvolve, ele cresce, sofre alterações, durante o ciclo de desenvolvimento e entrega contínua, e aumenta também o risco de problemas, juntamente com a complexidade do sistema em si. Portanto, iniciar um desenvolvimento, já construindo os respectivos testes em paralelo, pode representar um maior investimento de esforço inicial, mas que poupará esforço no futuro. Contudo, nem sempre esse é o caso.

Você é responsável por uma aplicação voltada a engenheiros que realizam cálculos de sistemas elétricos em campo.

Como o projeto surgiu de maneira muito despretensiosa e com orçamento limitado, você optou por não realizar testes automatizados no início.

A aplicação foi crescendo com o tempo e adquiriu grande popularidade. Dentre as funcionalidades, algumas são pouco utilizadas, mas é imprescindível que, quando utilizadas, forneçam resultados precisos.

O departamento de suporte registrou algumas chamadas de engenheiros, relatando que uma das funções estava retornando resultados incoerentes com os dados entrados.

Como responsável pelo desenvolvimento desta aplicação, de que maneira você solucionaria o problema, garantindo que todos os cálculos estejam com as fórmulas corretas? Crie uma solução e explique o porque de cada ação tomada por você e sua equipe.

Infográfico

Os testes automatizados trouxeram mais segurança para as equipes de desenvolvimento, seja na construção inicial ou na adição de funcionalidades - no desenvolvimento contínuo. Quanto mais complexo um sistema é, mais benefícios os testes automatizados trazem.

Contudo, existe um ciclo de vida dos casos de testes de uma aplicação. Acompanhe no infográfico um exemplo deste ciclo.

CICLO DE VIDA DOS TESTES AUTOMATIZADOS

Testes unitários (ou de unidade)

Cada unidade do *software*, seja uma classe, método ou função, serviço etc., pode ser testado por meio dos testes de unidade – que focam em testar a lógica do respectivo bloco de código.

Testes de integração

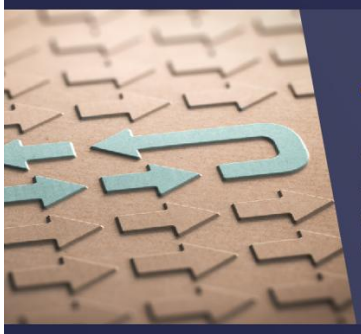
Após juntar algumas unidades em um conjunto maior, é necessário testar a integração entre as unidades. Essa etapa garante que tais partes trabalharão corretamente juntas.

Testes end-to-end ou ponta-a-ponta

Nessa parte, testa-se o sistema como um todo, inclusive a interface de usuários, por meio de ferramentas que simulam a interação do usuário com o sistema. É um tipo de teste que envolve investimento alto de recursos.

Testes de aceitação

Nessa etapa, o foco são as regras de negócio e a usabilidade do sistema. É quando o cliente testa o *software* para garantir que está de acordo com os requisitos funcionais solicitados.



Testes de regressão

Os testes desenvolvidos nas etapas anteriores são aplicados novamente na adição de funcionalidades, garantindo que o que estava funcionando antes, continue funcionando após as alterações.

Aponte a câmera para o código e acesse o link do conteúdo ou clique no código para acessar.

Conteúdo do Livro

Os padrões de qualidade, no desenvolvimento de *software*, são uma parte cada vez mais importante no processo. Da mesma forma, surgem ferramentas que permitem aos desenvolvedores se ater mais ao negócio e menos a atividades repetitivas, como os testes manuais. São os chamados *frameworks* de testes automatizados.

Esses *frameworks* também estão disponíveis para o desenvolvimento de aplicações *mobile* e o próprio Ionic traz consigo algumas ferramentas de testes como parte de seu conjunto.

No capítulo Testes em aplicativos móveis, da obra *Desenvolvimento para Dispositivos Móveis*, base teórica desta Unidade de Aprendizagem, você vai estudar um pouco mais sobre os testes e os princípios de qualidade, as ferramentas que o Ionic fornece para automatizar testes unitários e de interface e também vai estudar um caso de testes, como exemplo, para implementar essa prática nos seus desenvolvimentos.

Boa leitura!

DESENVOLVIMENTO PARA DISPOSITIVOS MÓVEIS

Victor Luiz Simas

SAGAH
SOLUÇÕES EDUCACIONAIS INTEGRADAS

Testes em aplicativos móveis

Objetivos de aprendizagem

Ao final deste texto, você deve apresentar os seguintes aprendizados:

- Definir os princípios básicos de testes em aplicativos móveis.
- Identificar as principais ferramentas utilizadas para o teste em aplicativos móveis.
- Aplicar testes em dispositivos móveis.

