

TÓPICOS AVANÇADOS DE  
DESENVOLVIMENTO ANDROID

# TESTES UNITÁRIOS E DE INTERFACE ANDROID

RICARDO DA SILVA OGLIARI



## LISTA DE FIGURAS

Figura 6.1 - Tela da aplicação .....	6
Figura 6.2 - Source sets do projeto .....	10
Figura 6.3 - Relatório da execução dos nossos primeiros testes .....	11
Figura 6.4 – Relatório da execução dos testes instrumentados .....	12
Figura 6.5 - Geração de um novo teste .....	15
Figura 6.6 - Janela de configurações do novo teste .....	15
Figura 6.7 - Test Driven Development.....	17
Figura 6.8 - Falha em um dos testes .....	18
Figura 6.9 - Escolha do androidTest para testes de UI .....	21
Figura 6.10 - Resultado de falha no teste de UI .....	25
Figura 6.11 - Resultado de sucesso no teste de UI.....	26

## LISTA DE CÓDIGOS-FONTE

Código-fonte 6.1 – Conteúdo do arquivo colors.xml.....	6
Código-fonte 6.2 – Conteúdo do arquivo dimens.xml.....	7
Código-fonte 6.3 – Conteúdo do arquivo strings.xml.....	7
Código-fonte 6.4 – Conteúdo do arquivo activity_main.xml.....	8
Código-fonte 6.5 – Conteúdo do arquivo strings.xml.....	9
Código-fonte 6.6 – Conteúdo do arquivo ExampleUnitTest.....	11
Código-fonte 6.7 – Conteúdo do arquivo ExampleUnitTest.....	12
Código-fonte 6.8 – Conteúdo do arquivo Util.kt.....	14
Código-fonte 6.9 – Novo conteúdo da MainActivity.....	14
Código-fonte 6.10 – Nova classe de teste UtilTest.....	16
Código-fonte 6.11 – Nova constante no Enum Result.....	17
Código-fonte 6.12 – Novo teste de falha codificado.....	18
Código-fonte 6.13 – Correção da função getImcResult.....	19
Código-fonte 6.14 – Adição da cor vermelha no colors.xml.....	20
Código-fonte 6.15 – Mudanças na activity_main.xml.....	20
Código-fonte 6.16 – Definição da regra na classe de teste.....	22
Código-fonte 6.17 – Versão final da MainScreentest.....	24
Código-fonte 6.18 – Versão final da MainActivity.....	26

## SUMÁRIO

6 TESTES UNITÁRIOS E INTERFACE ANDROID .....	5
6.1 Introdução .....	5
6.2 A aplicação de índice de massa corporal (IMC) .....	5
6.3 Criação do esboço do projeto.....	6
6.4 Primeiros passos com testes.....	9
6.5 Implementando teste local.....	12
6.6 Aplicando conceitos de Test Driven Development .....	16
6.7 Testes de Interface.....	19
CONCLUSÃO.....	27
REFERÊNCIAS.....	28

## 6 TESTES UNITÁRIOS E INTERFACE ANDROID

### 6.1 Introdução

No desenvolvimento de aplicativos em ambientes com muitas atualizações e aplicações complexas, os testes são cada vez mais necessários. As razões para isso são diversas: automatização dos testes e menor risco de falhas em testes humanos; um controle maior nas atualizações futuras e menor chance de efeitos colaterais em mudanças no código; desenvolvimento orientado a testes e o consequente aumento da qualidade de código; até mesmo, a modernização do mercado mobile e a necessidade de um fluxo de entrega contínua e segura.

Por essas razões, é comum vermos conceitos ganhando espaço no mundo de desenvolvimento de aplicativos em geral e, em especial, no Android. Como exemplo, podemos citar o desenvolvimento orientado a testes ou o TDD (Test-Driven Development).

Como de praxe, a biblioteca Android e suas IDEs (Integrated Development Environment) fornecem diversas facilidades para os desenvolvedores. Esse é o foco deste capítulo. Veremos como cobrir os principais testes em um aplicativo Android, usando, para isso, a IDE Android Studio.

### 6.2 A aplicação de índice de massa corporal (IMC)

Como base de nossos testes, vamos partir de uma aplicação simples para o cálculo do Índice de Massa Corporal (IMC), em que temos apenas uma tela para inserimos os dados referentes ao peso e altura. Ao clicar no botão *Calcular* usamos a fórmula do IMC para mostrar a situação do usuário em relação a sua saúde.



Figura 6.1 - Tela da aplicação  
Fonte: Elaborado pelo autor (2020)

### 6.3 Criação do esboço do projeto

Crie um projeto Android normalmente com o nome de *CalculoIMC*. Na escolha da *Activity*, opte por *Empty Activity*, e nas outras configurações, use os valores padrão.

Depois que a IDE fizer o trabalho inicial e criar todas as pastas e arquivos iniciais do projeto, vamos mudar alguns valores dos arquivos de *resources* que serão usados na aplicação. Na sequência, veremos: *colors.xml*, *dimens.xml* e *strings.xml*.

```
<?xml version="1.0" encoding="utf-8"?>
<resources>
    <color name="colorPrimary">#6200EE</color>
    <color name="colorPrimaryDark">#3700B3</color>
    <color name="colorAccent">#000000</color>
</resources>
```

Código-fonte 6.1 – Conteúdo do arquivo colors.xml  
Fonte: Elaborado pelo autor (2020)

```
<?xml version="1.0" encoding="utf-8"?>
<resources>
    <dimen name="padding_main_screen">10dp</dimen>
    <dimen name="label_imc">18sp</dimen>
</resources>
```

Código-fonte 6.2 – Conteúdo do arquivo dimens.xml  
Fonte: Elaborado pelo autor (2020)

```
<resources>
    <string name="app_name">CalculoIMC</string>
    <string name="peso">Peso</string>
    <string name="altura">Altura</string>
    <string name="calcular">Calcular</string>
</resources>
```

Código-fonte 6.3 – Conteúdo do arquivo strings.xml  
Fonte: Elaborado pelo autor (2020)

Vamos alterar o layout da única tela e definir o código completo, pelo menos aquele usado na primeira etapa do capítulo. Veja o próximo Código-fonte “Conteúdo do arquivo activity\_main.xml”.

Usamos o gerenciador de layout *LinearLayout* porque temos uma tela linear vertical, sem um nível de complexo grande na tela. Nos dois primeiros itens usamos o *TextInputLayout* para uma interface um pouco mais agradável. Por fim, um *Button* e um *TextView*. Perceba que tomamos cuidado de não deixar nenhum recurso *harcoded*.

