Indo Além com Automação e Testes de Apps Android

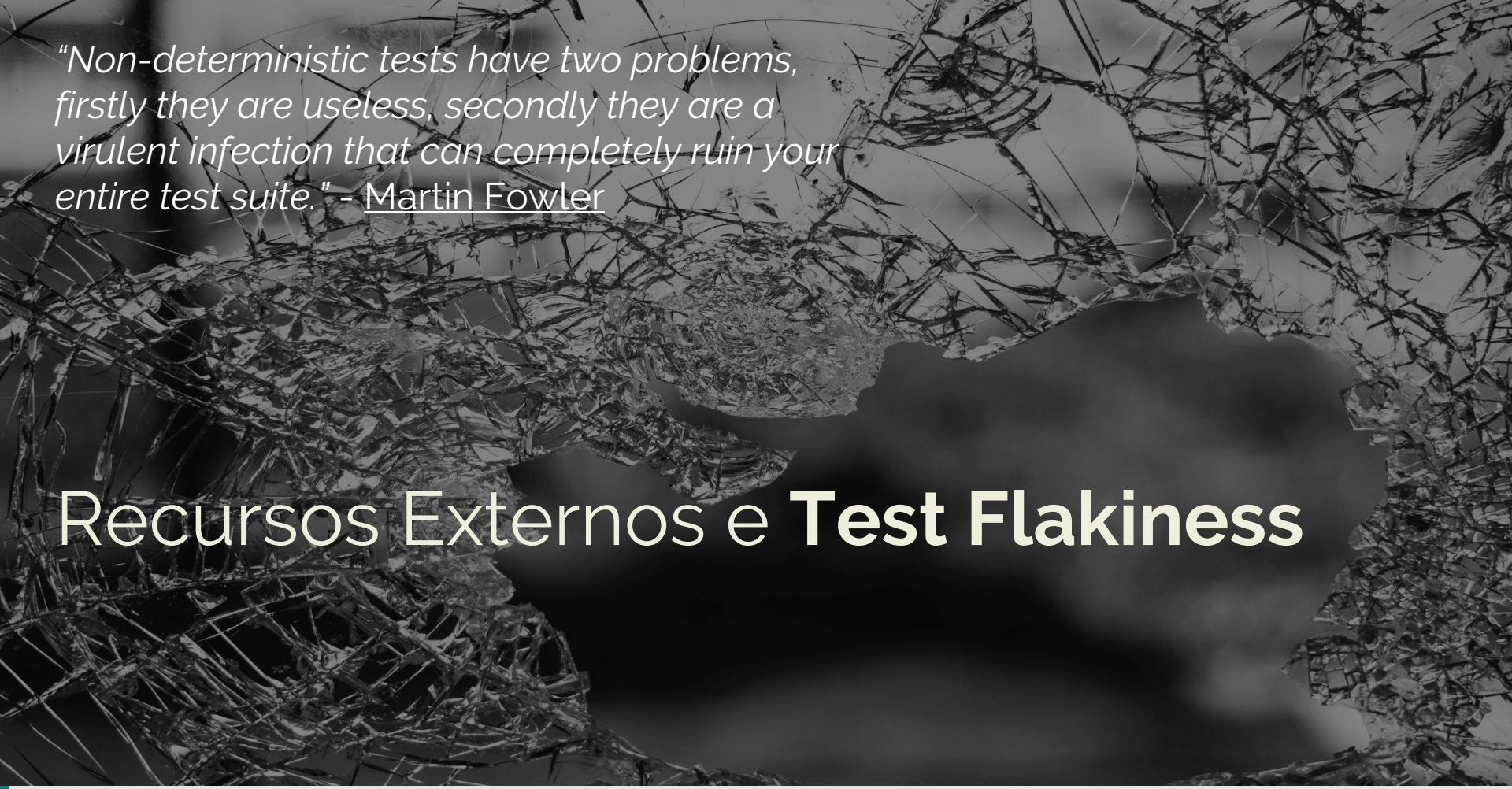
Eduardo Carrara
@DuCarrara

Você já se sentiu assim em seus projetos?


Infelizmente não há **bala de prata**...


Precisamos de **segurança**,
para **Iterações Rápidas.**


Testes Automatizados podem ajudar
Então o que vamos explorar hoje?
Recursos Externos & Test Flakiness
Simplificação de Dependências com Mocks
UI Tests e as Camadas de Testes



Recursos Externos e **Test Flakiness**

Test **Flakiness**

Falham e passam aleatoriamente

Escondem bugs

Diminuem a confiança na suite de testes

São difíceis de corrigir

Principais **Causas**

Concorrência

Dependência de comportamentos indefinidos ou não determinísticos

Código de terceiros que falham e funcionam aleatoriamente

Problemas de infraestrutura

<adicione sua causa aqui>

O que acontece no **Android**?