Introdução

Neste capítulo, você estudará os fundamentos dos testes de *software* e os princípios da qualidade no desenvolvimento de aplicação, as ferramentas utilizadas para testes unitários e ponta a ponta (e2e, em inglês *end-to-end*) com a plataforma Ionic, bem como exemplos sobre sua implantação.

Testes de *software* e princípios de qualidade

Uma aplicação é um *software* e, como tal, está sujeita a conter falhas ocorridas durante a codificação (*bugs*), como qualquer outro sistema. Assim, é fundamental manter a qualidade e assegurar que o comportamento da aplicação está de acordo com as regras de negócio definidas, bem como verificar se o resultado esperado da camada de interação do usuário realmente responde de forma adequada.

Segundo Pressman e Maxim (2016), muitas vezes, uma rotina de testes exige mais trabalho do que outras etapas da engenharia de *software*. Geralmente, planejar, preparar os ambientes e *frameworks*, codificar, executar e analisar os resultados dessas rotinas é mais dispendioso em termos de mão de obra e gasto de tempo, mas as vantagens obtidas com o nível de qualidade mais elevado compensam esse investimento, por exemplo, evitar o retrabalho tardio (débitos técnicos são problemas que causam impactos no futuro, quando o usuário já estiver utilizando a plataforma) e a má impressão que uma aplicação com muitas falhas possa ter, bem como tentar eliminar brechas de segurança.

Entretanto, de acordo com Pressman e Maxim (2016), não adianta testar apenas uma parte do sistema. O cenário ideal é ter uma aplicação totalmente realizada por testes ou chegar o mais perto possível desse objetivo. Na engenharia de testes de *software*, existe uma ampla gama de tipos diferentes, com foco em áreas distintas e que se complementam, mas do ponto de vista procedimental, eles são divididos em quatro etapas principais implementadas em sequência. Inicia-se com o teste de componentes de forma individual, como funções ou métodos, classes e objetos, garantindo que estejam retornando o resultado esperado. Essa etapa é chamada de teste de unidade ou unitário (PRESSMAN; MAXIM, 2016).

Na sequência, testa-se o sistema em um aspecto mais amplo, com suas funcionalidades já postas de forma integrada, e não mais em unidades individuais (PRESSMAN; MAXIM, 2016). Essa etapa é chamada de testes de integração, que podem verificar áreas do sistema ou o sistema como um todo, dependendo da sua complexidade. No caso de um sistema inteiro, pode-se chamar de testes e2e.

Depois, precisa-se validar se os requisitos funcionais, comportamentais e de desempenho foram atingidos na etapa de desenvolvimento. Assim, os testes são realizados mais a nível de negócios do que de código e se chamam testes de validação ou aceitação. Já na última etapa, o teste de sistema engloba todas as variáveis que compõem o sistema em si, como a aplicação, o banco de dados, o *hardware* e, dependendo da fase, os usuários (PRESSMAN; MAXIM, 2016).

Dentro das práticas de *Quality Assurance* ou Garantia de Qualidade (QA), os testes podem ser executados manualmente ou automatizados. Hoje, o ideal é que se foque mais nos testes automatizados por diversos motivos, como:

- repetição, pois os seres humanos se entediam facilmente com tarefas repetitivas e são limitados quanto a isso, assim, podem delegar uma tarefa a ser repetida milhares de vezes a uma máquina;
- entrega contínua, um princípio que utiliza mecanismos que compilam, empacotam e testam automaticamente os *softwares*, conforme são liberados pelos desenvolvedores (somente dependem da aprovação para entrar em produção).

Já em termos de técnicas, pode-se mencionar também os testes de caixa branca e caixa preta. O primeiro exige que o testador tenha acesso ao código fonte da aplicação e possa gerar roteiros de testes por meio de códigos, sendo recomendável para testes de unidade e integração. O segundo avalia os componentes mais funcionais e externos do *software*, bem como a entrada e

saída de dados, mas sem o testador precisar acessar o código fonte. Os dados entram e são visualizados por interfaces de usuário ou de *frameworks* que simulam a interação. Esse tipo de teste pode ser usado nas etapas de testes de integração, validação e sistema.

