01

Adicionando o formulário para despesa

Transcrição

Agora que já finalizamos essa nova refatoração, podemos começar com a implementação para permitir que o usuário adicione uma despesa. Então no momento em que o usuário clicar no botão vermelho que é o "Adiciona despesa", terá o mesmo comportamento que o botão "Adiciona receita". A diferença é que agora focaremos na *adição de despesas*!

Vamos começar com a implementação! O primeiro passo é pegar o componente `lista_transacoes_adiciona_despesa`, e em seguida, podemos implementar o *Listener* de Click da mesma forma que fizemos com a Receita:

```
lista_transacoes_adiciona_receita
    .setOnClickListener {
        AdicionaTransacaoDialog(window.decorView as ViewGroup, context: this)
            .configuraDialog(object : TransacaoDelegate {
                override fun delegate(transcricao: Transcricao) {
                    atualizaTransacoes(transacao)
                    lista_transacoes_adiciona_menu.close(animate: true)
                }
            })
    }


lista_transacoes_adiciona_despesa
    .setOnClickListener {

    }
```

Agora, vamos implementar a parte do Lambda. Copiamos o `setOnClickListener` da receita, e colamos na despesa.

```
lista_transacoes_adiciona_despesa
    .setOnClickListener {
        AdicionaTransacaoDialog(window.decorView as ViewGroup, context: this)
            .configuraDialog(object : TransacaoDelegate {
                override fun delegate(transcricao: Transcricao) {
                    atualizaTransacoes(transacao)
                    lista_transacoes_adiciona_menu.close(animate: true)
                }
            })
    }
```

Note que da maneira que está agora, será chamado o *formulário de receita*, sendo que agora, nós queremos chamar o formulário de despesa. Como vamos permitir essa personalização de informações? Uma das técnicas que podemos aplicar é, além de mandar o `delegate`, é mandar alguma informação que identifique o tipo da transação.

Portanto, podemos chegar no componente de *receita,* e dizer para ele fazer uma configuração para o *Dialog*, para o tipo de transação `RECEITA`. Essa é uma das abordagens que podemos fazer, mandando esse tipo como o primeiro parâmetro do `configuraDialog`. E da mesma maneira, faremos na despesa. O código ficará assim:

```
lista_transacoes_adiciona_receita
    .setOnClickListener {
        AdicionaTransacaoDialog(window.decorView as ViewGroup, context: this)
            .configuraDialog(Tipo.RECEITA, object : TransacaoDelegate {
                override fun delegate(transcricao: Transcricao) {
                    atualizaTransacoes(transacao)
                    lista_transacoes_adiciona_menu.close(animate: true)
                }
            })
    }


lista_transacoes_adiciona_despesa
    .setOnClickListener {
        AdicionaTransacaoDialog(window.decorView as ViewGroup, context: this)
            .configuraDialog(Tipo.DESPESA, object : TransacaoDelegate {
                override fun delegate(transcricao: Transcricao) {
                    atualizaTransacoes(transacao)
                    lista_transacoes_adiciona_menu.close(animate: true)
                }
            })
    }
```

Assim, somos capazes de distinguir as chamadas. Em `AdicionaTransacaoDialog`, ele ainda não sabe lidar com essas informações. Portanto, `configuraDialog()` também lidará com o tipo da transação.

```
fun configuraDialog(tipo: Tipo, transacaoDelegate: TransacaoDelegate) {...}
```

Agora, é necessário fazer a **configuração**, a modificação do código, para que ele seja flexível o suficiente para modificar os parâmetros desejados.

Em `configuraCampoData()`, podemos ver que ele cria o componente de data, mas em nenhum momento ele tem alguma informação que reflete no *tipo* da transação, somente no campo da data. Não há nada no corpo dele que diferencie as transações de receitas, das de despesas. Por isso, não vamos mexer nessa função.

Vamos dar uma olhada em `configuraCampoCategoria()`. Repare que aqui, temos uma certa diferença. Quando criamos o `adapter`, estamos utilizando o **array de strings** `categorias_de_receita`. Esse *array* se refere às categorias de receitas. Portanto, podemos utilizar uma das técnicas vistas aqui, que é o **If Expression**.