Rede

Recursos de Hardware do telefone (ex.: sensores)

Interação com outros apps

Dependências de serviços, apis e sdks externos

Exemplificando

*Caso comum: chamadas a **serviços REST***

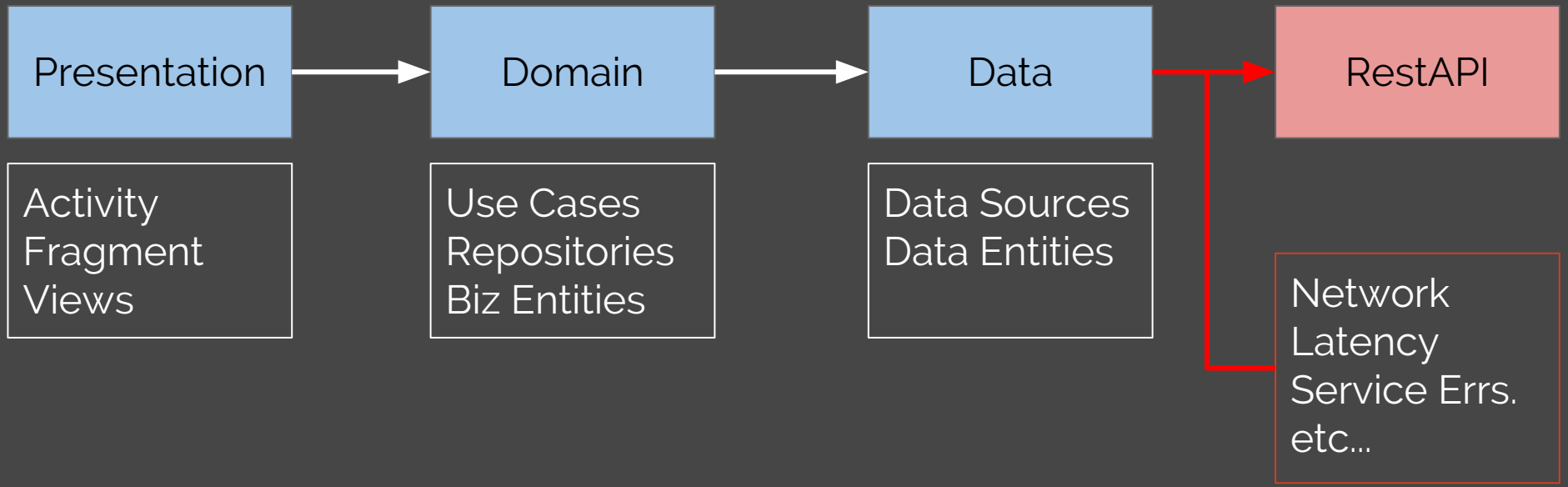
***Dependência** de recursos **de rede**, disponibilidade **de serviços** etc*