Quanto às técnicas, existem também os testes de regressão, que reaplicam a mesma rotina já existente de testes em novas versões, para verificar se os novos elementos não causaram nenhum problema. Você ainda pode aplicar outro ramo de testes que fogem de requisitos funcionais. Assim, a usabilidade, a escalabilidade, capacidade de carga e desempenho são efetivamente testados e inseridos no controle de qualidade.

Em termos de computação móvel, destaca-se a usabilidade, cujo principal fator analisado é a experiência do usuário com a aplicação. Um bom *software* deve ser claro, simples e intuitivo para que a pessoa precise de pouco ou nenhum treinamento para operar essa aplicação. Isso é particularmente percebido no universo *mobile*, que tem metodologia de interação diferenciada da computação tradicional por se basear sobretudo em uma tela de toque e pequenas proporções.

Testes em aplicações Ionic

Devido à divisão do mercado de dispositivos móveis em duas plataformas independentes e incompatíveis, opta-se por usar uma ferramenta de desenvolvimento multiplataforma capaz de gerar aplicações com código nativo, sem, contudo, utilizar o *software development kit* (SDK) e linguagens particulares de cada sistema, eliminando a necessidade de manter duas bases de código. Por isso, emprega-se o *framework* Ionic, baseado na linguagem TypeScript, um superconjunto do JavaScript no desenvolvimento, e que traz um conjunto de bibliotecas e *frameworks* de testes para garantir a qualidade da aplicação.

Testes unitários

Os testes unitários são a unidade elementar dos testes, que, no Ionic, pode ser uma função, um componente, uma página, um *service*, um *pipe,* entre outros. Neste caso, será testada uma unidade de código isolada do resto do sistema, impedindo que falhas em outras unidades possam interferir.

Para executar os testes unitários em aplicações Ionic, usa-se uma plataforma denominada Jasmine, a qual se baseia no conceito de *behaviour driven development* ou desenvolvimento orientado a comportamento. Devido a esse

conceito, os testes são baseados nos comportamentos que as unidades devem executar e no resultado esperado.

Saiba mais

O *behaviour driven development* (ou *behavior driven development*, no inglês norte-americano), surgiu como uma abordagem de desenvolvimento *outside in,* de fora para dentro. Como o nome sugere, seu foco é trabalhar o desenvolvimento do código sobre as necessidades de negócios, baseadas em como o sistema se comportará. Ele é derivado diretamente do *Domain Driven Development* (DDD), sendo uma alternativa ao *Test Driven Development* (TDD), o qual tem um apelo mais técnico (NORTH, 2006, documento *on-line*).

Jasmine se baseia em três funções principais: a `describe`, que agrupa especificações relacionadas e, em geral, faz parte de cada arquivo englobando os testes em si e recebendo uma *string* com a descrição do teste e uma função; a `it`, a qual contém uma descrição do caso de teste específico e uma função que será realizada; e a `expect`, que explicita o resultado esperado no teste.

Exemplo

```
describe("Conjunto maior de testes", () => {
    var exemplo;
    it("testa se a variável exemplo é verdadeira", () => {
        exemplo = true;
        expect(exemplo).toBe(true);
    });
});
```

Fonte: Adaptado de Your first... (2019, documento *on-line*).

Como estas funções são JavaScript ou TypeScript, o padrão de escopo de variáveis é aplicado, sendo que a variável exemplo estaria acessível a todos os

casos `it` dentro da `describe`. Todavia, existem momentos em que se deve testar funções e elementos do código que dependem dos dados provindos de outra parte ou função, ou que sejam externos à aplicação.

Neste caso, como o objetivo é isolar efetivamente uma unidade de código do restante do sistema, não se pode contar com as dependências externas que, em tese, poderiam influenciar no resultado dos testes. Por exemplo, ao testar um componente que efetua a chamada a uma *Application Programming Interface* (API) externa por meio de um serviço. Se, na prática, você fizer a chamada, corre o risco de suas falhas anularem o teste. Outro exemplo é uma chamada a um banco de dados que pode existir, ou não estar com sua estrutura totalmente definida.

Para lidar com esse tipo de situação, existe um objeto chamado *mock*, que é um tipo de objeto com dados fictícios, previamente estabelecidos, contendo o mínimo necessário para o teste da funcionalidade e permite saber se foi ou não acessado corretamente (se houve o acesso, quantas vezes, etc.).