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout
    xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:tools="http://schemas.android.com/tools"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:orientation="vertical"
    android:padding="@dimen/padding_main_screen"
    tools:context=".MainActivity">
    <com.google.android.material.textfield.TextInputLayout
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:hint="@string/peso">

    <com.google.android.material.textfield.TextInputEditText
        android:inputType="numberDecimal"
        android:id="@+id/edt_peso"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"/>
    </com.google.android.material.textfield.TextInputLayout>
    <com.google.android.material.textfield.TextInputLayout
```

```
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:hint="@string/altura">

<com.google.android.material.textfield.TextInputEditText
    android:id="@+id/edt_altura"
    android:inputType="numberDecimal"
    android:layout_width="match_parent"
    android:layout_height="wrap_content"/>

</com.google.android.material.textfield.TextInputLayout>

<Button
    android:id="@+id/btn_make_calc"
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
    android:text="@string/calcular"/>

<TextView
    android:textSize="@dimen/label_imc"
    android:layout_marginTop="10dp"
    android:id="@+id/txt_result_imc"
    android:layout_width="match_parent"
    android:layout_height="wrap_content"/>

</LinearLayout>
```

Código-fonte 6.4 – Conteúdo do arquivo activity\_main.xml

Fonte: Elaborado pelo autor (2020)

Por usarmos o *TextInputLayout*, é necessário também adicionar mais uma dependência ao projeto. Sendo assim, no arquivo *build.gradle*, na parte de *dependencies*, é necessário adicionar a linha abaixo. E, conseqüentemente, depois disso clicar no *Sync Now*, para re-sincronizar o projeto e baixarmos a nova biblioteca. A linha é: *compile 'com.android.support:design:25.3.1'*

Finalmente a última alteração feita é na *MainActivity*. Confira o Código-fonte “Conteúdo do arquivo strings.xml”. Além do código já ter sido criado pela IDE, estamos respondendo ao evento de clique no elemento com id *btn\_make\_calc*.

No bloco de código do *onClickListener* estamos lendo os valores inseridos nos campos de texto; depois, fizemos os casts necessários para transformar as Strings em floats; na sequência, usamos a fórmula de cálculo do IMC para chegar ao resultado final e informamos ao usuário qual seu status conforme o peso e a altura.



```

import androidx.appcompat.app.AppCompatActivity
import android.os.Bundle
import kotlinx.android.synthetic.main.activity_main.*

class MainActivity : AppCompatActivity() {
    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContentView(R.layout.activity_main)

        btn_make_calc.setOnClickListener {
            val pesoTxt = edt_peso.text.toString()
            val alturaTxt = edt_altura.text.toString()

            val peso = pesoTxt.toFloat()
            val altura = alturaTxt.toFloat()

            val imc = peso / (altura * altura)

            if (imc < 16){
                txt_result_imc.text = "Magreza grave"
            } else if (imc < 17){
                txt_result_imc.text = "Magreza moderada"
            } else if (imc < 18.5){
                txt_result_imc.text = "Magreza leve"
            } else if (imc < 25){
                txt_result_imc.text = "Saudável"
            } else if (imc < 30){
                txt_result_imc.text = "Sobrepeso"
            } else if (imc < 35){
                txt_result_imc.text = "Obesidade Grau I"
            } else if (imc < 40){
                txt_result_imc.text = "Obesidade Grau II
(severa)"
            } else {
                txt_result_imc.text = "Obesidade Grau III
(mórbida)"
            }
        }
    }
}

```

Código-fonte 6.5 – Conteúdo do arquivo strings.xml

Fonte: Elaborado pelo autor (2020)

## 6.4 Primeiros passos com testes

O Android Studio já traz toda uma configuração pronta para testes. Perceba que o projeto criado apresenta três pastas principais, que são conhecidas como *source sets*. O primeiro deles contém a *MainActivity.kt*, que estudamos anteriormente.

As duas pastas seguintes são os testes instrumentados e os testes locais. Perceba que nestas duas pastas também tem uma especificação do lado direito, escrito entre parênteses: (*androidTest* e *test*).

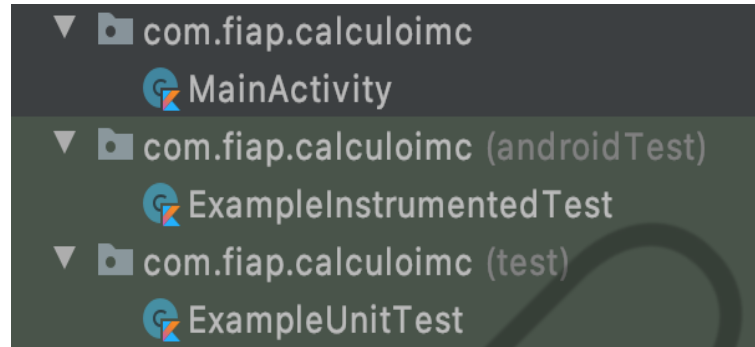


Figura 6.2 - Source sets do projeto  
Fonte: Elaborado pelo autor (2020)

A diferença entre os dois tipos de testes é basicamente a necessidade de um emulador ou de um dispositivo físico, no caso dos testes instrumentados, ou uso apenas da JVM presente na máquina local. Ou seja, nos testes locais ganhamos em tempo de execução, porém perdemos na relação de fidelidade com o ambiente porque estamos em um PC ou notebook, não no próprio dispositivo móvel.

Abra o arquivo *ExampleUnitTest* para nosso primeiro contato com código de teste. Seu conteúdo é mostrado no Código-fonte “Conteúdo do arquivo *ExampleUnitTest*”. Se você está tendo seu primeiro contato com este conceito, imagino que tenha ficado impressionado com a simplicidade que viu. Temos apenas uma classe, com uma única função (*addition\_isCorrect*) e, no seu interior, um código que deixa bem claro que está checando se o valor esperado (primeiro parâmetro) é igual ao atual (segundo parâmetro).

Um detalhe que já chama a atenção é o uso da anotação *Test*, que faz parte do pacote *JUnit*. Outro detalhe é a documentação da classe, que já deixa claro o que é um teste local.