***Como resolver** o flakiness no teste da chamada de um serviço?*

*Resposta: **Desacoplar e Isolar***

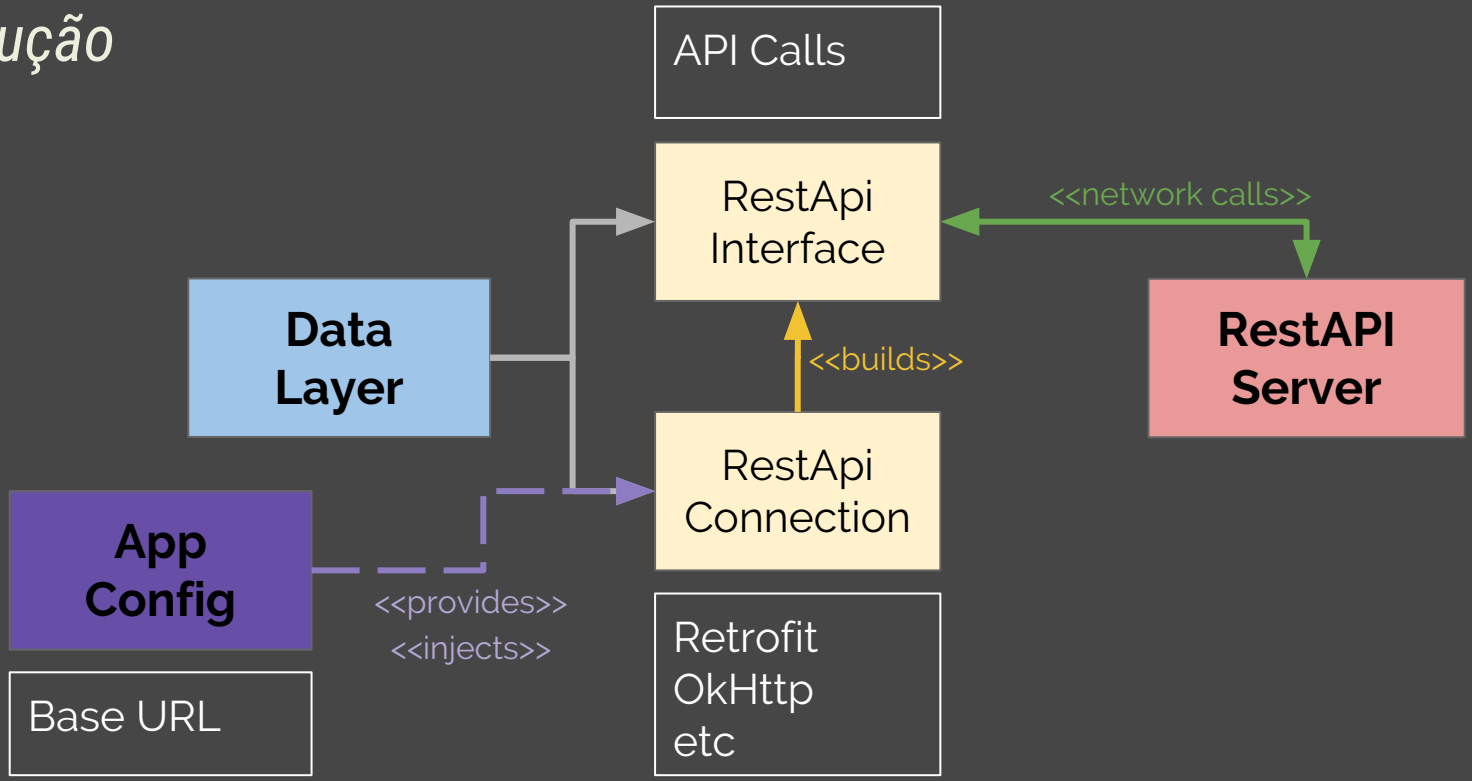
Desacoplando e Isolando

Origem do flakiness



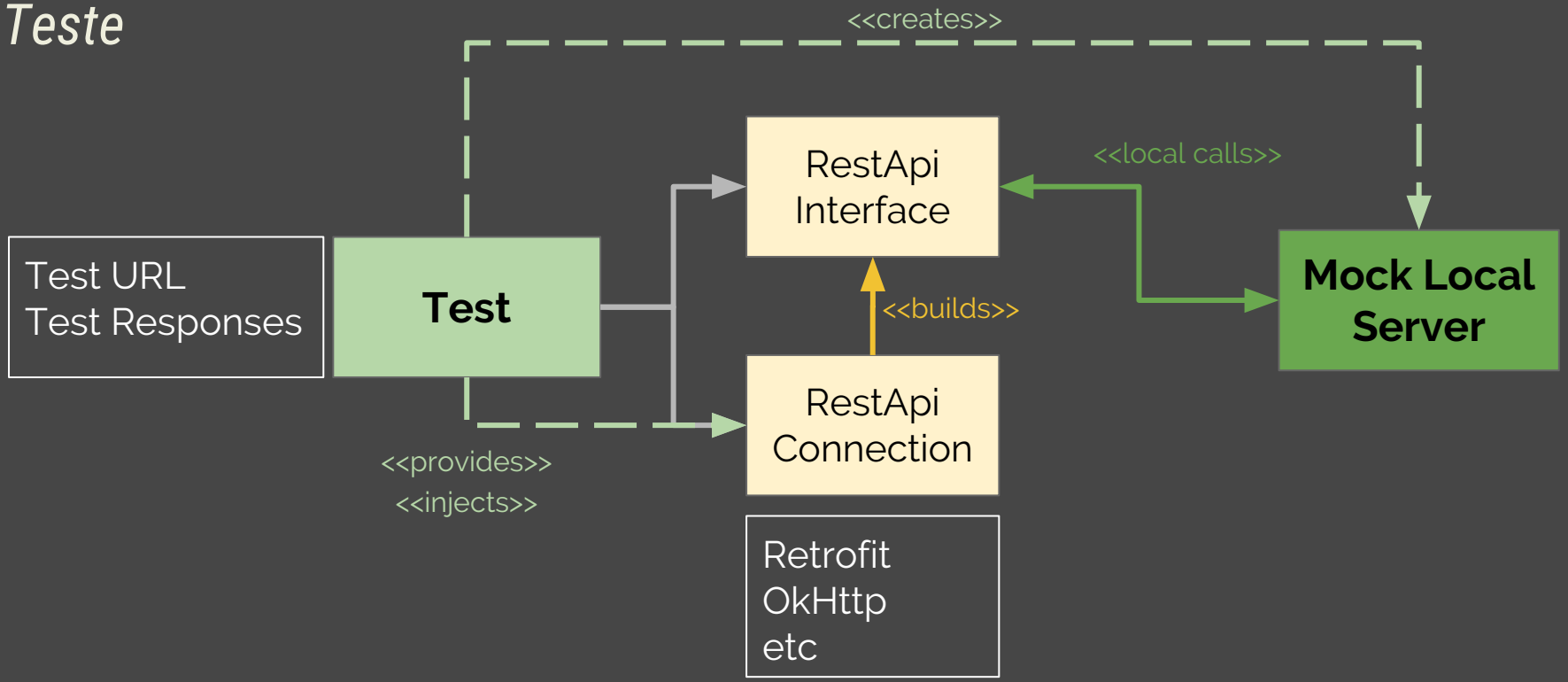
Desacoplando e Isolando

Produção



Desacoplando e Isolando

Teste



E o código...

Com Retrofit + OkHttp ***+ MockWebServer***

```
dependencies {
    testCompile 'com.squareup.okhttp3:mockwebserver:3.3.0'
}
```

E o código...

Exemplo em NEOM on Github: ListNearEarthObjectsTest.java

```
@Before public void setUp() throws IOException {
    prepareExpectedTestData();
    server = new MockWebServer();
    server.enqueue(
            new MockResponse()
                    .setBody(TestServiceResponseFor20150907.getJson())
    );
    server.start();
    restApi = new RestApi(server.url("/").toString());
}

@After public void tearDown() throws IOException {
    server.shutdown();
}
```

E o código...

Exemplo em NEOM on Github: ListNearEarthObjectsTest.java

```
@Test
public void testSuccessfulListNearEarthObjects() throws Exception {
    List<NearEarthObjectJsonEntity> nearEarthObjectJsonEntityList =
            restApi.listNearEarthObjectsFor(requestDate);

    assertThat(nearEarthObjectJsonEntityList,
            is(expectedNearEarthObjectJsonEntityList));
}
```

Em nosso exemplo...