No uso de Jasmine, os *mocks* são chamados de *spies* (*spy*, no singular), os quais existem no contexto das funções `describe` e `it`, podendo simular dados, elementos ou funções (característica inerente do JavaScript). Por exemplo, é possível simular uma função que chama um conjunto de dados de uma API do *back-end* e retorna dados predeterminados no seu código. Jasmine também possui métodos como `toHaveBeenCalled` e `toHaveBeenCalledTimes`, que permitem saber se o objeto ou função foi acessado (primeiro) e quantas vezes foi requisitado (segundo). Tanto essas como outras funções são bastante úteis ao executar os testes unitários com o *framework* Jasmine nas aplicações Ionic.

Testes *end-to-end*

Os testes unitários visam desvendar os problemas encontrados na lógica do código, considerando que uma unidade de código pode funcionar perfeitamente bem isolada, mas haver um problema de integração entre as unidades. Por isso, existem os testes de integração.

Quanto ao Ionic, esses testes são geralmente realizados como testes e2e, cobrindo desde a integração das unidades até a interface de usuário. Na plataforma, eles são tratados como um projeto separado da aplicação principal. Ao gerar um projeto Ionic usando a ferramenta de linha de comando `ionic--cli`, cria-se também a aplicação de testes e2e, em uma pasta homônima.

No Ionic, utiliza-se duas ferramentas nesses testes: o Jasmine, para estruturação e execução de testes, por exemplo, escrever os casos de teste com `describe, it` e `expect`; e o Protractor, que controla o uso do *browser*.

Para o Protractor, deve-se informar à ferramenta qual interação que ele deve simular, como determinar que clicará em um botão da interface ou digitar um texto no campo de entrada. A identificação dos elementos ocorre por classes *Cascading Style Sheets* (CSS), identificadores de modelo, etc. (TUTORIAL, 2018, documento *on-line*).

Teste de aplicações na prática

Agora, você analisará alguns casos de testes práticos com as ferramentas do Ionic, o qual é estruturado já pensando na capacidade de teste das aplicações. Portanto, determinados elementos são gerados durante a criação do projeto; e outros, na criação dos elementos, quando se utiliza a ferramenta de linha de comando do Ionic (`ionic-cli`).

Uma estrutura básica de testes do Ionic consiste nos elementos apresentados no exemplo a seguir.

Exemplo

```
describe('Descrição do Teste', () => {

    let varTeste;

    beforeEach(async () => {
        // Operações a serem executadas antes
        // de cada etapa dos testes, como reset
        // de variáveis, por exemplo

        varTeste = true; // Define como True
    });

    it('Testando se VarTeste é true', () => {
        expect(varTeste).toBe(true);
    });
});
```

Fonte: Adaptado de Testing... (2019, documento *on-line*).

Testes unitários

Veja, agora, os testes de alguns elementos do Ionic. Em relação aos componentes `page` e `componente`, o padrão de testes segue exatamente a metodologia do Angular, pois seus elementos são semelhantes.

Exemplo

```
import { CUSTOM _ ELEMENTS _ SCHEMA } from '@angular/core';
import { async, ComponentFixture, TestBed } from '@angular/
core/testing';
import { InicioPage } from './inicioPage';

describe('Casos de testes da tela (page) inicial', () => {
    let componente: InicioPage;
    let fixture: ComponentFixture<InicioPage>;

    beforeEach(async () => {
        TestBed.configureTestingModule({
            declarations: [InicioPage],
            schemas: [CUSTOM _ ELEMENTS _ SCHEMA]
        }).compileComponents();
    });

    // Podemos ter mais de uma instrução beforeEach
    beforeEach(() => {
        fixture = TestBed.createComponent(InicioPage);
        componente = fixture.componentInstance;
        fixture.detectChanges();
    });

   it('Testando se a página é criada corretamente', () => {
        expect(componente).toBeTruthy();
    });
});
```

Fonte: Adaptado de Testing... (2019, documento *on-line*).

Agora, você pode ver como funciona o teste de um *service*, que geralmente faz parte dos casos que usam os *mocks* e já foram mencionados. Nesse caso, os *mocks* permitem que o serviço possa ser testado de forma isolada, sem depender de fontes de dados externas.

Exemplo

```
import { LocalidadeService } from './LocalidadeService';

describe('Testes para a classe de serviços que fornece as
condições de localidades', () => {
    let service: LocalidadeService;
    let locSpy; // Variável para armazenar o mock

    beforeEach(async () => {
         locSpy = jasmine.createSpyObj('MeteorologiaServ
ice', {
            vento: 10,
            pressao: 1013,
            temperatura: 18
        });
        service = new LocalidadeService(locSpy);
    });

    it('Buscando a pressão do ar de uma determinada lo-
calidade', () => {
        locSpy.pressao.and.returnValue(1013);
    });
});
```

Fonte: Adaptado de Testing... (2019, documento *on-line*).