```
/**
 * Example local unit test, which will execute on the
 * development machine (host).
 *
 * See [testing
 * documentation] (http://d.android.com/tools/testing).
 */
class ExampleUnitTest {
    @Test
```

```
fun addition_isCorrect() {  
    assertEquals(4, 2 + 2)  
}
```

Código-fonte 6.6 – Conteúdo do arquivo ExampleUnitTest

Fonte: Elaborado pelo autor (2020)

Agora, com o botão direito do mouse, clique na mesma classe e depois clique em *Run ExampleUnitTest*. Os testes passarão e teremos um relatório completo, mostrado na parte do rodapé da IDE. Para ser mais específico, estará na aba *run*.

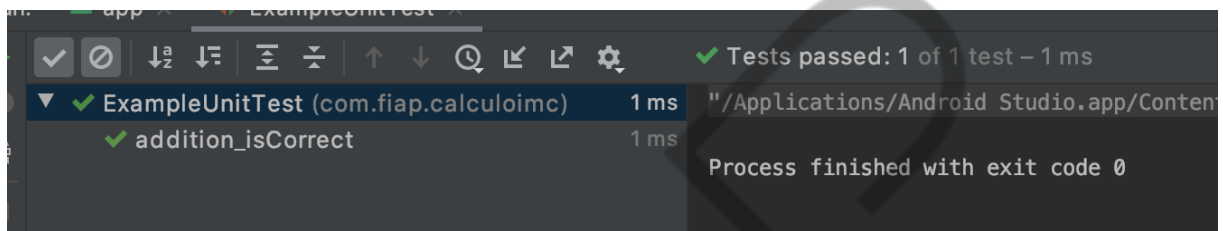


Figura 6.3 - Relatório da execução dos nossos primeiros testes

Fonte: Elaborado pelo autor (2020)

Faça um novo teste, mas alterando o valor esperado na função de teste. Será possível ver como a IDE responde a uma falha, em algum teste. O Android Studio também nos permite rodar os testes de diferentes lugares, não só com o botão direito do mouse, no nome da classe, como fizemos anteriormente. Algumas delas são:

- No código-fonte, clicando com o botão direito no nome da classe ou no nome da função, para rodar todos os testes na classe ou só de um método específico.
- Na parte inferior, na aba de run, clicando com o botão direito também no nome da classe ou da função.

Agora rode o teste instrumentado e veja o resultado na mesma aba *run*. Confira o resultado na Figura “Relatório da execução dos testes instrumentados”. Veja no log que o teste rodou no aparelho que estava conectado ao meu notebook, um Samsung J4. Outra prova disso é o próprio uso do ADB (*Android Debug Bridge*), que fica explícito no log também.

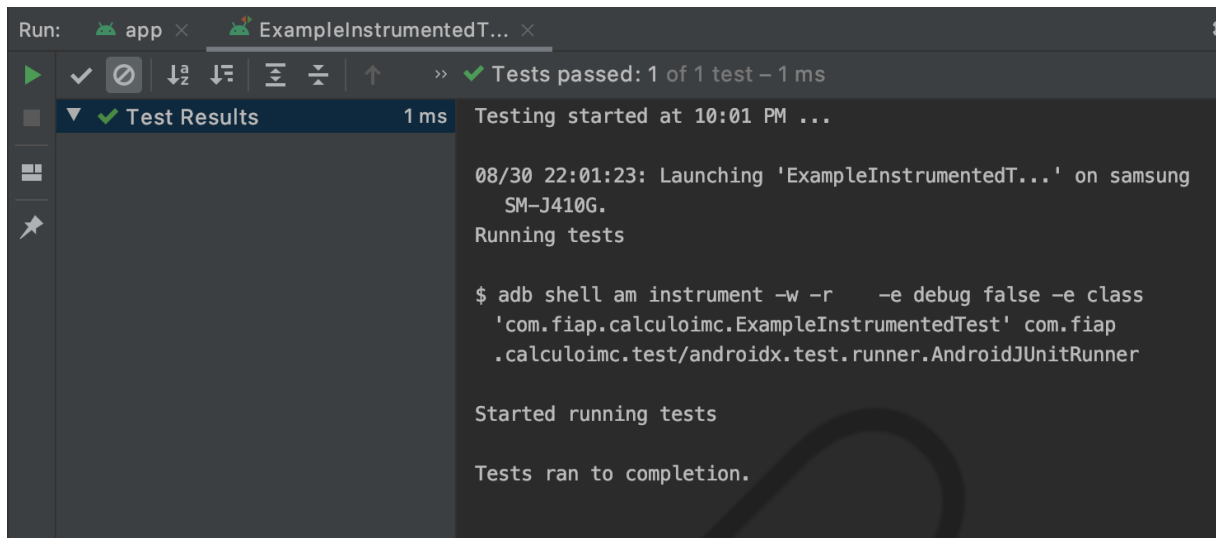


Figura 6.4 – Relatório da execução dos testes instrumentados  
 Fonte: Elaborado pelo autor (2020)

O código-fonte do teste instrumentado é semelhante ao teste local, na questão facilidade de compreensão. Veja que no Código-fonte “Conteúdo do arquivo ExampleUnitTest” usamos o mesmo *annotation Test*. É usado o mesmo método *assertEquals* para conferir a igualdade do pacote instalado no dispositivo físico. O único ponto que requer atenção é a notação *RunWith*, acima do nome da classe. É necessário para que não exista problema ao utilizar o Junit 3 e Junit 4 no teste instrumentado, em um dispositivo físico.

```
@RunWith(AndroidJUnit4::class)
class ExampleInstrumentedTest {
    @Test
    fun useAppContext() {
        // Context of the app under test.
        Val appContext =
        InstrumentationRegistry.getInstrumentation().targetContext
        assertEquals("com.fiap.calculoimc",
        appContext.packageName)
    }
}
```

Código-fonte 6.7 – Conteúdo do arquivo ExampleUnitTest  
 Fonte: Elaborado pelo autor (2020)

## 6.5 Implementando teste local

Vamos então aplicar nossos conceitos em nossa aplicação, que neste momento está com um código extremamente inadequado, em relação aos testes. Se verificar a *MainActivity* e seu código atual, notará que não temos nenhuma separação

de lógica, tudo está sendo feito na mesma classe. Se isso já está parecendo estranho, imagine em projetos grandes e/ou complexos.

Nossa primeira abordagem para melhorar a qualidade do código e 13ndro-lo preparado para os testes será criar um arquivo chamado *Util*. O respectivo código é apresentado no Código-fonte “Conteúdo do arquivo Util.kt”.

No arquivo temos um *enum* de nome *Result*, que recebe por parâmetro um dado String nomeado como *label*. Os valores possíveis desse enum são os níveis resultantes do cálculo de IMC. E como era de se esperar, os *labels* também são correspondentes.

O *data class ImcResult* funciona basicamente como um *wrapper*, um objeto que encapsula a resposta, que será o nosso enum falado no parágrafo anterior. Dessa forma, tornamos o código bem separado e pronto para ser usado de forma adequada nos testes.

Por fim, perceba que na *internal fun* recebemos os parâmetros de peso e altura textuais. O corpo da função é a mesma lógica que ainda temos na *MainActivity*. Porém no final da lógica, apenas passamos como retorno uma instância de *ImcResult* com o seu rótulo correspondente.

