Agora vamos detalhar o **Passo 6**, que envolve a **testes e depuração** do aplicativo de barbearia. Este passo cobre testes de unidade para validar a lógica de negócios, testes de interface usando a biblioteca **Espresso**, e o processo de depuração usando as ferramentas disponíveis no Android Studio.

6. Testes e Depuração

6.1 Testes de Unidade

Os **testes de unidade** garantem que a lógica do aplicativo funcione como esperado. No caso do aplicativo de barbearia, podemos realizar testes para validar funcionalidades como a criação de agendamentos, cálculos de preços ou a autenticação de usuários.

6.1.1 Configurar dependências para testes de unidade no `build.gradle`

No arquivo `build.gradle (Module: app)`, adicione as dependências para testes:

groovy

Copiar código

```
// Testes de unidade testImplementation 'junit:junit:4.13.2' // Mockito para simulações testImplementation 'org.mockito:mockito-core:3.11.2' // Coroutines para testes com Room testImplementation "org.jetbrains.kotlinx:kotlinx-coroutines-test:1.5.0"
```

6.1.2 Teste de Unidade: Validação de Agendamento

Aqui, faremos um teste para verificar se a função de criação de agendamentos está funcionando corretamente. Crie um arquivo de teste para a classe `Agendamento`.

Teste de unidade (`AgendamentoTest.kt`):

kotlin

Copiar código

```
import org.junit.Assert.assertEquals import org.junit.Before import org.junit.Test class AgendamentoTest { private lateinit var agendamento: Agendamento @Before fun setup() { agendamento = Agendamento( id = 1, clienteId = 1, barbeiroId = 2, servicoId = 3, data = "01/12/2023", horario = "14:00" ) } @Test fun testCriacaoAgendamento() { // Verifica se o agendamento foi criado com os dados corretos assertEquals(1, agendamento.id) assertEquals(1, agendamento.clienteId) assertEquals(2, agendamento.barbeiroId)
```

```
assertEquals(3, agendamento.servicoId) assertEquals("01/12/2023", agendamento.data) assertEquals("14:00", agendamento.horario) }
}
```

Explicação:

- `@Before` : Define a configuração inicial antes de cada teste.
- `assertEquals` : Verifica se os valores do agendamento correspondem aos esperados.

6.1.3 Teste de Unidade: Validação de Preço de Serviço

Outro exemplo de teste pode validar se o cálculo de preço de um serviço está correto, considerando possíveis descontos ou taxas.

Teste de unidade (`ServicoTest.kt`):

kotlin

Copiar código

```kotlin
import org.junit.Assert.assertEquals import org.junit.Before import org.junit.Test class ServicoTest { private lateinit var
servico: Servico @Before fun setup() { servico = Servico( id = 1, nome = "Corte de Cabelo", preco = 50.0, descricao = "Corte de
cabelo masculino" ) } @Test fun testCalculoPrecoComDesconto() { val desconto = 10.0 // 10% de desconto val precoComDesconto =
servico.preco - (servico.preco * (desconto / 100)) // Verifica se o preço com desconto foi calculado corretamente
assertEquals(45.0, precoComDesconto, 0.01) } }
```

Explicação:

- Este teste simula um desconto de 10% e valida se o cálculo está correto.

6.2 Testes de Interface com **Espresso**

A biblioteca **Espresso** permite realizar testes automatizados da interface gráfica. Vamos verificar se os elementos da interface estão presentes e se as interações do usuário estão funcionando corretamente.

6.2.1 Configurar dependências do Espresso no `build.gradle`

Adicione as dependências no `build.gradle` para habilitar os testes de interface com Espresso:

groovy

Copiar código

```
// Testes de interface com Espresso androidTestImplementation 'androidx.test.ext:junit:1.1.2' androidTestImplementation
'androidx.test.espresso:espresso-core:3.3.0'
```

6.2.2 Teste de Interface: Tela de Login

Este teste automatizado verifica se os campos de email e senha estão visíveis e se o botão de login funciona corretamente.

Teste de Interface (`LoginActivityTest.kt`):

kotlin

Copiar código

```
import androidx.test.ext.junit.runners.AndroidJUnit4 import androidx.test.rule.ActivityTestRule import
androidx.test.espresso.Espresso.onView import androidx.test.espresso.action.ViewActions.click import
androidx.test.espresso.action.ViewActions.typeText import androidx.test.espresso.matcher.ViewMatchers.withId import
androidx.test.espresso.matcher.ViewMatchers.isDisplayed import androidx.test.espresso.assertion.ViewAssertions.matches import
org.junit.Rule import org.junit.Test import org.junit.runner.RunWith @RunWith(AndroidJUnit4::class) class LoginActivityTest {
@get:Rule var activityRule: ActivityTestRule<LoginActivity> = ActivityTestRule(LoginActivity::class.java) @Test fun
testLoginScreenElements() { // Verifica se os campos de email e senha estão visíveis
onView(withId(R.id.emailEditText)).check(matches(isDisplayed()))
onView(withId(R.id.passwordEditText)).check(matches(isDisplayed())) // Simula o preenchimento dos campos de email e senha
onView(withId(R.id.emailEditText)).perform(typeText("teste@exemplo.com"))
onView(withId(R.id.passwordEditText)).perform(typeText("senha123")) // Simula o clique no botão de login
onView(withId(R.id.loginButton)).perform(click()) } }
```