Do mesmo modo, outros elementos podem ser testados usando a metodologia e as ferramentas descritas.

Testes *end-to-end*

Para o Ionic, assim como para o Angular, os elementos *hypertext markup language* (HTML) da página são encapsulados em uma classe TypeScript, permitindo testá-la de forma similar a uma API. Nos exemplos deste capítulo, usa-se os seletores CSS para manipulação do *document object model* (DOM).

Voltando ao projeto gerado pelo Ionic para os testes de integração, sua pasta contém inicialmente quatro arquivos:

- `protractor.conf.ts` — arquivo de configuração do Protractor, em que se pode configurar a *uniform resource locator* (URL) e o arquivo de especificações (*specs*), que contém os casos de teste.
- `tsconfig.e2e.json` — configurações específicas do TypeScript para a aplicação de testes, as quais também estão na aplicação principal.
- `src/app.po.ts` — um objeto do tipo *page*, que contém métodos para a navegação nas interfaces e a manipulação de elementos.
- `src/app.e2e-spec.ts` — *script* de testes.

Exemplo

A seguir, você verá a configuração de um arquivo base com os métodos que serão usados para testar as *pages*.

```
import { browser, by, element, ExpectedConditions } from
'protractor';

export class PageObjBase {
    private caminho: string;
    protected tag: string;

    constructor(tag: string, caminho: string) {
        this.caminho = caminho;
        this.tag = tag;
    }
```

```
    load() {
        return browser.get(this.caminho);
    }

    elementoRaiz() {
        return element(by.css(this.tag));
    }

   aguardarOcultacaoElemento() {
        browser.wait(
               ExpectedConditions.invisibilityOf(this.
elementoRaiz()
        );
    }

    aguardarCriacaoElemento() {
        browser.wait(
         ExpectedConditions.presenceOf(this.elementoRaiz()),
            5000);
    }

    aguardarEnquantoNaoCriado() {
        browser.wait(
            ExpectedConditions.not(
                    ExpectedConditions.presenceOf(this.
elementoRaiz())
            ), 5000);
    }

    aguardarAteSerVisivel() {
        browser.wait(
                  ExpectedConditions.VisibilityOf(this.
elementoRaiz()),
        ), 5000);
    }
```

```
    getTituloDaPagina() {
         return element(by.css(`${this.tag} ion-title`)).
getText();
    }

   protected digitarTexto(seletor: string, texto: string) {
     const elemento = element(by.css(`${this.tag}${seletor}`));
        const entrada = elemento.element(by.css('input'));
        entrada.sendKeys(texto);
    }

    protected clicarBotao(seletor: string) {
     const elemento = element(by.css(`${this.tag}${seletor}`));
        browser.wait(
         ExpectedConditions.elementToBeClickable(elemento)
        );
        elemento.click();
    }
}
```

Fonte: Adaptado de Testing... (2019, documento *on-line*).

Ao criar uma classe padrão para testar as *pages*, há muitas partes reutilizáveis, evitando que se escreva um código redundante. Assim, apesar de precisar criar uma classe de teste para cada *page* que quiser testar, pode-se estender essa classe padrão e utilizar os métodos genéricos.

Exemplo

```
import { browser, by, element, ExpectedConditions } from
'protractor';
import { PageObjBase } from './pageTestBase.po';
```

```
export class LoginPage extends PageObjBase {
    constructor() {
        super('app-login', '/login');
    }

    aguardarErro() {
        browser.wait(
           ExpectedConditions.presenceOf(element(by.css('.
error'))),
            5000
        );
    }

    getMensagemErro() {
        return element(by.css('.error')).getText();
    }

    digitarUsername(username: string) {
       this.digitarTexto('#username-input', username); //
reaproveitando método herdado
    }

    digitarSenha(senha: string) {
         this.digitarTexto('#senha-input', senha); // rea-
proveitando método herdado
    }

    efetuarLogin() {
        this.clicarBotao('#login-button');
    }
}
```

Fonte: Adaptado de Testing... (2019, documento *on-line*).