```
Internal fun getImcResult(pesoTxt: String, alturaTxt:
String): IMCResult {
    val peso = pesoTxt.toFloat()
    val altura = alturaTxt.toFloat()

    val imc = peso / (altura * altura)

    if (imc < 16){
        return IMCResult(Result.MAGREZA_III)
    } else if (imc < 17){
        return IMCResult(Result.MAGREZA_II)
    } else if (imc < 18.5){
        return IMCResult(Result.MAGREZA_I)
    } else if (imc < 25){
        return IMCResult(Result.OK)
    } else if (imc < 30){
        return IMCResult(Result.SOBREPESO)
    } else if (imc < 35){
        return IMCResult(Result.OBESIDADE_I)
    } else if (imc < 40){
        return IMCResult(Result.OBESIDADE_II)
    } else {
        return IMCResult(Result.OBESIDADE_III)
    }
}
```

```

    }
}

data class IMCResult(val result: Result)

enum class Result(val label: String) {
    MAGREZA_III("Magreza Severa"),
    MAGREZA_II("Magreza moderada"),
    MAGREZA_I("Magreza leve"),
    OK("Saudável"),
    SOBREPESO("Sobrepeso"),
    OBESIDADE_I("Obesidade Grau I"),
    OBESIDADE_II("Obesidade Ghrau II (severa)"),
    OBESIDADE_III("Obesidade Ghrau III (mórbida)")
}

```

Código-fonte 6.8 – Conteúdo do arquivo Util.kt  
 Fonte: Elaborado pelo autor (2020)

Próximo passo é retirar a lógica da *MainActivity* e usar a função criada no arquivo de *Util*. Veja o Código-fonte “Novo conteúdo da MainActivity” e perceba como o código ficou mais sucinto, deixando as responsabilidades das classes mais claras e organizadas. Além disso, configuramos o texto do *txt\_result\_imc* apenas uma única vez possível, com o retorno de rótulo da Util.

```

import androidx.appcompat.app.AppCompatActivity
import android.os.Bundle
import kotlinx.android.synthetic.main.activity_main.*

class MainActivity : AppCompatActivity() {
    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContentView(R.layout.activity_main)

        btn_make_calc.setOnClickListener {
            val pesoTxt = edt_peso.text.toString()
            val alturaTxt = edt_altura.text.toString()

            val imcResult = getImcResult(pesoTxt, alturaTxt)

            txt_result_imc.text = imcResult.result.label
        }
    }
}

```

Código-fonte 6.9 – Novo conteúdo da MainActivity  
 Fonte: Elaborado pelo autor (2020)

Agora que já separamos nossa função, fica fácil criar o teste. E novamente o Android Studio nos auxiliará graficamente nesta tarefa. No arquivo *Util*, na função *getImcResult*, clique com o botão direito do mouse e, em seguida, clique na opção *generate*. Essa ação fará com que um novo diálogo, pequeno, apareça praticamente em cima da função, conforme a Figura “Geração de um novo teste”. Agora clique em *test*.



Figura 6.5 - Geração de um novo teste  
Fonte: Elaborado pelo autor (2020)

Um novo diálogo vai aparecer, desta vez um pouco maior, como apresenta a Figura “Janela de configurações do novo teste”. Tenha o cuidado de selecionar *JUnit4* no *Testing Library*. Em *Class Name*, altere para *UtilTest* e selecione o combo box com o nome do método da classe *Util*.

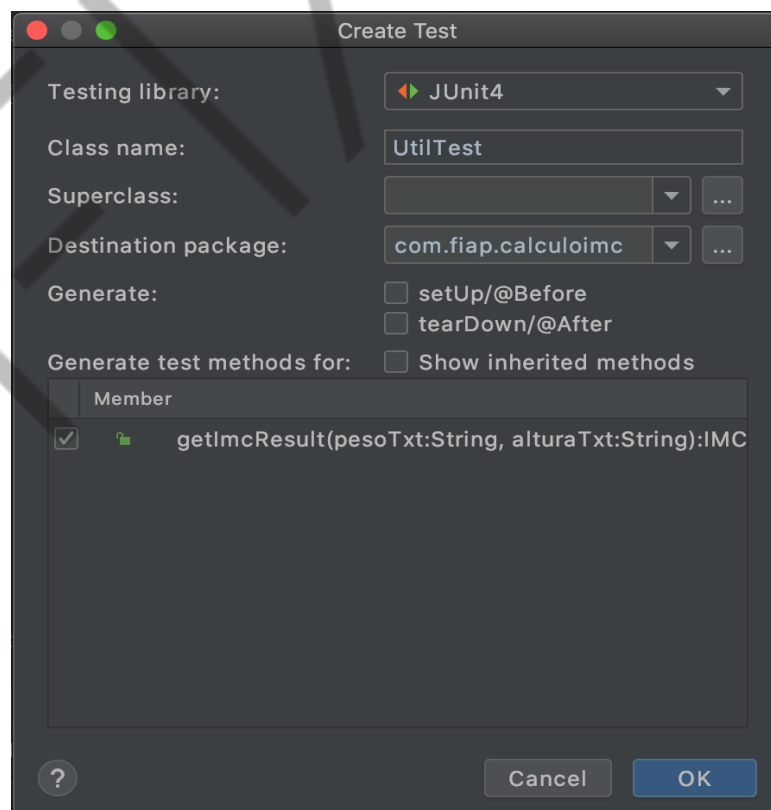


Figura 6.6 - Janela de configurações do novo teste  
Fonte: Elaborado pelo autor (2020)

No código gerado pela IDE, faremos algumas mudanças simples, que podem ser vistas no Código-fonte “Nova classe de teste UtilTest”. O que mudamos então?

- Anotação *Test*.
- Mudamos o nome do método. Seguimos o conceito de *Given-When-Then*. Ou seja, dado o nosso método *getImcresult*, quando o peso é 90 e a altura 1.80, então o resultado deve ser sobrepeso.
- Na lógica da função apenas checamos se o resultado esperado coincide com o resultado encontrado.