Isolamos a chamada à nossos serviços

Substituímos o servidor por um componente sob nosso controle

Podemos testar esta camada com testes unitários

Tomamos controle sobre um recurso que poderia ser Flaky

Simplificação com Mocks

Dependências

+ Dependências = + Acoplamento

Alto acoplamento dificulta a criação de testes e a automação

Sistemas 100% desacoplados são raros

Mocking

Objetos dublês

Controle sobre o comportamento e estado

Isolamento da unidade a ser testada

Deve nos ajudar a lidar com as dependências de maneira simples

O que acontece no **Android**?

Dependência de componentes do framework é frequente

Ex.: Context, Dados de Sensores, Services, Activities etc

Complexidade dos testes aumenta muito

Mockito

API fácil de entender

Configuração de comportamentos e Stubs

Permite a verificação de comportamentos

Mas como funciona?

Exemplo em NEOM on Github: NearEarthObjectListPresenterTest.java

```
@RunWith(MockitoJUnitRunner.class)
public class NearEarthObjectListPresenterTest {

    NearEarthObjectListPresenter presenter;
    @Mock private ObservableInteractor<List<NearEarthObject>> mockInteractor;
    @Mock private NearEarthObjectsListView mockView;

    @Captor
    private ArgumentCaptor<Observer<List<NearEarthObject>>> observerArgumentCaptor;

    @Before private void setUp() {
        presenter = new NearEarthObjectListPresenter(mockInteractor);
    }
}
```

Mas como funciona?

Exemplo em NEOM on Github: NearEarthObjectListPresenterTest.java

```
@Test public void testSuccessfulPresenterInitialization() {
    presenter.attachTo(mockView);
    verify(mockView).hideRetry();
    verify(mockView).showLoading();
    // Here we capture the observer registration to mock the update call
    verify(mockInteractor, times(1))
        .addObserver(observerArgumentCaptor.capture());
    verify(mockInteractor).execute();

    observerArgumentCaptor.getValue()
            .update(mockInteractor, new ArrayList<NearEarthObject>());

    verify(mockView).renderNearEarthObjectsList(any(List.class));
    verify(mockView).hideLoading();
}
```

Robolectric

Testes que dependem do Android na JVM

Provê mocks e stubs ao Android SDK

Execução mais rápida de testes

Demora para disponibilizar versões mais recentes do SDK

Mas como funciona?

Exemplo em NEOM on Github: NearEarthObjectsListActivityTest.java

```
@RunWith(RobolectricGradleTestRunner.class)
@Config(constants = BuildConfig.class, sdk = Build.VERSION_CODES.LOLLIPOP)
public class NearEarthObjectsListActivityTest {

    private NearEarthObjectsListActivity nearEarthObjectsListActivity;

    @Before
    public void setup() {
        nearEarthObjectsListActivity =
                Robolectric.setupActivity(NearEarthObjectsListActivity.class);
    }
}
```

Mas como funciona?

Exemplo em NEOM on Github: NearEarthObjectsListActivityTest.java

```
@Test
public void testErrorMessageDisplay() {
    final String expectedErrorMessage = "Error message";

    nearEarthObjectsListActivity.showError(expectedErrorMessage);

    View errorView = findById(nearEarthObjectsListActivity, R.id.error_view);
    TextView errorTextView = findById(nearEarthObjectsListActivity,
            R.id.error_message_text_view);

    assertThat(errorView.getVisibility(), is(VISIBLE));
    assertThat(errorTextView.getText().toString(), is(expectedErrorMessage));
}
```

Combinação Poderosa

Testes Rodando na JVM Local = Produtividade

Ajuda com dependências indiretas do Android Framework

Permite o mocking e uso de classes do Android na JVM Local

Exemplificando

*Caso: Uso de dados obtidos de um **Content Provider***

O Content Provider nos provê um Cursor

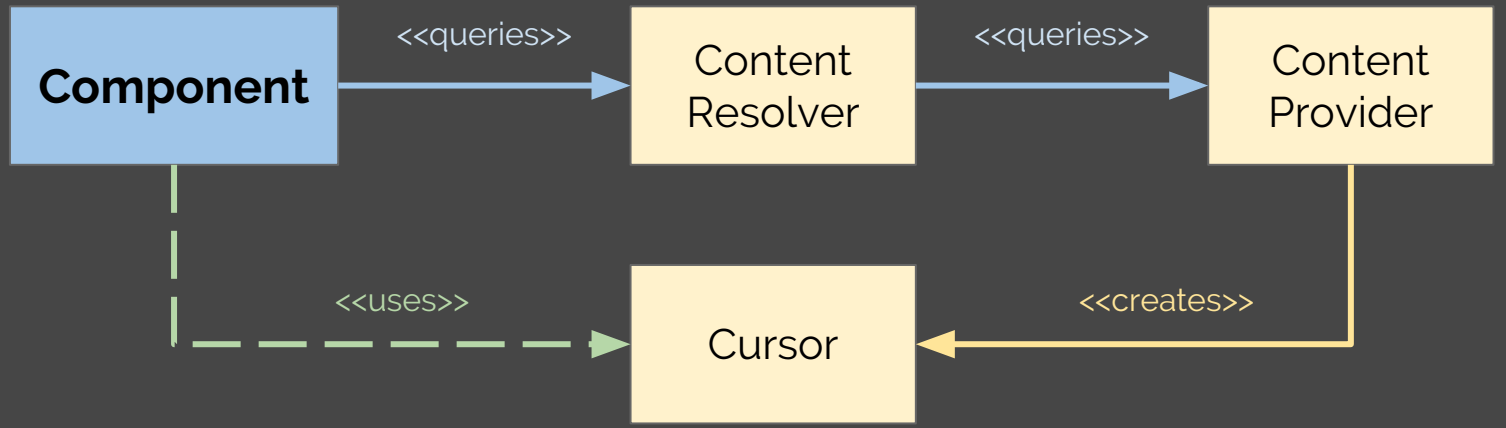
*Como testar um componente que dependa de um **Cursor**?*

