

Introdução A Testes de Unidade em Aplicativos Android

Fernando Avanzo

fernando.avanzo@gmail.com

22 de junho de 2020

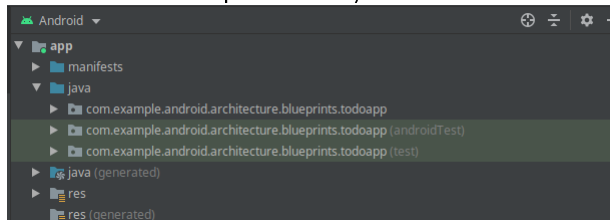
- Fernando Avanzo
 - Engenheiro de Software na Cinq/Dextra
 - Bacharel em Informática pela Universidade Estadual de Maringá
 - Desde 2010 trabalhando com desenvolvimento de sistemas.
- Redes Sociais
 - Github: <https://github.com/FernandoAvanzo>
 - LinkedIn: <https://www.linkedin.com/in/fernandoavanzo/>

1 O que vamos ver hoje

- Como funciona a estrutura de teste no android
- Caixa de ferramentas
- Configuração mínima
- Caso de Uso: ToDo App
- Escrendo testes de unidade

Como funciona a estrutura de teste no android

Os projetos android criados a partir do Android Studio já vem com uma estrutura mínima para a criação de testes automatizados



Os arquivos são organizados da seguinte maneira

- diretório **main** onde fica todo código "funcional" do projeto
- diretório **test** onde fica todo código de teste que é executado localmente
- diretório **androidTest** onde fica todo código de teste que é executado no dispositivo

Ao se escrever testes de unidade automatizados existe uma série de ferramentas úteis que auxiliam o workflow de trabalho. Abaixo vamos ver algumas:

- **Junit** framework xUnit para escrita de testes de unidade
- **Hamcrest** biblioteca que permite a escrita de testes de forma mais idiomática, torna sua leitura mais fácil das pessoas entenderem.
- **AndroidX Test Library** bibliotecas android que auxiliam na simulação de componentes do ciclo de vida da aplicação, (Context, Activitys, Fragments, etc)

Configuração mínima

Abaixo podemos ver qual o aspecto geral de um arquivo *build.gradle* de um projeto orientado a testes.

Example (*build.gradle* default config)

```
defaultConfig {  
    applicationId "com.example.android..."  
    minSdkVersion rootProject.minSdkVersion  
    targetSdkVersion rootProject.targetSdkVersion  
    versionCode 1  
    versionName "1.0"  
    testInstrumentationRunner  
        "androidx.test.runner.AndroidJUnitRunner"  
}
```

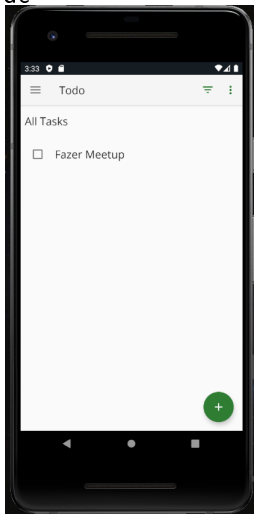
Configuração mínima

Example (*build.gradle* dependencies)

```
// Dependencies for local unit tests
testImplementation "junit:junit:$junitVersion"
// AndroidX Test - Instrumented testing
androidTestImplementation
"androidx.test.ext:junit:$androidXTestExtKotlinRunnerVersion"
androidTestImplementation
"androidx.test.espresso:espresso-core:$espressoVersion"
// AndroidX Test - JVM testing
testImplementation
"androidx.test.ext:junit-ktx:$androidXTestKotlinRunnerVersion"
testImplementation
"androidx.test:core-ktx:$androidXTestCoreVersion"
testImplementation
"org.robolectric:robolectric:$robolectricVersion"
// Other dependencies
testImplementation
"org.hamcrest:hamcrest-all:$hamcrestVersion"
```

Caso de Uso: ToDo App

Este é o aplicativo que vamos usar para estudar alguns casos práticos de teste de unidade



Clássico aplicativo de lista tarefas, que permite que usuário adicione uma nova tarefa na lista, "check" aquelas que já foram concluídas, veja uma estatística simples de quantas atividades ele já concluiu e quantas ainda possui na lista.