```
import org.junit.Assert.*
import org.junit.Test

class UtilTest {

    @Test
    fun testGetImcResult_90_180_sobrepeso() {
        val result = getImcResult("90", "1.80")

        assertEquals(result.result, Result.SOBREPESO)
    }
}
```

Código-fonte 6.10 – Nova classe de teste UtilTest  
Fonte: Elaborado pelo autor (2020)

## 6.6 Aplicando conceitos de Test Driven Development

Vamos aproveitar o avanço nos testes para conhecer um conceito muito difundido no mundo do desenvolvimento e, mais ainda, no mundo dos testes. Estamos falando do *Test Driven Development*.



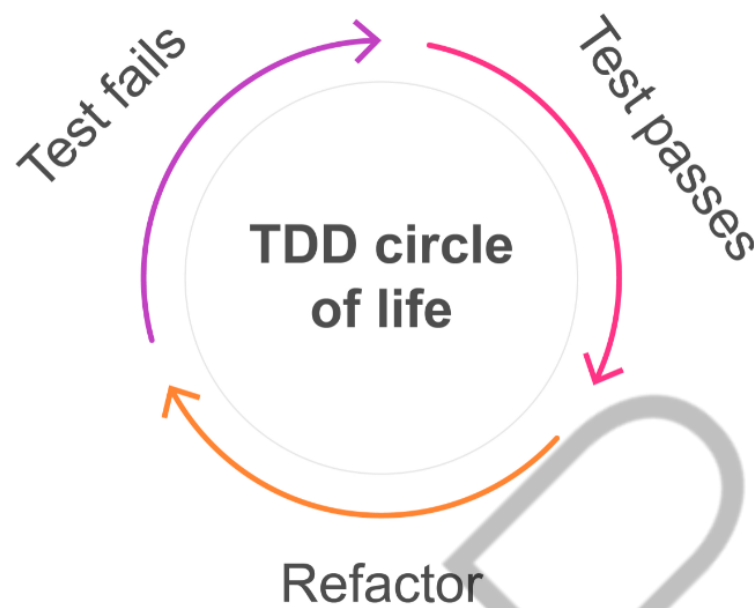


Figura 6.7 - Test Driven Development  
Fonte: Costa (2017)

A ideia central é que devemos criar um teste que deverá falhar. Isso criará a necessidade do código de correção e o consequentemente refatoramento no código. E como isso entra na nossa aplicação? Se o leitor fez alguns testes manuais na nossa aplicação de IMC pode ter se deparado com uma falha: o botão de calcular permite a ação, mesmo quando nenhum valor foi inserido em algum dos campos ou em ambos.

Começaremos com uma pequena alteração no enum *Result* que está escrito no arquivo *Util.kt*. Observe no Código-fonte “Nova constante no Enum Result” que apenas adicionamos mais uma constante ao enum. Ele tem o identificador BLANK e como label tem um aviso que o usuário precisa inserir os valores corretamente.

```
enum class Result(val label: String) {
    MAGREZA_III("Magreza Severa"),
    MAGREZA_II("Magreza moderada"),
    MAGREZA_I("Magreza leve"),
    OK("Saudável"),
    SOBREPESO("Sobrepeso"),
    OBESIDADE_I("Obesidade Grau I"),
    OBESIDADE_II("Obesidade GHrau II (severa)"),
    OBESIDADE_III("Obesidade GHrau III (mórbida)"),
    BLANK("Insira os valores de peso e altura corretamente!")
}
```

Código-fonte 6.11 – Nova constante no Enum Result  
Fonte: Elaborado pelo autor (2020)

Vamos criar nosso teste que falhará. Além da função que já existia e continuará com seu código inalterado, criamos a função *testGetImcResult\_embranco\_aviso*.

Para quando for dado a função *getImcResult* e uma entrada em branco (de peso ou altura) o resultado deverá ser o aviso e não um crash de resultado no fechamento inesperado da aplicação.

Perceba que chamamos a função *getImcResult* em dois momentos, um não passando o peso e outro não passando a altura. Em ambos os casos, estamos comparando os resultados com aquilo que entendemos como correto, que neste caso, é o *Result.BLANK*.

```
class UtilTest {  
  
    @Test  
    fun testGetImcResult_90_180_sobrepeso() {  
        ...  
    }  
  
    @Test  
    fun testGetImcResult_embranco_aviso() {  
        val semPeso = getImcResult("", "1.80")  
        assertEquals(semPeso.result, Result.BLANK)  
  
        val semAltura = getImcResult("90", "")  
        assertEquals(semAltura.result, Result.BLANK)  
    }  
}
```

Código-fonte 6.12 – Novo teste de falha codificado  
Fonte: Elaborado pelo autor (2020)

Ao rodar a classe de teste teremos um sucesso e uma falha, conforme a Figura “Falha em um dos testes”. Neste caso, o erro é o “sucesso”. Isso porque nos obriga a refatorar o teste para conseguir rodar o teste com 2 *passed*. Portanto, ficaremos neste ciclo por todo tempo no desenvolvimento da aplicação.

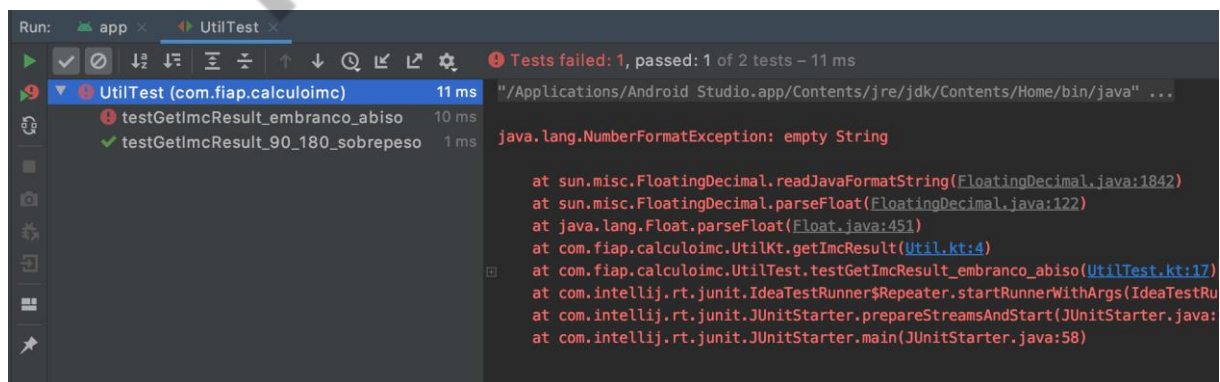


Figura 6.8 - Falha em um dos testes  
Fonte: Elaborado pelo autor (2020)

Vamos voltar à função `getImcResult` dentro do `Util.kt`. Veja no Código-fonte “Correção da função `getImcResult`” como ficou a correção da nossa função. Parte do código foi escondido com os três pontos. Perceba, portanto, que a única alteração foi a inclusão do teste lógico condicional, logo no início da função. Se um dos dois valores forem vazios, o retorno será o `Result.BLANK`. Pronto, agora podemos voltar à classe teste, rodar e receber a aprovação total.