*Resposta: Mocks**

** A classe MockCursor do Android Framework foi Deprecated na API Level 24. Uma das sugestões é usar o Mockito com ATSL.*

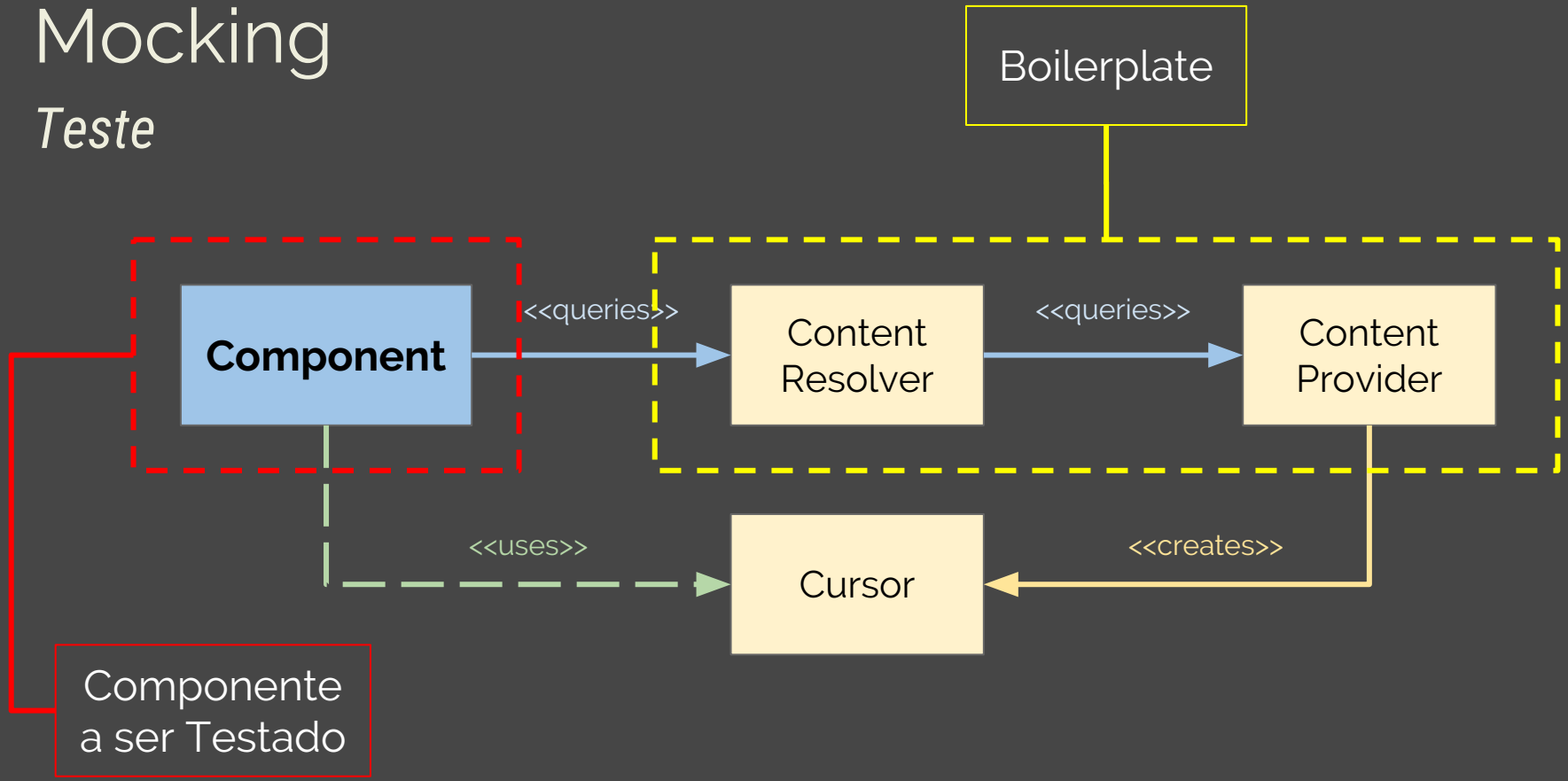
Mocking

Produção



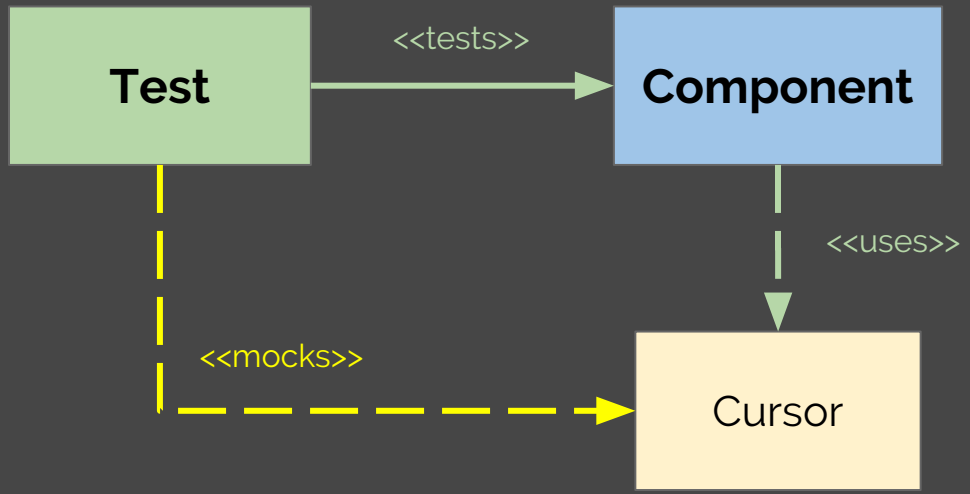
Mocking

Teste



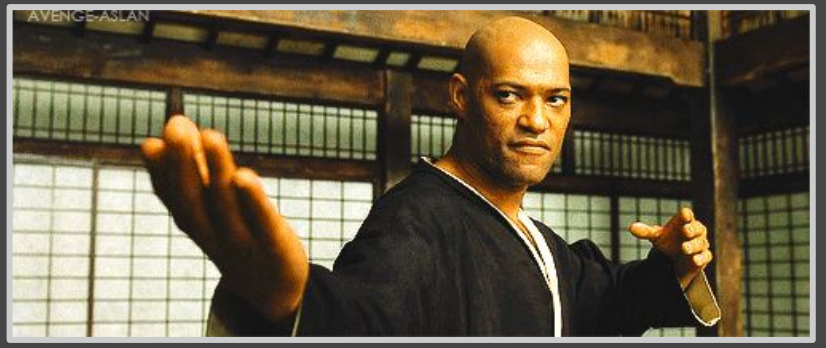
Mocking

Teste



E o código...

Com Mockito + Robolectric




```
dependencies {
    testCompile 'org.robolectric:robolectric:3.0'
    testCompile 'org.mockito:mockito-core:2.+'
}
```

E o código...

Exemplo em Vilibra on Github: BookBorrowingContentProviderMapperTest.java

```
@RunWith(RobolectricGradleTestRunner.class)
@Config(constants = BuildConfig.class)
public class BookBorrowingContentProviderMapperTest {

    @Before public void prepareTest() {
        this.bookBorrowingContentProviderMapper = new BookBorrowingContentProviderMapper();
    }

    @Test public void testTransformEmptyCursorToBookBorrowing() throws Exception {
        Cursor cursor = mock(Cursor.class);
        when(cursor.moveToFirst()).thenReturn(false);

        List<BookBorrowing> bookBorrowings =
                this.bookBorrowingContentProviderMapper.transform(cursor);

        assertThat(bookBorrowings, is(empty()));
    }
}
```

Em nosso exemplo...

Isolamos o componente a ser testado de dependências boilerplate

Usamos mocks para criar as situações de teste desejadas

Podemos rodar nossos testes como testes unitários na JVM

Usando mocks lembre-se...