Exemplo

Neste exemplo, cria-se o *script* que executará os testes na página de *login* da aplicação.

```
import { AppPage } from '../page-objects/pages/app.po';
import { LoginPage } from '../page-objects/pages/login.po';
import { DadosPage } from '../page-objects/pages/dados.po';

describe('Efetuando login na aplicação' () => {
    const app = new AppPage();
    const login = new LoginPage();
    const dados = new DadosPage();
});

beforeEach(() => {
    app.load();
});

describe('Antes de logar', () => {
    it('mostrar tela de login', () => {
      expect(login.rootElement().isDisplayed()).toEqual(true);
    });

   it('Mostrar mensagem de erro de o login falhar', () => {
        login.digitarUsername('meuUsuario');
        login.digitarSenha('senhaErrada');
        login.efetuarLogin();
        login.aguardarErro();
          expect(login.getMensagemErro()).toEqual('Usuário
ou senha inválidos');
    });
```

```
    it('Navegar para a página dos dados se o login ocor-
rer' () => {
        login.digitarUsername('meuUsuario');
        login.digitarSenha('senhaCerta');
        login.efetuarLogin();
        dados.aguardarAteSerVisivel();
    });
});
```

Fonte: Adaptado de Testing... (2019, documento *on-line*).

Antes de finalizar o caso de teste, você precisa fazer dois ajustes. O primeiro ajuste é criar um arquivo `environment.e2e.ts`, que será responsável por gerar um ambiente de execução para os testes.

Exemplo

```
export const environment = {
  production: false,
  databaseURL: 'url _ da _ base _ de _ dados',
  projectId: 'app _ de _ testes'
};
```

Fonte: Adaptado de Testing... (2019, documento *on-line*).

Depois, você deve modificar o arquivo angular.json para inserir o ambiente de testes que acabou de criar.

Exemplo

```
...
projects {
    app {
        architect {
            ...
            build {
                ...
                configurations {
                    "test": {
                        "fileReplacements": [
                        {
                            "replace": "src/environments/
environment.ts",
                               "with": "src/environments/
environment.e2e.ts"
                        }
                        ]
                    }
                }
            }
            ...
            serve {
                configurations {
                    "test": {
                        "browserTarget": "app:build:test"
                    }
                }
            }
            ...
```

```
            e2e {
                configurations {
                    "test": {
                      "devServerTarget": "app:serve:test"
                    }
                }
            }
            ...
        }
    }
}
```

Fonte: Adaptado de Testing... (2019, documento *on-line*).

Este é um exemplo simples de caso de teste para o Ionic, cujas ferramentas são poderosas o suficiente para abranger um padrão elevado de testes automatizados, sem, contudo, requerer desenvolvimentos de alta complexidade.

Referências

NORTH, D. Introducing BDD. *Dan North & Associates*, London, Mar. 2006. Disponível em: https://dannorth.net/introducing-bdd/. Acesso em: 30 jun. 2019.

PRESSMAN, R. S; MAXIM, B. R. *Engenharia de software*: uma abordagem profissional. 8. ed. Porto Alegre: AMGH; Bookman, 2016. 968 p.

TESTING — Ionic Documentation. *Ionic Framework*, [*S. l.*], 2019. Disponível em: https://ionicframework.com/docs/building/testing. Acesso em: 30 jun, 2019.

TUTORIAL. *Protractor — end to end testing for AngularJS*, [*S. l.*], 2018. Disponível em: https://www.protractortest.org/#/tutorial. Acesso em: 30 jun. 2019.

YOUR FIRST suíte — Jasmine Tutorials. *GitHub*, [*S. l.*], 2019. Disponível em: https://jasmine.github.io/tutorials/your_first_suite. Acesso em: 30 jun. 2019.

Leituras recomendadas

ANGULAR — Testing. *Angular. One framework. Mobile & desktop*, [*S. l.*], 2019. Documentação Oficial Angular: https://angular.io/guide/testing#component-test-basics. Acesso em: 30 jun. 2019.

MILCZEWSKI, M. Os princípios de testes E2E com Protractor. *TOTVS Developers — Medium*, [*S. l.*], 6 mar. 2019. Disponível em: https://medium.com/totvsdevelopers/os-princípios--de-testes-e2e-com-protractor-b6e70501158a. Acesso em: 30 jun. 2019.

NEWARD, T. O Programador Profissional — Como ser MEAN: Teste Angular. *Microsoft Developer Network Magazine*, [*S. l.*], v. 33, n. 11, Nov. 2018. Disponível em: https://msdn.microsoft.com/pt-br/magazine/mt830368.aspx?v=1021937632. Acesso em: 30 jun. 2019.