```
class UtilTest {  
    internal fun getImcResult(pesoTxt: String, alturaTxt:  
String): IMCResult {  
        if (pesoTxt.isEmpty() || alturaTxt.isEmpty()){  
            return IMCResult(Result.BLANK)  
        }  
  
        val peso = pesoTxt.toFloat()  
        val altura = alturaTxt.toFloat()  
  
        val imc = peso / (altura * altura)  
  
        ...  
    }  
}
```

Código-fonte 6.13 – Correção da função `getImcResult`  
Fonte: Elaborado pelo autor (2020)

## 6.7 Testes de Interface

Testes de interface também são simplificados pelo poder da IDE e das bibliotecas do Android SDK. Inclusive, é possível seguir o mesmo conceito de desenvolvimento orientado a testes.

Para estudarmos esse conceito, vamos mudar sensivelmente a aplicação. Vamos imaginar que a única tela da aplicação mostrará a mensagem de campo(s) vazio(s) em um texto em negrito e com cor vermelha, para chamar mais atenção do usuário. Se os campos estiverem certos, o valor será apresentado na tela com a mensagem já trabalhada no exemplo anterior.

A primeira mudança será no arquivo de cores, o `colors.xml`. Neste, apenas crie mais um recurso de cor, que pode ser visualizado no Código-fonte “Adição da cor vermelha no `colors.xml`”. O recurso se chama `colorRed` e contém o hexadecimal da cor vermelha.

```
<?xml version="1.0" encoding="utf-8"?>
<resources>
    <color name="colorPrimary">#6200EE</color>
    <color name="colorPrimaryDark">#3700B3</color>
    <color name="colorAccent">#000000</color>
    <color name="colorRed">#ff0000</color>
</resources>
```

Código-fonte 6.14 – Adição da cor vermelha no colors.xml  
Fonte: Elaborado pelo autor (2020)

Em seguida, vamos mudar o final do *activity\_main.xml*. Veja o detalhe dessa mudança no Código-fonte “Mudanças na activity\_main.xml”. O código que não sofreu alterações foi demarcado com três pontos. O penúltimo *TextView* já existia, porém, recebeu um *id* diferente e a definição de *visibility* inicialmente *GONE*. Já o último *TextView* é totalmente novo. Detalhe que ele tem a cor alterada para vermelho e seu estilo definido como negrito.

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout ... >

    ...

    <TextView
        android:textSize="@dimen/label_imc"
        android:visibility="gone"
        android:layout_marginTop="10dp"
        android:id="@+id/txt_result_imc_success"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"/>

    <TextView
        android:visibility="gone"
        android:textColor="@color/colorRed"
        android:textSize="@dimen/label_imc"
        android:textStyle="bold"
        android:layout_marginTop="10dp"
        android:id="@+id/txt_result_imc_error"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"/>

</LinearLayout>
```

Código-fonte 6.15 – Mudanças na activity\_main.xml  
Fonte: Elaborado pelo autor (2020)

Antes de partirmos diretamente para o teste, é importante indicar um detalhe de um erro que causamos na aplicação. Na *MainActivity* ainda estamos usando o *id* antigo da *TextView*, que era *txt\_result\_imc*. Logo, na linha de código que configuramos

a propriedade `text`, faz-se necessário usar o `text_result_imc_success`, pois renomeamos este elemento no XML.

Para a criação da classe de teste de interface repetiremos os passos iniciais feitos nos testes de unidade. Abra a classe `MainActivity`, clique com o botão direito no nome da classe e navegue pelas seguintes opções de menu: *generate -> Test*.

Teremos a mesma janela de diálogo para criação de testes, também já vimos anteriormente. Apenas preste atenção na definição dos seguintes atributos:

- Testing library: JUnit4
- Class name: altere para `MainScreenTest`.

Ao clicar *OK*, precisamos prestar muita atenção na caixa de diálogo apresentada na sequência, conforme a Figura “Escolha do androidTest para testes de UI”. Nesse caso, estamos falando de teste de interface, portanto, são testes instrumentados. Então, diferentemente do que acontece com testes unitários, devemos escolher o *source set* da pasta `androidTest`. Essa configuração é de suma importância. Depois, basta clicar no botão *OK*.

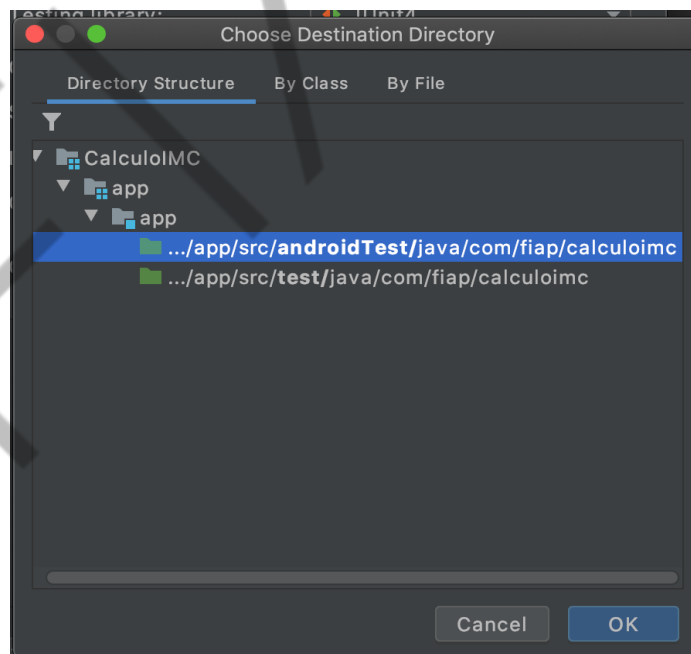


Figura 6.9 - Escolha do androidTest para testes de UI  
Fonte: Elaborado pelo autor (2020)

Antes de começarmos a codificação dos testes, também é necessário que o você modifique seu `build.gradle` ao nível do módulo `app`, acrescentando as dependências necessárias:

- androidTestImplementation 'androidx.test:runner:1.3.0'
- androidTestImplementation 'androidx.test:rules:1.3.0'
- androidTestImplementation 'androidx.test.espresso:espresso-core:3.3.0'

No Código-fonte “Definição da regra na classe de teste” faremos nossas primeiras mudanças. A primeira alteração é o *RunWith*, que já havíamos utilizado nos testes unitários. A grande diferença é a variável criada com nome *activityRule* e o uso da *annotation get:Rule*.