Se você cria mocks para tudo você não testa nada

Mocks nos ajudam a isolar o componente a ser testado e

a ter controle sobre dependências e comportamentos externos

Podem ajudar a reduzir flakiness

UI Tests e as **Camadas de Testes**

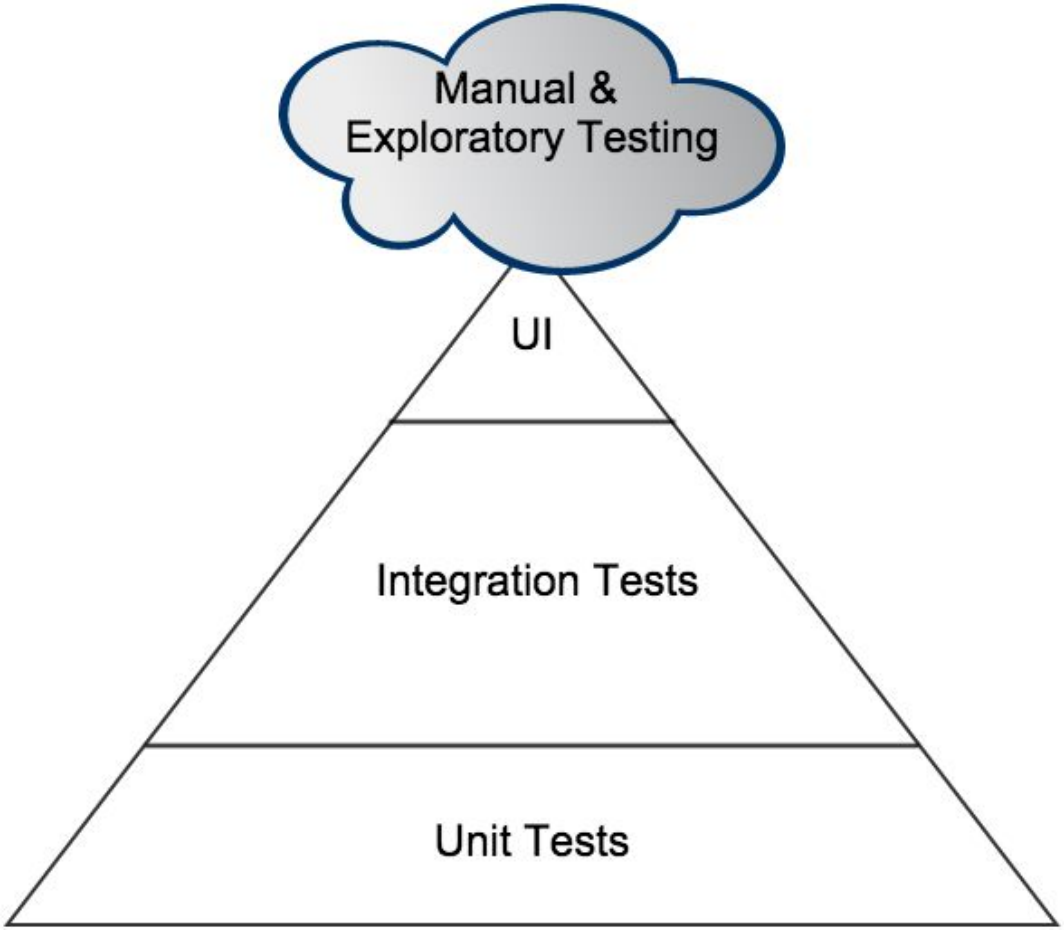
Camadas de Testes

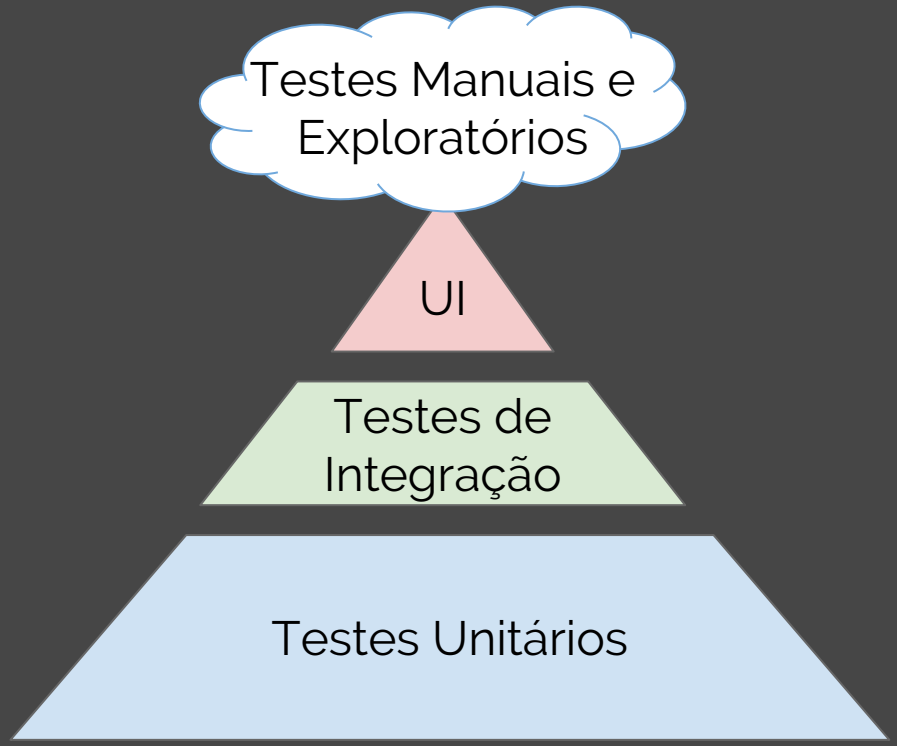
Estratégia para automação de testes

Onde investir nossos recursos para automação?

Layers of Test by Mike Cohn

Fonte: Layers of Test Automation by Stuart Ashman





	Custo	Tempo
Testes Manuais e Exploratórios	$$$$	🕒🕒🕒🕒
UI	$$$	🕒🕒🕒
Testes de Integração	$$	🕒🕒
Testes Unitários	$	🕒
	Criação e Manutenção	Execução e Criação

Testes de Interface do Usuário (UI)

Devem focar em funcionalidades, user stories, casos de uso etc

Testes do ponto de vista do usuário (UI)

Retira o trabalho repetitivo do time

QA pode se dedicar a testes mais complexos

Algumas Ferramentas

ATST

appium

selendroid
SELENIUM FOR ANDROID

Robotium

Calaba.sh

Android **T**esting **S**upport **L**ibrary

Execução relativamente rápida

Bem próximo da plataforma e exige conhecimentos de dev.