No lugar desse *array*, usaremos uma variável para definir essas categorias:

```
private fun configuraCampoCategoria() {
    val adapter = ArrayAdapter
        .createFromResource(context,
                categorias,
                android.R.layout.simple_spinner_dropdown_item)

    viewCriada.form_transacao_categoria.adapter = adapter
}
```

Ao invés de simplesmente colocar um valor fixo, vamos dizer que estamos criando uma variável `categorias` que virá baseada em um **if expression**. Esse `if()` será responsável por criar o tipo da transação. Verificamos esse tipo recebendo via parâmetro.

```
private fun configuraCampoCategoria() {

    val categorias = if(tipo ) {

    }

    val adapter = ArrayAdapter
        .createFromResource(context,
                categorias,
                android.R.layout.simple_spinner_dropdown_item)

    viewCriada.form_transacao_categoria.adapter = adapter
}
```

Esse `tipo` é igual ao `Tipo.RECEITA` ? Caso seja, o recurso será `R.array.categorias_de_receita` . Da mesma maneira, se não for, indicaremos que ele é um recurso de array de despesas:

```
val categorias = if(tipo == Tipo.RECEITA) {
    R.array.categorias_de_receita
} else {
    R.array.categorias_de_despesa
}
```

Desta maneira, conseguimos fazer a distinção de *categorias*, tanto de receita quanto de despesa. Dentro do `configuraDialog()` , é necessário enviar esse `tipo` para o `configuraCampoCategoria(tipo)` .

```
fun configuraDialog(tipo: Tipo, transacaoDelegate: TransacaoDelegate) {
    configuraCampoData()
    configuraCampoCategoria(tipo)
    configuraFormulario(transacaoDelegate)
}
```

Agora vamos modificar o `configuraFormulario()` . Podemos reparar logo no início da função que temos um recurso que define o título do nosso formulário. Vamos recortar `R.string.adiciona_receita` e no lugar, colocaremos a variável `titulo` :

```
private fun configuraFormulario(transacaoDelegate: TransacaoDelegate) {
    AlertDialog.Builder(context)
        .setTitle(titulo)
        // resto do código
}
```

Com essa variável `titulo` , podemos fazer um *if expression*. Primeiro, precisaremos do `tipo` como primeiro parâmetro do `configuraFormulario()` , e então faremos a verificação.

```
private fun configuraFormulario(tipo: Tipo, transacaoDelegate: TransacaoDelegate) {

    val titulo = if(tipo == Tipo.RECEITA) {
        R.string.adiciona_receita
    } else {
        R.string.adiciona_despesa
    }

    AlertDialog.Builder(context)
        .setTitle(titulo)
        // resto do código
}
```

Se o `tipo` for igual ao `Tipo.RECEITA`, então devolveremos o recurso `R.string.adiciona_receita`. Se não, devolveremos o recurso `R.string.adiciona_despesa`. Essa é a diferença que terá em nosso formulário.

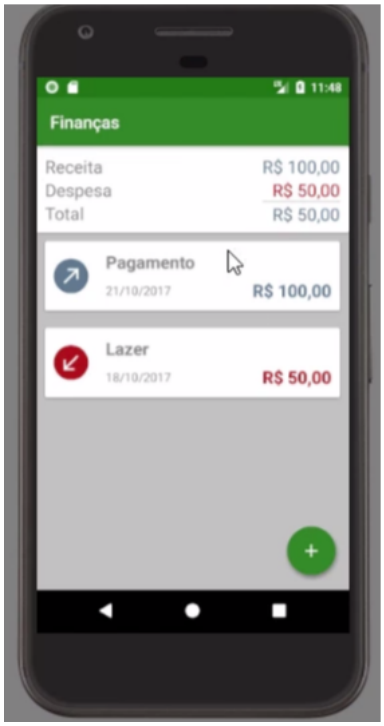
No `AlertDialog.Builder(context)`, temos o `transacaoCriada`. Repare que nós temos um valor fixo para o tipo. E então, por estarmos recebendo `tipo` como parâmetro da `configuraFormulario()`, faremos com que `tipo` seja do mesmo tipo que foi enviando via parâmetro.

```
val transacaoCriada = Transacao(tipo = tipo,
        valor = valor,
        data = data,
        categoria = categoriaEmTexto)
```

Dessa maneira, conseguimos fazer com que essa chamada seja flexível o suficiente para poder chamar tanto uma *receita* quanto uma *despesa*. Antes de testar o código, podemos mandar o `tipo` em `configuraFormulario(tipo, transacaoDelegate)`. Podemos perceber que a `ListaTransacoesActivity` voltou a compilar, pois estamos mandando o `tipo`, e ela já sabe lidar com esses tipos diferentes.

Ao executar o projeto novamente, vamos adicionar uma nova receita, só para ter a certeza de que o código não está quebrado.

Legal! Conseguimos adicionar uma receita. Agora testaremos uma despesa.



Como podemos ver, conseguimos fazer a adição da despesa, e o resumo também está calculando.

Conseguimos fazer com que o usuário consiga adicionar transações, seja de receita, seja de despesa, e de uma maneira muito mais objetiva. Por isso que o momento de fazer a refatoração foi muito importante, justamente para facilitar esse processo de conseguirmos customizar e reutilizar esse código.