03

Refatorando o código

Transcrição

Agora que já colocamos as *features* que permitem o usuário adicionar transações do tipo **receita** ou **despesa**, daremos início ao processo para melhorar o código implementado anteriormente. Então, o foco agora é *melhorar o aspecto visual do código*, para facilitar a compreensão.

Esse processo será iniciado na *activity*. Repare na função `onCreate()`, existe bastante código que precisa ser interpretado. Podemos começar a dar uma atenção especial para os *Listeners* primeiramente.

```
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)
        }
    })
```

Todo esse código que está sendo utilizado, serve para que nós chamemos o *Dialog* que permite adicionar transação. Portanto, podemos extrair uma função através do atalho "Ctrl + Alt + M". Essa nova função será `chamaDialogDeAdicao()`. Mas, assim que nós utilizamos o recurso do Android Studio para nos ajudar, não foi identificado que era também importante para nós definir o parâmetro **tipo**. Então vamos fazer isso manualmente.

```
lista_transacoes_adiciona_receita
    .setOnClickListener {
        chamaDialogDeAdicao()
    }

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

Agora, esse `tipo` terá de ser especificado por quem chamar a função, e vamos recebê-lo via parâmetro. Dentro do *Listener* de `adiciona_receita`, podemos enviar o `Tipo.RECEITA`:

```
lista_transacoes_adiciona_receita
    .setOnClickListener {
        chamaDialogDeAdicao(Tipo.RECEITA)
    }
```

Da mesma maneira, no *Listener* de despesa podemos chamar o método `chamaDialogDeAdicao()` passando o tipo `Tipo.DESPESA`:

```
lista_transacoes_adiciona_receita
    .setOnClickListener {
        chamaDialogDeAdicao(Tipo.RECEITA)
}

lista_transacoes_adiciona_despesa
    .setOnClickListener {
        chamaDialogDeAdicao(Tipo.DESPESA)
}
```

Repare que todo o código dos *Listeners*, é para configurar todo o *action button*, e assim realizar as ações que eles precisam fazer. Portanto, podemos também extrair esse pedaço utilizando o atalho "Ctrl + Alt + M" para uma função chamada de `configuraFab()`.

```
override fun onCreate(savedInstanceState: Bundle?) {
    super.onCreate(savedInstanceState)
    setContentView(R.layout.activity_lista_transacoes)

    configuraResumo()
    configuraLista()
    configuraFab()
}

private fun configuraFab() {
    lista_transacoes_adiciona_receita
        .setOnClickListener {
            chamaDialogDeAdicao(Tipo.RECEITA)
        }

    lista_transacoes_adiciona_despesa
        .setOnClickListener {
            chamaDialogDeAdicao(Tipo.DESPESA)
        }
}
```

Agora, conseguimos fazer com que todas as chamadas grandes ficassem dentro de funções da mesma como fizemos no `configuraResumo()` e no `configuraLista()`.

O `onCreate()` está bem mais fácil de ser compreendido.

Sabemos que a função `chamaDialogDeAdicao()`, obviamente chama um *Dialog*, e podemos melhorar o seu aspecto de leitura. Estamos chamando o `AdicionaTransacaoDialog()`, mas logo em seguida é feita a chamada `configuraDialog()`, sendo que na verdade, ele só está chamando *Dialog*. Então, podemos alterar o nome da chamada de `configuraDialog()` para `chama()`, através do atalho "Shift + F6".

```
private fun chamaDialogDeAdicao(tipo: Tipo) {
    AdicionaTransacaoDialog(window.decorView as ViewGroup, context: this)
        .chama(tipo, object : TransacaoDelegate {
            override fun delegate(transcricao: Transcricao) {
```

```
                    atualizaTransacoes(transacao)
                    lista_transacoes_adiciona_menu.close(animate: true)
                }
            })
    }
```

Já modificamos a nossa *activity*, a deixamos mais simples e mais compreensível, e agora podemos passar para o próximo passo, que é verificar como está o `AdicionaTransacaoDialog`.

Ao acessar essa classe, logo de início podemos perceber que existe um *import* que não está sendo utilizado. Para apagá-lo, podemos utilizar o atalho "Ctrl + Alt + O".

> Sempre que houver imports que não estão mais sendo utilizados, é uma boa prática utilizar esse atalho para que o código sempre tenha somente o que for necessário.

Bom, podemos ver que está tudo separado e bem simples dentro da função `chama()`. Mas, vamos acessar o primeiro método `configuraCampoData()` e dar uma olhada em seu código. Podemos perceber que estamos utilizando várias vezes o componente `form_transacao_data`.

Isso também acontece dentro do método `configuraCampoCategoria()`, onde chamamos várias vezes o mesmo componente `form_transacao_categoria`. Isso significa que, se estamos realizando essas chamadas diversas vezes, podemos **extrair** para uma *property*, pois ela está sendo acessível por todos os membros praticamente.

Vamos selecionar `viewCriada.form_transacao_valor` da variável `val valorEmTexto`. Após ter selecionado somente esse pedaço, utilizaremos o atalho "Ctrt + Alt + F". Será apresentado duas opções para nós: criar a *property* para a classe ou para o arquivo Kotlin. Selecionamos a primeira opção. E então daremos o nome de `campoValor`:

```
    private val campoValor = viewCriada.form_transacao_valor
```

Dessa maneira, estamos fazendo com que todos que estejam utilizando a referência de `viewCriada.form_transacao_valor`, agora utilizarão `campoValor`. Nosso código ficou muito mais fácil de se ler.

Vimos que temos mais pontos a serem refatorados, como o `viewCriada.form_transacao_categoria`. Selecionando esse pedaço de código, e utilizando o atalho "Ctrl + Alt + F", criamos a *property*:

```
    private val campoValor = viewCriada.form_transacao_valor

    private val campoCategoria = viewCriada.form_transacao_categoria
```

Agora só nos resta o `viewCriada.form_transacao_data`. Fazendo todo o processo de refatoração, temos esse resultado:

```
    private val campoValor = viewCriada.form_transacao_valor
    private val campoCategoria = viewCriada.form_transacao_categoria
    private val campoData = viewCriada.form_transacao_data
```

Para melhorar ainda mais a leitura do nosso código, vamos selecionar essas três linhas, e **mover** esse bloco para cima. Utilizamos o atalho "Ctrl + Shift + seta(para cima)", e assim, deixamos todas as *properties* juntas.

```
private val viewCriada = criaLayout()
private val campoValor = viewCriada.form_transacao_valor
private val campoCategoria = viewCriada.form_transacao_categoria
private val campoData = viewCriada.form_transacao_data
```

E o nosso código ficou muito mais fácil de ser visualizado.

Agora, começaremos com outros passos da refatoração. Todas as *properties* que estão utilizando `tipo`, elas utilizam por trás uma *if expression* para distinguir as opções. E como sabemos, também podemos melhorar esse código, e extrair para uma função que irá devolver esses valores para nós.

Selecionaremos essa parte do código:

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

E utilizamos o atalho "Ctrl + Alt + M". Essa função será a `tituloPor()`!

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

    val titulo = tituloPor(tipo)

    //AlertDialog.Builder
}

private fun tituloPor(tipo: Tipo): Int {
    return if (tipo == Tipo.RECEITA) {
        R.string.adiciona_receita
    } else {
        R.string.adiciona_despesa
    }
}
```

Dentro da função, podemos modificar esse *if expression*, pois como sabemos, ele é um recurso exclusivo do Kotlin. Então, podemos deixar dessa maneira:

```
private fun tituloPor(tipo: Tipo): Int {
    if (tipo == Tipo.RECEITA) {
        return R.string.adiciona_receita
    }
    return R.string.adiciona_despesa
}
```

Agora, podemos ver um outro *if expression* que está no `campoCategoria`.

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

```
        } else {
            R.array.categorias_de_despesa
        }
    }
```

Vamos selecionar a partir do **if**, e extrair com o atalho "Ctrl + Alt + M". Chamaremos essa nova função de `categoriasPor()`.

```
    private fun configuraCampoCategoria(tipo: Tipo) {
        val categorias = categoriasPor(tipo)
        // código do adapter
    }

    private fun categoriasPor(tipo: Tipo): Int {
        if (tipo == Tipo.RECEITA) {
            return R.array.categorias_de_receita
        }
        return R.array.categorias_de_despesa
    }
```

Vamos recapitular!

O `configuraCampoData()` não precisa sofrer alterações. O `configuraCampoCategoria()` também não precisa ser refatorado. Já o `configuraFormulario()`, existe um bloco que pode ser refatorado, no `DatePickerDialog`.

Lembre-se que não precisamos mandar a interface que estamos implementando. Portanto, podemos omitir essa informação.

```
    DatePickerDialog(context,
            { _, ano, mes, dia ->
                val dataSelecionada = Calendar.getInstance()
                dataSelecionada.set(ano, mes, dia)
                campoData.setText(dataSelecionada.formataParaBrasileiro())
            }
            , ano, mes, dia)
            .show()
```

Com isso, conseguimos realizar a refatoração do nosso código. Logo depois que refatoramos, é preciso executar para ver se está tudo funcionando.

Agora, conseguimos ler com mais facilidade os pontos que são executados sem ficar com dúvida, sem ficar tendo que interpretar o que significa cada uma das partes. Conseguimos também, entregar para o usuário as funcionalidades para adicionar tanto uma receita quanto uma despesa.