Encerra aqui o trecho do livro disponibilizado para esta Unidade de Aprendizagem. Na Biblioteca Virtual da Instituição, você encontra a obra na íntegra.

Conteúdo:

SAGAH
SOLUÇÕES EDUCACIONAIS INTEGRADAS

Dica do Professor

A integração de metodologias de desenvolvimento com os testes de *software* acaba por gerar novos padrões. Assim, surgiu o BDD (*Behaviour Driven Development)* ou Desenvolvimento orientado ao comportamento. Derivado do DDD (*Domain-Driven Development*), ou Desenvolvimento orientado ao domínio, no tocante às terminologias simples e ligadas direto às áreas de negócios; porém, focado em algo testável, no qual é possível escrever casos de teste de forma fácil e entendível por todos no projeto.

Nesta Dica do Professor, você vai entender um pouco sobre o BDD, de onde ele surgiu e vai poder fazer uma análise prática de uma *story*.

Aponte a câmera para o código e acesse o link do conteúdo ou clique no código para acessar.

Exercícios

1) **Para garantir a qualidade dos *softwares* lançados, é importante manter uma rotina de testes. Sobre os níveis de teste é correto afirmar que:**

A) o teste de integração é o nível mais abstrato, no qual é validada a integração do sistema com os outros sistemas da empresa.

B) o teste de unidade é aquele que analisa o *software* como uma unidade complexa em si, avaliando se esta unidade atende aos requisitos.

C) o teste de validação é o nível de testes, utilizado para validar funções, métodos, classes e objetos e suas entradas e saídas.

D) o teste de integração é o tipo de teste voltado para verificar a integração entre as unidades, ou seja, um bloco inteiro da aplicação.

E) o teste de sistema é todo aquele utilizado para testar sistemas computacionais, de maneira manual ou automatizada.

2) **Os testes de *software* são divididos em algumas técnicas fundamentais denominadas Teste de caixa branca e Teste de caixa preta. Sobre estes, é correto afirmar que:**

A) o teste de caixa branca exige que o testador tenha conhecimento do código fonte.

B) o teste de caixa preta exige que o testador tenha conhecimento do código fonte.

C) testes de caixa preta são indicados na fase de testes unitário e de integração.

D) testes de caixa branca são os mais indicados na fase de testes de aceitação e sistema.

E) testes de caixa branca são voltados para as fases de aceitação e validação.

3) **Codificar testes automatizados acaba por gerar uma necessidade de investimento de tempo e custos no projeto como um todo. Em relação a esses investimentos é correto afirmar que:**

A) quanto mais próximo ao nível do usuário, mais abrangente é o teste e maior será o investimento de tempo e dinheiro.

B) testes de integração de unidades são considerados os mais custosos em termos de tempo e dinheiro em um projeto.

C) o investimento de tempo e dinheiro é o mesmo em todas as fases de um projeto de testes de *software*.

D) testes unitários têm um custo maior (por teste), pois demandam maior conhecimento técnico de codificação.

E) os testes de interface de usuário não são possíveis de serem automatizados, pois requerem o uso da interface.

4) As aplicações desenvolvidas com o Ionic também são possíveis de serem testadas de maneira automatizada - inclusive, é recomendável. Sobre os testes com as plataformas do Ionic é correto afirmar que:

A) o Ionic possui funcionalidade interna para testes unitários, não dependendo de biblioteca de terceiros.

B) o Ionic integra a plataforma Jasmine e Protractor para testes unitários e *end-to-end*.

C) a programação do Ionic se dá em TypeScript, mas os testes, utilizando Jasmine, são codificados em JavaScript.

D) as bibliotecas de testes do Ionic são baseadas no JUnit e no Selenium para testes unitários e de integração respectivamente.

E) os elementos testáveis de maneira unitária no Ionic são os Services, Pipes e Components. As *pages* não são testáveis diretamente.

5) Por vezes, durante o desenvolvimento dos testes de unidade, é necessário acessar dados ou funções externas à unidade que se está testando. Sobre essa situação é correto afirmar que:

A) é uma prática que pode ser usada, porém, não é recomendada. Nesse caso, normalmente se trata de um erro no projeto de teste.

B) pode-se testar, normalmente, as funções e dados externos em testes unitários, da mesma forma que a aplicação faria.

C) para simular os dados e funções externos à unidade testada, faz-se o uso de *mocks*, evitando interferências externas no teste.