Escrevendo nosso primeiro teste

O app **ToDo** tem uma funcionalidade que permite que o usuário tenha uma métrica de quantas tarefas ele terminou e quantas ainda faltam, levando isso em conta, nosso primeiro teste deve verificar que **Dado** uma lista de tarefas, **Quando** a lista tem apenas uma tarefa e ela não está completa, **Então** a porcentagem de tarefas ativas deve ser 100 e a porcentagem de tarefas completas deve ser 0

Escrevendo nosso primeiro teste

Example (Escrevendo o teste)

```
class StatisticsUtilsTest {

    @Test
    fun getActiveAndCompletedStats_noCompleted_returnsHundredZero()
    {
        val tasks = listOf(
            Task("title", "desc", isCompleted = false)
        )

        val result = getActiveAndCompletedStats(tasks)

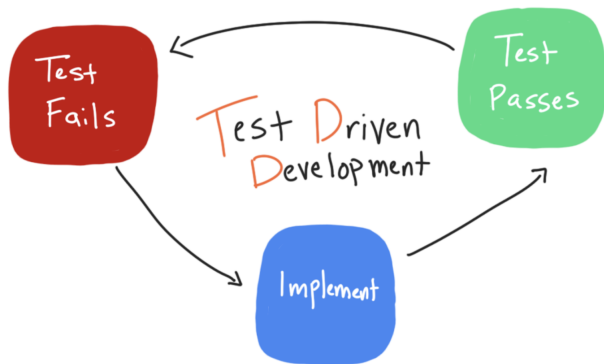
        assertThat(result.activeTasksPercent, 'is'(100f))
        assertThat(result.completedTasksPercent, 'is'(0f))
    }
}
```

Example (Escrevendo o código funcional)

```
internal fun getActiveAndCompletedStats(tasks: List<Task>?)
: StatsResult {
    val totalTasks = tasks!!.size
    val numberOfActiveTasks = tasks.count { it.isActive }
    return StatsResult(
        activeTasksPercent = 100f *
            numberOfActiveTasks / tasks.size,
        completedTasksPercent = 100f *
            (totalTasks - numberOfActiveTasks) / tasks.size)
}
```

O que é TDD?

TDD, (Test Drive Development), é uma prática de desenvolvimento de software que orienta a antes de escrever qualquer código funcional, escrever um código de teste, em seguida escrever o mínimo de código funcional que faça o teste passar. Então trabalhar esse fluxo de forma incremental durante todo o processo de desenvolvimento.



O Jeito de Pensar TDD

Um teste de unidade pode ser escrito em três etapas:

Dado (Given)

Qual é o estado atual? Nesta etapa os dados necessários para simular o estado atual da aplicação é fornecido.

Quando (When)

Quando o estado muda? Nesta etapa o trecho de código (método, função, procedimento, etc..), que muda o estado é executado.

Então (Then)

A resposta(saída), está certa? Nesta etapa é verificado se após o estado inicial ter sido alterado o resultado corresponde ao esperado.

Vamos ver um exemplo prático da abordagem GWT, (Given, When, Then) com o aplicativo ToDo

Cenário de uso: O usuário tem uma lista de tarefas

- **Dado** que usuário possui uma lista de tarefas
- **Quando** a lista de tarefas houver apenas uma tarefa e ela estiver completa.
- **Então** a porcentagem de tarefas ativas deve ser 0 e a porcentagem de tarefas concluídas deve ser 100

Example (Escrevendo o teste)

```
@Test
fun getActiveAndCompletedStats_noActive_returnsZeroHundred() {
    //Given
    val tasks = listOf(Task("title", "desc", isCompleted = true))

    // When
    val result = getActiveAndCompletedStats(tasks)

    // Then
    assertThat(result.activeTasksPercent, 'is'(0f))
    assertThat(result.completedTasksPercent, 'is'(100f))
}
```

Example (Escrevendo o código funcional)

```
internal fun getActiveAndCompletedStats(tasks: List<Task>?)
: StatsResult {

    val totalTasks = tasks!!.size
    val numberOfActiveTasks = tasks.count { it.isActive }
    val activePercent = 100 * numberOfActiveTasks / totalTasks
    val completePercent = 100 *
        (totalTasks - numberOfActiveTasks) / totalTasks

    return StatsResult(
        activeTasksPercent = activePercent.toFloat(),
        completedTasksPercent = completePercent.toFloat())
}
```


Cenário de Bug: Temos uma lista nula ou vazia

- **Dado** que uma lista nula ou vazia é enviada
- **Quando** a lista de tarefas for processada.
- **Então** a porcentagem de tarefas ativas e concluídas deve ser 0

Example (Escrevendo o teste)

```
@Test
fun getActiveAndCompletedStats_error_returnsZeros() {
    //Given
    val tasks = null

    // When
    val result = getActiveAndCompletedStats(null)

    //Then
    assertThat(result.activeTasksPercent, 'is'(0f))
    assertThat(result.completedTasksPercent, 'is'(0f))
}
```

Example (Escrevendo o código funcional)

```
internal fun getActiveAndCompletedStats(tasks: List<Task>?)
: StatsResult {
    return if (tasks == null || tasks.isEmpty()) {
        StatsResult(0f, 0f)
    } else {
        val totalTasks = tasks.size
        val numberOfActiveTasks = tasks.count { it.isActive }
        StatsResult(
            activeTasksPercent = 100f *
                numberOfActiveTasks / tasks.size,
            completedTasksPercent = 100f *
                (totalTasks - numberOfActiveTasks) / tasks.size)
    }
}
```

-  [Code Labs Google \(2020\)](#)
Advanced Android in Kotlin 05.1: Testing Basics

The End