04

Refatorando o Adapter

Transcrição

Continuando o processo de refatoração, vamos acessar o *adapter* e verificar o passo a passo para simplificarmos o código. Inicialmente, deletaremos o comentário que aparece no topo.

Na classe `ListaTransacoesAdapter`, estamos recebendo os parâmetros via construtor primário, atribuindo-os às *properties*. É possível deixá-las no momento em que as recebemos no construtor primário. E no corpo da classe, as *properties* não são necessárias, então as deletaremos também.

```
class ListaTransacoesAdapter(private val transacoes: List<Transacao>, private val context: Conte

    private val limiteDaCategoria = 14

    //...
}
```

Desta forma, deixamos o código bem mais resumido!

A constante `limiteDaCategoria` define o limite da categoria de cada uma das transações em um padrão não muito comum ao Java, em que costumamos usar `LIMITE_DE_CATEGORIA` por exemplo, com todas as letras em maiúsculo e separadas por *underscores*.

No Kotlin, não existe a obrigação de usarmos este padrão, portanto, poderemos deixar a nossa constante - que é uma *property* - da maneira como estava, com *Camel case*. É claro que não está errado deixar como se faz em Java, isto parte de sua preferência pessoal.

Em seguida, a função `getView()` aparece bem extensa. Ela pega a `viewCriada` e, a partir de `transacoes[posicao]` retornamos uma `transacao`. Realizamos diversos processos, inicialmente com o tipo da transação, verificando se é uma receita e atribuindo a cor correspondente.

Se não for, colocamos outra cor, representando uma despesa. Em `transacao_valor` lidamos com o campo referente ao valor. Mais abaixo, fazemos um processo parecido para atribuirmos o ícone correto para cada tipo de transação por meio de `transacao_icone`.

Depois, voltamos a mexer com a transação, com `transacao_valor`, `transacao_categoria` e `transacao_data`. Vamos agrupar todos os códigos de acordo com os campos com que lidam, para organizá-los.

Copiaremos o código abaixo e o colaremos logo após o `if()` e o `else` que se referem a `transacao_valor`:

```
if(transacao.tipo == Tipo.RECEITA){
    viewCriada.transacao_valor
            .setTextColor(ContextCompat.getColor(context, R.color.receita))
} else {
    viewCriada.transacao_valor
            .setTextColor(ContextCompat.getColor(context,R.color.despesa))
}
```

```
viewCriada.transacao_valor.text = transacao.valor
        .formataParaBrasileiro()
```

Feito isto, indicaremos que este bloco de código está adicionando o valor. Selecionando todo o código acima, usaremos o atalho "Ctrl + Alt + M" para extraí-lo em uma função chamada "adicionaValor", deixando-se assim o código encapsulado, indicado por `adicionaValor(transacao, viewCriada)`.

Faremos o mesmo com a parte de adicionar o ícone, criando `adicionaIcone`, e também para a parte de adicionar categoria e outra para a data. Para isso, selecionaremos os respectivos códigos, e com "Ctrl + Alt + M" nomearemos as funções com "adicionaCategoria" e "adicionaData".

Nosso `getView` ficou bem mais claro; pegamos a `viewCriada`, a `transacao`, e adicionamos valor, ícone, categoria e data:

```
val transacao = transacoes[posicao]

adicionaValor(transacao, viewCriada)
adicionaIcone(transacao, viewCriada)
adicionaCategoria(viewCriada, transacao)
adicionaData(viewCriada, transacao)

return viewCriada
```

Ao observarmos `adicionaValor()`, porém, por mais que saibamos que todo o código relativo a ele adiciona valor, como o próprio nome indica, fica difícil entendermos exatamente o que vem a seguir. Isto é, a complexidade do `if()` não é tão clara, portanto é recomendado refatorarmos isso também.

Se analisarmos cada um dos escopos, tanto de `if()` quanto de `else`, passamos pelo campo de valor e uma cor é setada, algo que seria melhor se acontecesse fora do escopo, pois isto ocorrerá independentemente de se tratar de `if()` ou `else`.

Para definirmos uma cor para receita e outra para despesa dentro de `if()`, enviando-as para fora do escopo, é preciso criar uma variável e, quando formos retornar a cor, o faremos com um `Int`, inicializando-a com um número inteiro, `0`.

Ao acessarmos o `if()` referente à receita, faremos com que a cor correspondente seja atribuída. `cor` é uma variável `val`, mas como queremos mudar seu valor, usaremos `var`.

Em `else`, o valor receberá a cor de despesa, bastando então colocar a variável `cor` no momento em que a setamos, fazendo com que nosso *set* de cor não dependa do `if()`. O código ficará assim:

```
private fun adicionaValor(transacao: Transacao, viewCriada: View) {
    var cor = 0
    if (transacao.tipo == Tipo.RECEITA) {
        cor = ContextCompat.getColor(context, R.color.receita)
    } else {
        cor = ContextCompat.getColor(context, R.color.despesa)
    }
    viewCriada.transacao_valor
            .setTextColor(cor)
```

```
    viewCriada.transacao_valor.text = transacao.valor
            .formataParaBrasileiro()
}
```

No entanto, colocamos uma cor com inicialização `0`, sendo que este número não traz significado algum. Será que existe uma maneira mais objetiva de declararmos este tipo de variável?