Utiliza as mesmas ferramentas que usamos para criar as apps

Difícil para não desenvolvedores

Android Testing Support Library

Ferramentas específicas para testes de componentes Android

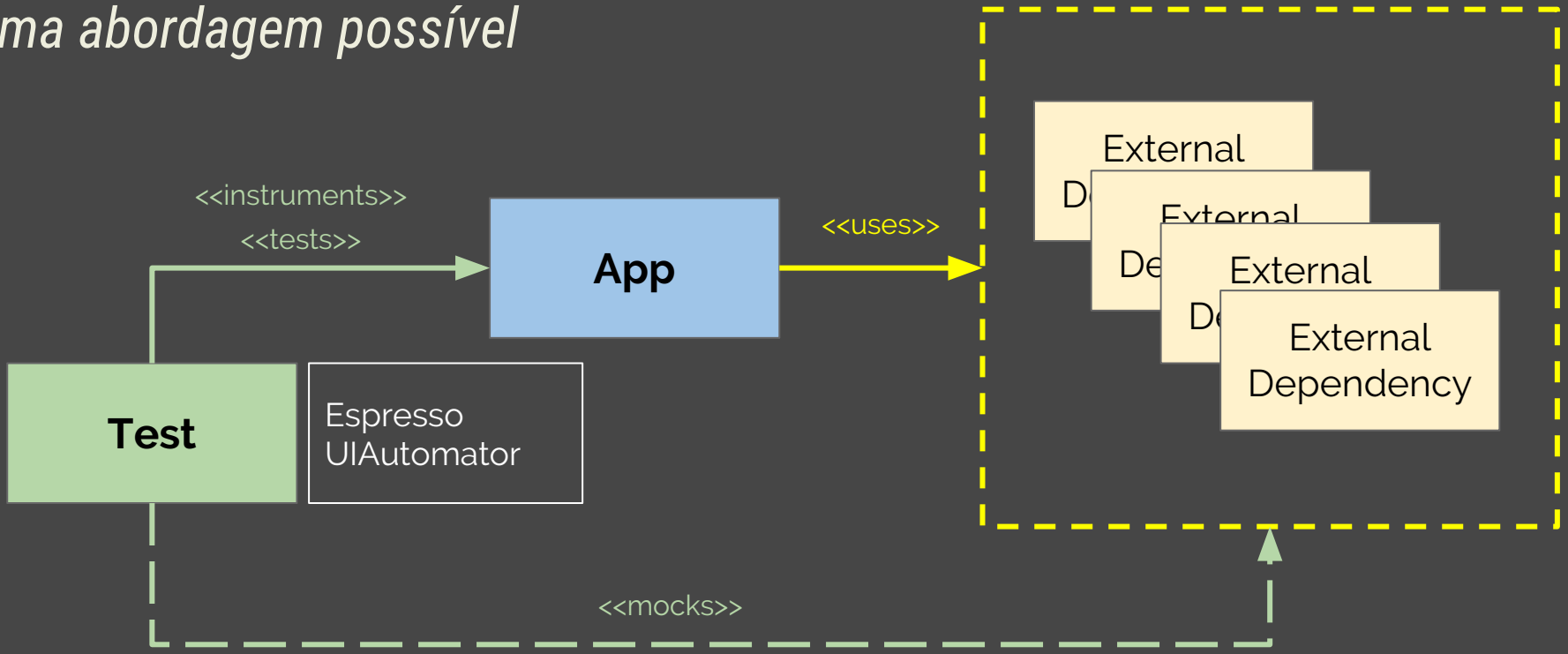
AndroidJUnitRunner + JUnit4 Rules + Espresso + UIAutomator

*Em geral compatível com **Api Level 8** ou superior*

*UIAutomator é compatível com **Api Level 18** ou superior*

UI Tests

Uma abordagem possível



Um exemplo simples...

Exemplo em Vilibra on Github: TestEmptyLoanedBooksList.java

```
android {
    defaultConfig{
        testInstrumentationRunner "android.support.test.runner.AndroidJUnitRunner"
    }
}
```

```
dependencies {
    androidTestCompile 'com.android.support.test:runner:0.5'
    androidTestCompile 'com.android.support.test:rules:0.5'
    androidTestCompile 'com.android.support.test.espresso:espresso-core:2.2.2'
    androidTestCompile 'com.android.support.test.uiautomator:uiautomator-v18:2.1.2'
}
```

Um exemplo simples...

Exemplo em Vilibra on Github: TestEmptyLoanedBooksList.java

```
@RunWith(AndroidJUnit4.class)
public class TestEmptyLoanedBooksList {

   private Context context;

   @Rule public ActivityTestRule<BookListActivity> activityRule =
           new ActivityTestRule<>(BookListActivity class);

   @Before public void setUp() {
       this.context = getTargetContext();
       clearTestData();
   }
   @Test public void simpleTestEmptyStateOfLoanedBooksList() {
       onView(withId(R.id.loaned_book_list_view)).check(matches(not(isDisplayed())));
       onView(withId(R.id.empty_book_list_icon)).check(matches(isDisplayed()));
   }
}
```

Testes de Interface do Usuário (UI)

São difíceis de criar e manter

Sua execução geralmente é mais lenta, deixe-os para o final

Selecione a ferramenta adequada aos conhecimentos do time

Quer saber mais? Dê uma olhada em minhas outras apresentações!

Pensamentos Finais

Mantenha o Test Flakiness sob controle

Mocks podem aumentar sua produtividade e controlar o flakiness

Testes de UI são difíceis de criar e manter, use-os sabiamente

Cuide de seus códigos de testes assim como você cuida de seus códigos de produção!

Perguntas?

Material em:

http://goo.gl/oms4Hs

"... if you are afraid to change something it is clearly poorly designed."

- Martin Fowler

Obrigado!

Eduardo Carrara

@DuCarrara

github.com/ecarrara-araujo

Referências

1. Building Effective Unit Tests by Android Documentation
2. How to Deal With and Eliminate Flaky Tests
3. Flaky Tests at Google and How We Mitigate Them
4. Eradicating Non-Determinism in Tests by Martin Fowler
5. Retrofit 2 – Mocking HTTP Responses
6. The Difference Between Mocks and Stubs by Martin Fowler
7. Mockito
8. Robolectric
9. Layers of Test Automation
10. Android Test Support Library
11. Espresso Test Recorder Docs