Essa regra vai ser criada antes das funções, com anotação *Test* ou *Before* e serão liberadas da memória depois de todos os testes e métodos marcados com a anotação *After*. Basicamente, estamos dizendo qual a *Activity* estes testes instrumentados usarão. Isso é necessário porque nesse tipo de teste, o Espresso instalará o aplicativo novamente no emulador ou dispositivo físico, lançando a *activity* em primeiro plano.

```
@RunWith(AndroidJUnit4::class)
class MainScreenTest {

    @get:Rule
    var activityRule: ActivityTestRule<MainActivity>
        = ActivityTestRule(MainActivity::class.java)

}
```

Código-fonte 6.16 – Definição da regra na classe de teste  
Fonte: Elaborado pelo autor (2020)

Posteriormente passados, de maneira direta, para o método que será responsável pelo teste. Perceba como fica a classe *MainScreenTest* no Código-fonte “Versão final da MainScreentest”.

Logo de início, percebemos a anotação *Test* que já não é mais uma novidade. Observe a presença de uma lógica que se repete nesse tipo de teste. Primeiro recuperamos um elemento que está presente na interface e depois é feita uma dada ação, como uma edição de texto, um clique, uma rolagem em uma lista, dentre outros. E, por fim, verificamos se o resultado de interface gráfica reflete o desejado, considerado livre de erros.

Para pesquisar um elemento na interface é usado o *Espresso.onView*. Como parâmetro teremos um *Matcher*, portanto, já temos mais um facilitador, que é a

*ViewMatchers* e seus métodos estáticos. No exemplo, estamos usando o *withId*, em que passamos o id desejado na busca, mas a lista de possibilidades é imensa, por exemplo:

- *withClassName*
- *withHint*
- *withParent*
- *withResourceName*
- *withTagKey*
- *withText*

E a lista é extensa, mais do que esses seis exemplos citados. Depois que o *onView* é processado, teremos uma instância de *ViewInteraction*, nele, podemos fazer uma ação ou uma checagem.

No exemplo, no início do método de teste, estamos fazendo uma ação na *ViewInteraction*. Estamos usando o método *perform* e indicando qual ação desejamos simular. Para isso, usamos a *ViewActions* e seus métodos estáticos, como o *click*. Porém, novamente, essa lista de possibilidades é imensa:

- *typeText*
- *longClick*
- *pressBack*
- *pressKey*
- *scrollTo*
- *swipeUp*

Essa lista de possibilidades é bem maior do que os seis exemplos citados acima.

Perceba que no nosso teste clicamos no botão de calcular o imc sem preencher valores nos *EditText*'s, sendo assim, o comportamento desejado é mostrar o *TextView* de erro e não mostrar o de sucesso. E é exatamente isso que estamos fazendo na sequência. Recuperamos o *ViewInteraction* do *TextView* de erro com o *onView* e o seu id correspondente. Depois, usamos o *check* e verificamos a assertividade do teste

com o *ViewAssertions* e seu método estático *matches*. O *ViewMatchers.isDisplayed()* retornará sucesso, se o elemento predecessor estiver visível na interface de usuário.

E, por fim, a lógica é muito parecida com o *TextView* de sucesso. No *matches*, estamos testando se ele tem um efeito de visibilidade aplicado, que é o GONE: (*ViewMatchers.withEffectiveVisibility(ViewMatchers.Visibility.GONE)*).

```
@RunWith(AndroidJUnit4::class)
class MainScreenTest {

    @get:Rule
    var activityRule: ActivityTestRule<MainActivity>
        = ActivityTestRule(MainActivity::class.java)

    @Test
    fun camposEmBranco_textViewCorreta() {
        Espresso
            .onView(
                ViewMatchers.withId(R.id.btn_make_calc)
            )
            .perform(
                ViewActions.click()
            )

        Espresso
            .onView(
                ViewMatchers.withId(R.id.txt_result_imc_error)
            )
            .check(
                ViewAssertions.matches(ViewMatchers.isDisplayed())
            )

        Espresso
            .onView(
                ViewMatchers.withId(R.id.txt_result_imc_success)
            )
            .check(
                ViewAssertions.matches(ViewMatchers.withEffectiveVisibility(
                    ViewMatchers.Visibility.GONE)
                )
            )
    }
}
```

Código-fonte 6.17 – Versão final da MainScreentest  
Fonte: Elaborado pelo autor (2020)



Agora, podemos rodar esse teste. Inicie o emulador ou tenha um dispositivo conectado e configurado para receber o teste. Depois, clique com o botão direito sobre o nome da classe que criamos e busque por *Run MainScreenTest*. Poderá ver a sua aplicação sendo iniciada no dispositivo físico ou virtual e, logo em seguida, fechada também de forma automática.