No Kotlin, há um recurso conhecido por *Expression*, que permite que, ao usarmos `if()`, possamos solicitar que um valor seja devolvido caso isto seja desejado. Por exemplo, pedindo para que seja devolvida uma cor, atribuição que será processada pelo `if()`, que nos devolverá um valor.

Então, quando for do tipo `RECEITA`, queremos que seja devolvida a cor correspondente. E quando este não for o caso, ele automaticamente será do tipo `DESPESA`, ou seja, o próprio `if()` é capaz de nos devolver valores.

Porém, ao fazermos este tipo de atribuição, é preciso tomar cuidado. Neste caso, estamos devolvendo tipos compatíveis. Se devolvermos uma *string*, por exemplo, e mais abaixo um `Int`, o que será que acontecerá? A cor ainda é atribuída, e o `if()` continua sendo compilado, mas isto deixa de acontecer em `setTextColor(cor)`.

Isto porque `cor` está recebendo um valor *n*, um objeto qualquer, então nada garante que seja um `Int`, uma cor que desejamos. Portanto, **quando utilizamos o *if expression*, é preciso deixar explícito o tipo que esperamos.** A partir daí, não importa se é uma *string*, um `Int` ou qualquer outro valor, a devolução de um `Int` será obrigatória em todos os escopos, garantindo-se assim nosso código.

```
val cor: Int = if (transacao.tipo == Tipo.RECEITA) {
    ContextCompat.getColor(context, R.color.receita)
} else {
    ContextCompat.getColor(context, R.color.despesa)
}
```

Mesmo assim, da maneira em que está o código, ainda temos que interpretar o `if()` e, para melhorar isto, faremos o mesmo procedimento do `adicionaValor`, extraindo todo o `if()` para que ele tenha significado.

Ou seja, poderemos selecioná-lo e utilizar o "Ctrl + Alt + M" para indicar que o `if()` está adicionando ou devolvendo uma cor pelo tipo, por meio da função `corPor()`. Por isso, não precisamos mais enviar `transacao`, como o Android Studio fez automaticamente.

Indicaremos que estamos recebendo `tipo`, que será nosso **Enum**. No entanto, por mais que tenhamos simplificado bastante, utilizamos um recurso muito específico do Kotlin, portanto não conseguiremos usá-lo no Java, sendo que é possível obtermos o mesmo comportamento com algo que seja compatível com as duas linguagens.

Às vezes pensamos em fazer implementações de novas funcionalidades, por ser novidade e parecer uma solução muito boa, mas na verdade podemos usar recursos já existentes sem comprometer compatibilidades. Em outras palavras, **em vez de utilizarmos o *if expression*, poderemos optar pelo *early return***!

O código ficará da seguinte forma:

```
private fun adicionaValor(transacao: Transacao, viewCriada: View) {
    val cor: Int = corPor(transacao.tipo)
    viewCriada.transacao_valor
            .setTextColor(cor)
```

```
    viewCriada.transacao_valor.text = transacao.valor
            .formataParaBrasileiro()
}

private fun corPor(tipo: Tipo) : Int {
    if (tipo == Tipo.RECEITA) {
        return ContextCompat.getColor(context, R.color.receita)
    }
    return ContextCompat.getColor(context, R.color.despesa)
}
```

Deste modo, modificamos `adicionaValor()`, deixando mais claro que estamos devolvendo uma cor pelo seu tipo, e estamos setando-a e definindo seu valor depois.

Veremos que `adicionaIcone()` possui praticamente o mesmo comportamento visto em `adicionaValor()`, com o `if()` grande demais. Então, colocaremos a adição fora do escopo, já que ela será acionada de qualquer forma. Precisaremos devolver o ícone de acordo com seu tipo, os quais serão comparados.

Obteremos:

```
private fun adicionaIcone(transacao: Transacao, viewCriada: View) {
    val icone = iconePor(transacao.tipo)
    viewCriada.transacao_icone
            .setBackgroundResource()
}

private fun iconePor(tipo: Tipo) : Int {
    return if (tipo == Tipo.RECEITA) {
        R.drawable.icone_transacao_item_receita
    } else {
        R.drawable.icone_transacao_item_despesa
    }
}
```

No entanto, mais uma vez devolvemos um *if expression*, sabendo que há uma forma mais sucinta de fazê-lo, com *early return*:

```
private fun adicionaIcone(transacao: Transacao, viewCriada: View) {
    val icone = iconePor(transacao.tipo)
    viewCriada.transacao_icone
            .setBackgroundResource(icone)
}

private fun iconePor(tipo: Tipo) : Int {
    if (tipo == Tipo.RECEITA) {
        return R.drawable.icone_transacao_item_receita
    }
    return R.drawable.icone_transacao_item_despesa
}
```

Acessaremos agora `adicionaCategoria` que, como o nome indica, simplesmente adiciona uma categoria, não havendo muito o que ser feito. O mesmo ocorre em `adicionaData`, em relação à data. Já fizemos todas as refatorações de que

precisávamos!

Alcançamos um momento muito interessante da refatoração: a **verificação de seu impacto em nossa aplicação**, isto é, se tudo está funcionando da maneira como estava antes. Estes são os principais focos da refatoração: **tornar o código mais limpo mantendo-se aquilo que funcionava corretamente antes da refatoração**.

Executaremos a app com "Alt + Shift + F10" e veremos que seu aspecto visual permanece o mesmo! Tivemos como ganho um código muito mais fácil de ser compreendido, facilitando também sua manutenção e a implementação de novas funcionalidades.