D) funcionalidades que dependem de outras não podem ser enquadradas em testes unitários, devendo ser tratadas apenas em testes de integração.

E) neste caso, o correto é buscar os dados necessários e mantê-los em memória; e, então, executar os testes unitários sobre os dados.

Na prática

Os testes de *software*, independentemente da plataforma utilizada, são uma realidade e devem estar presentes em todos os projetos. Discussões à parte, sobre se devem ser desenvolvidos antes ou depois do código, é necessário manter a qualidade da aplicação e a integridade dos dados.

Confira, Na Prática, o caso de uma fábrica de *softwares* que desenvolve aplicações médicas e como os testes garantiram o sucesso de uma aplicação e ajudaram a manter a boa imagem perante os clientes.

TESTES DE *SOFTWARE* REDUZEM GASTOS COM SUPORTE PÓS-IMPLANTAÇÃO E MELHORAM A IMAGEM DA EMPRESA PERANTE CLIENTES

Testes de software: investimento em qualidade

Empresa do setor de informática para a saúde cria *software* para gestão de resultados dos exames. Contudo, com o crescimento da plataforma e novas funcionalidades adicionadas, **o sistema começou a apresentar algumas falhas como: troca de imagens e laudos, permissão de acesso a exames indevidos, falha na geração do formulário para impressão**.

Os testes automatizados não foram desenvolvidos, logo de início, com o projeto. Ademais, após os problemas apresentados, **houve um aumento da demanda no setor de suporte da empresa e a redução da confiança dos clientes** – no caso, clínicas e hospitais – em relação ao produto e à empresa.

Visando reduzir essa demanda por suporte e recuperar a credibilidade, os testes foram implementados em etapas. Iniciando a partir dos testes unitários de novas funcionalidades e, em paralelo, foram implementados também nos elementos antigos.

Na sequência, foi realizada uma etapa de implementação de testes de integração, em que foram identificadas falhas na forma como as imagens e os laudos eram correlacionados.

Com o teste *end-to-end*, pôde-se verificar que a velocidade na geração de formulários para impressão estava insuficiente e acabava atingindo o limite de tempo – nesse caso, gerava uma falha e o laudo não era mostrado –, para isso, um plano de melhoria foi apresentado.

Os resultados dos testes demonstraram fragilidade em trechos do código, bem como na interface de usuários. Houve um investimento para a implantação da rotina de testes e esse investimento foi rapidamente recuperado, com o aumento da confiança na aplicação e com menor demanda de chamados de suporte.

Também foi reduzido o risco de erros médicos, ao descobrir a falha que, por vezes, mostrava imagens e laudos trocados.

Aponte a câmera para o código e acesse o link do conteúdo ou clique no código para acessar.

Saiba mais

Para ampliar o seu conhecimento a respeito desse assunto, veja abaixo as sugestões do professor:

Introduzindo o BDD

Compreenda melhor o padrão BDD, que é bem testável e compreensível para não-técnicos. É uma metodologia que está sendo amplamente adotada.

Aponte a câmera para o código e acesse o link do conteúdo ou clique no código para acessar.

Os princípios de testes E2E com Protractor

Veja nesse artigo do Mateus Milczewski, no blog da Totvs, algumas informações sobre testes E2E com o Protractor.

Aponte a câmera para o código e acesse o link do conteúdo ou clique no código para acessar.

O programador profissional - Como ser MEAN: Teste Angular

Confira este artigo, do Ted Newman, publicado no MSDN, sobre mecanismos de testes com Angular.

Aponte a câmera para o código e acesse o link do conteúdo ou clique no código para acessar.

Testando, princípios de teste

Veja este acesso à documentação oficial de testes do Ionic Framework (em Inglês).

Aponte a câmera para o código e acesse o link do conteúdo ou clique no código para acessar.

Metodologia de testes do Angular

Assim como o Ionic é uma derivação do Angular, só que para *mobile*, a maneira de testar também é bem semelhante. Nesta página, você pode conhecer um pouco mais da metodologia de testes do Angular (em Inglês).

Aponte a câmera para o código e acesse o link do conteúdo ou clique no código para acessar.

Teste e análise de *software* - Processos, princípios e técnicas

Neste livro, você poderá se aprofundar nos conceitos e metodologias de testes de *software*. O conteúdo é vasto e abrange, de forma ampla, diversas metodologias.

Conteúdo interativo disponível na plataforma de ensino!