Porém receberemos um aviso que o teste não passou, como mostra a Figura “Resultado de falha no teste de UI”:

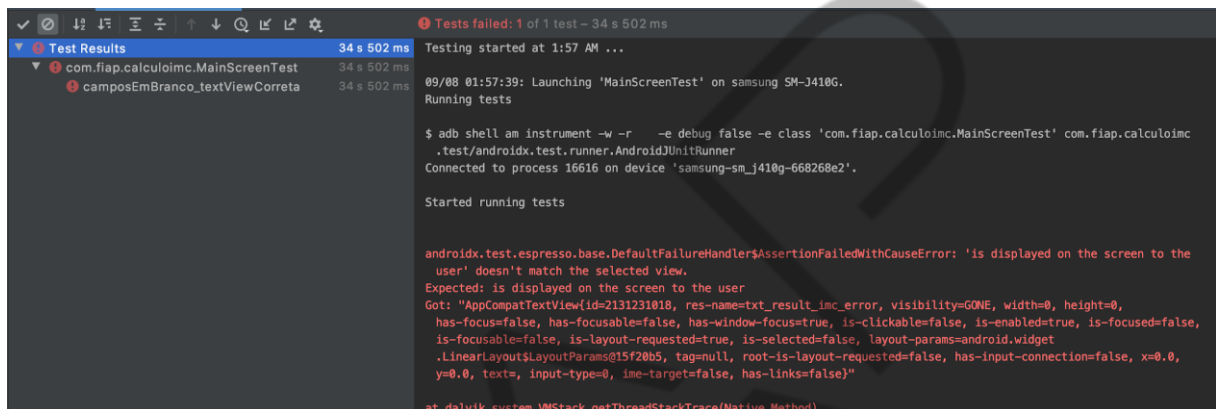


Figura 6.10 - Resultado de falha no teste de UI

Fonte: Elaborado pelo autor (2020)

Isso acontece simplesmente pelo fato de que ainda não fizemos esse tratamento no código. Nossos dois *TextViews* estão com o estado de visibilidade inicial GONE, e não foram alterados. Perceba que na mensagem de erro, isso fica muito claro; veja a seguir:

*androidx.test.espresso.base.DefaultFailureHandler\$AssertionFailedWithCauseError: 'is displayed on the screen to the user' doesn't match the selected view.*

*Expected: is displayed on the screen to the user*

Seguindo a ideia do TDD, vamos refatorar o código da classe *MainActivity* para atender ao nosso teste. Veja o Código-fonte “Versão final da MainActivity” com as alterações na *MainActivity*. Basicamente temos um teste lógico condicional a mais, que habilita ou esconde os *TextViews*, dependendo do resultado da chamada ao método *getImcResult*.

```
class MainActivity : AppCompatActivity() {
    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContentView(R.layout.activity_main)
```

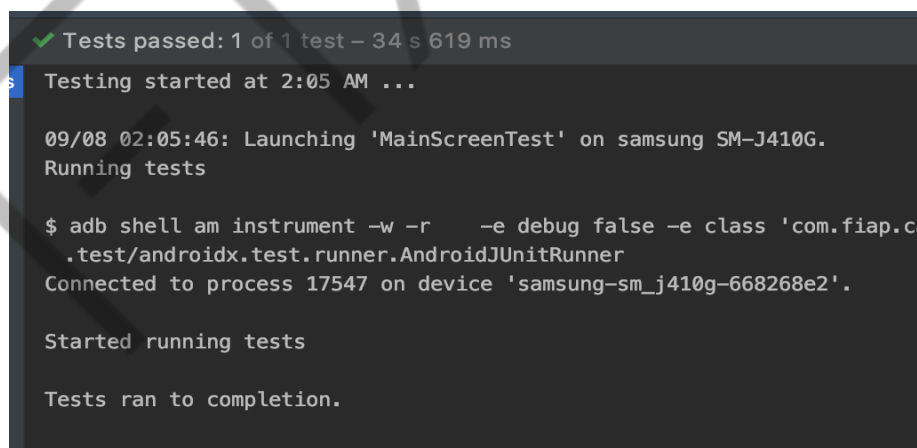
```
btn_make_calc.setOnClickListener {
    val pesoTxt = edt_peso.text.toString()
    val alturaTxt = edt_altura.text.toString()

    val imcResult = getImcResult(pesoTxt, alturaTxt)

    if (imcResult.result == Result.BLANK){
        txt_result_imc_success.visibility = View.GONE
        txt_result_imc_error.visibility = View.VISIBLE
        txt_result_imc_error.text =
imcResult.result.label
    } else {
        txt_result_imc_success.visibility =
View.VISIBLE
        txt_result_imc_error.visibility = View.GONE
        txt_result_imc_success.text =
imcResult.result.label
    }
}
```

Código-fonte 6.18 – Versão final da MainActivity  
Fonte: Elaborado pelo autor (2020)

Depois desta alteração, rode o teste instrumentado novamente. O resultado obterá sucesso no final da execução, como mostra a Figura “Resultado de sucesso no teste de UI”:



```
✓ Tests passed: 1 of 1 test – 34 s 619 ms
Testing started at 2:05 AM ...
09/08 02:05:46: Launching 'MainScreenTest' on samsung SM-J410G.
Running tests
$ adb shell am instrument -w -r -e debug false -e class 'com.fiap.c
.test/androidx.test.runner.AndroidJUnitRunner
Connected to process 17547 on device 'samsung-sm_j410g-668268e2'.
Started running tests
Tests ran to completion.
```

Figura 6.11 - Resultado de sucesso no teste de UI  
Fonte: Elaborado pelo autor (2020)

O projeto que utilizamos neste capítulo, pode ser baixado no repositório <https://github.com/FIAPON/TestesUnidadeUIAndroid>.

## CONCLUSÃO

Desenvolver aplicações sem ter um olhar para testes, não permite que elas cresçam e que os times trabalhem com eficiência. Mesmo que demande um pouco mais tempo, os testes garantem que novas funcionalidades não impactam negativamente as funcionalidades existentes e, com isso, os fluxos de entrega de aplicações conseguem ter mais confiabilidade na entrega das aplicações.

A cultura de testes de TDD e outros conceitos importantes pode nos livrar da maioria dos bugs, diminuir efeitos colaterais em mudanças de código e melhorar a qualidade do nosso código em geral. Afinal, inverter a lógica de pensamento, codificando os testes que buscam as falhas e, depois, o código que se defende destas falhas, eleva o nível do próprio desenvolvedor.

## REFERÊNCIAS

ANDROID DEVELOPER. **Conceitos básicos de testes**. [s.d.]. Disponível em: <https://developer.android.com/training/testing/fundamentals?hl=pt-br>. Acesso em: 18 set. 2020.

ANDROID DEVELOPER. **Criando testes de unidade locais**. [s.d.]. Disponível em: <https://developer.android.com/training/testing/unit-testing>. Acesso em: 18 set. 2020.

ANDROID DEVELOPER. **UI de teste para um único app**. [s.d.]. Disponível em: <https://developer.android.com/training/testing/ui-testing/espresso-testing?hl=pt-br>. Acesso em: 18 set. 2020.

COSTA, H. **O que Test-Driven-Development não é**. 2017. Disponível em: <https://medium.com/creditas-tech/o-que-test-driven-development-n%C3%A3o-%C3%A9-584af2d4c65a>. Acesso em: 18 set. 2020.