Explicação:

- `ActivityTestRule` : Inicia a atividade `LoginActivity` para testes.
- `onView(withId())` : Seleciona os elementos da interface pelo ID.
- `perform(typeText())` : Simula a digitação nos campos.
- `perform(click())` : Simula o clique no botão.

6.2.3 Teste de Interface: Navegação entre Telas

Este teste verifica se a navegação entre a tela de login e a tela principal funciona corretamente.

Teste de Interface (`NavigationTest.kt`):

kotlin

Copiar código

```
import androidx.test.espresso.Espresso.onView import androidx.test.espresso.action.ViewActions.click import
androidx.test.espresso.action.ViewActions.typeText import androidx.test.espresso.matcher.ViewMatchers.withId import
androidx.test.espresso.matcher.ViewMatchers.isDisplayed import androidx.test.espresso.assertion.ViewAssertions.matches import
androidx.test.rule.ActivityTestRule import org.junit.Rule import org.junit.Test class NavigationTest { @get:Rule var
activityRule: ActivityTestRule<LoginActivity> = ActivityTestRule(LoginActivity::class.java) @Test fun
testNavigationToMainActivity() { // Simula o login onView(withId(R.id.emailEditText)).perform(typeText("teste@exemplo.com"))
onView(withId(R.id.passwordEditText)).perform(typeText("senha123")) onView(withId(R.id.loginButton)).perform(click()) // Verifica
se a tela principal foi exibida onView(withId(R.id.toolbar)).check(matches(isDisplayed())) } }
```

Explicação:

- Este teste navega da tela de login para a tela principal e valida se o **Toolbar** está visível.

6.3 Depuração usando Android Studio

O **Android Studio** fornece várias ferramentas para depuração, como o **Debugger**, **Logcat**, e **Breakpoints**. Aqui está como você pode utilizá-las:

6.3.1 Usando o Debugger

O debugger permite acompanhar a execução do código em tempo real. Você pode definir **pontos de interrupção (breakpoints)** em linhas específicas do código para inspecionar variáveis e ver como o código está se comportando.

- **Definir um breakpoint**: No editor de código, clique na margem esquerda ao lado de uma linha de código. Um círculo vermelho aparecerá, indicando que um breakpoint foi definido.
- **Executar o app em modo de depuração**: Clique no ícone de depuração (um inseto verde) ou pressione **Shift + F9** para iniciar o aplicativo em modo de depuração.
- **Inspecionar variáveis**: Quando o aplicativo atingir um breakpoint, o Android Studio pausará a execução e você poderá inspecionar o valor das variáveis no painel **Debugger**.

6.3.2 Logcat: Visualizando Mensagens de Log

O **Logcat** exibe mensagens de log em tempo real, permitindo que você veja o que está acontecendo no aplicativo durante a execução.

- Para enviar mensagens ao Logcat, use o método `Log.d` (debug), `Log.i` (informação), `Log.w` (aviso), ou `Log.e` (erro).

Exemplo de uso de `Log` em Kotlin:

kotlin

Copiar código

```kotlin
import android.util.Log class MainActivity : AppCompatActivity() { override fun onCreate(savedInstanceState: Bundle?) {
super.onCreate(savedInstanceState) setContentView(R.layout.activity_main) Log.d("MainActivity", "Aplicativo iniciado com
sucesso!") } }
```

No painel **Logcat**, você pode filtrar as mensagens usando o nome da tag ("MainActivity" neste exemplo).

6.3.3 Usando o Perfilador de Desempenho

O **Android Profiler** ajuda a monitorar o desempenho do aplicativo, como uso de CPU, memória, e rede.

- **Abrir o Profiler**: No Android Studio, clique em **View** > **Tool Windows** > **Profiler**.
- **Monitorar uso de memória e CPU**: O Profiler mostra gráficos em tempo real do uso de recursos do dispositivo enquanto o aplicativo está em execução.
- **Identificar problemas de desempenho**: Você pode identificar gargalos de desempenho, como alto consumo de memória ou processamento.

Conclusão

O **Passo 6** abrange a implementação de **testes de unidade**, **testes de interface** com **Espresso**, e o uso de ferramentas de depuração do **Android Studio** para melhorar a qualidade do código e detectar problemas. Esses testes garantem que a lógica do aplicativo esteja correta e que a interface funcione como esperado. Além disso, a depuração permite resolver erros durante o desenvolvimento, garantindo um aplicativo estável e eficiente.