

SUMÁRIO

1. O QUE É REFATORAÇÃO	1
2. REFATORAÇÃO NO FLUTTER I: EXTRAÇÃO DE CLASSES	1
2.1. CLASSES NO FLUTTER: TODO WIDGET É UMA CLASSE DART	1
2.1.1. OS WIDGETS PRONTOS SÃO CÓDIGOS ENCAPSULADOS	1
2.2. CRIANDO WIDGETS MANUALMENTE: EXTRAINDO CLASSES	2
3. REFATORAÇÃO NO FLUTTER II: IMPORTAÇÃO DOS WIDGETS.....	2

1. O QUE É REFATORAÇÃO

A refatoração é uma forma disciplinada de reestruturar o código para melhorar seu design, e principalmente sua organização e compreensibilidade.

Um aspecto importante de uma refatoração é que ela altera o funcionamento do sistema, e os seus fluxos, mas apenas a sua, digamos, seu design. Ou seja, uma refatoração não adiciona nem remove funcionalidades.

A refatoração permite evoluir o código lentamente no tempo, de forma que se tenha uma abordagem iterativa, reaproveitável e incremental para a implementação. Exemplos de métodos para realizar isso são: renomear métodos, encapsular atributos, especializar métodos, entre outros, como **extrair classes**, que é um dos pilares das **boas práticas de desenvolvimento com Flutter**.

2. REFATORAÇÃO NO FLUTTER I: EXTRAÇÃO DE CLASSES

2.1. CLASSES NO FLUTTER: TODO WIDGET É UMA CLASSE DART

Como já deve ser do seu conhecimento, o aspecto de widgets do Flutter foi inspirado na orientação a componentes do framework React, que utiliza Javascript como linguagem de programação, e uma simulação de HTML programado, o JSX.

Eles são semelhantes. No React um componente é uma função Javascript que retorna uma estrutura visual codificada com JSX e compilada para HTML, e **no Flutter as estruturas visuais são construídas puramente com Dart e retornadas, mas, neste caso, por classes Dart**.

Portanto, os **widgets do flutter são classes Dart** que retornam estruturas visuais quando são instanciadas. A **maioria dessas classes Dart, como sabemos, estão prontas para uso**, e por isso são classificadas como **widgets ready-to-use**.

2.1.1. OS WIDGETS PRONTOS SÃO CÓDIGOS ENCAPSULADOS

Sendo verdade que todo widget é uma classe, os que são disponibilizados para uso no site oficial do Flutter, são, na verdade, códigos muito extensos encapsulados como Classes e que são utilizados a partir de instâncias.

É importante que isso seja entendido, porque a extração manual de classes se baseia no mesmo princípio.

2.2. CRIANDO WIDGETS MANUALMENTE: EXTRAINDO CLASSES

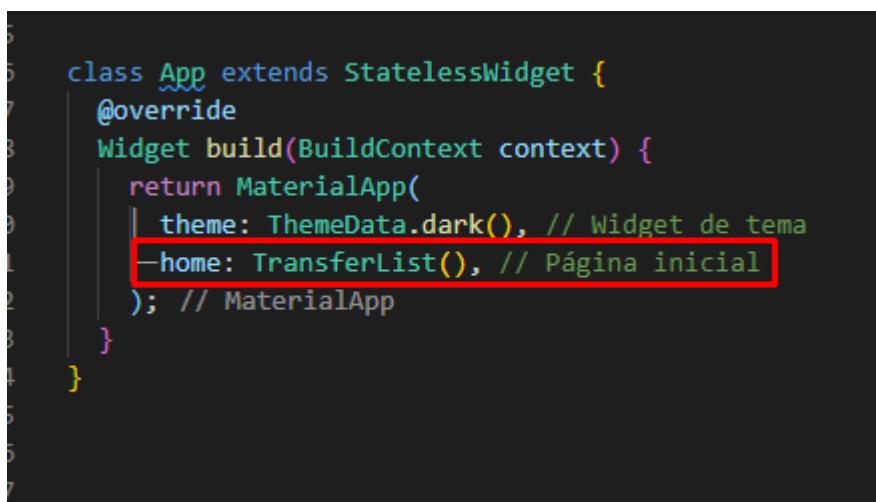
A prática de criar classes durante o desenvolvimento de um aplicativo Flutter segue o mesmo princípio de uso dos widgets que já estão prontos para uso: tornar o código mais enxuto e compreensível a partir do encapsulamento de códigos.

Extrair classes no Flutter, além de uma técnica de refatoração, faz parte da cultura de desenvolvimento da tecnologia. É uma prática obrigatória.

A ideia é simples: ao invés de diversas linhas de código, uma após a outra, partindo do widget root, e formando todo o aplicativo, esse primeiro widget, o root, passa a ter pouca codificação porque o todo é quebrado em classes instanciadas.

Além disso, quando possível dentro do planejamento do sistema, **é importante que as classes tenham um funcionamento genérico e dinâmico para que sejam reaproveitáveis**. Uma classe escrita com definições escritas pelo próprio desenvolvedor até pode ser útil em mais de um contexto, mas uma que é construída com valores variáveis possui uma adaptabilidade muito maior.

Veja o exemplo abaixo. O widget “TransferList” é uma classe criada, instanciada no encaixe “home” do widget “MaterialApp”, e que serve como página. Se não fosse uma classe, ao invés da sua mera instância, todo o código que a define estaria naquele encaixe.



```
class App extends StatelessWidget {  
  @override  
  Widget build(BuildContext context) {  
    return MaterialApp(  
      theme: ThemeData.dark(), // Widget de tema  
      home: TransferList(), // Página inicial  
    ); // MaterialApp  
  }  
}
```

3. REFATORAÇÃO NO FLUTTER II: IMPORTAÇÃO DOS WIDGETS

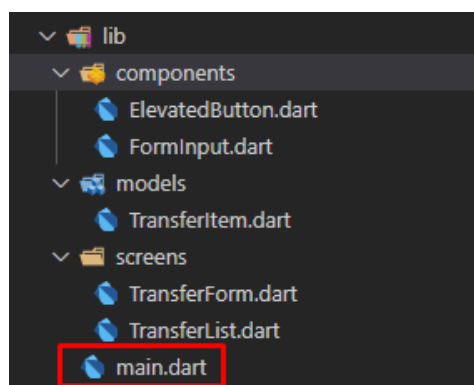
A extração de classes mostrada no tópico anterior, e demonstrada por fim naquela imagem, ainda pode ser refinada alocando as classes extraídas em arquivos separados.

Antes da extração das classes do código mostrado na imagem, e do isolamento delas em arquivos próprios, ele – o código da imagem – possuía cerca de 210 linhas. Até para mim, o criador do código, a visualização dos fluxos era complicada.

Agora, após a refatoração, o arquivo principal, main.dart, que antes possuía essas 210 linhas, possui 14, contando com 2 linhas vazias, e as 2 utilizadas para realizar a importação de arquivos.

```
lib > main.dart > ...
1  import 'package:flutter/material.dart';
2  import 'package:primeiro_projeto/screens/TransferList.dart';
3
4  Run | Debug | Profile
5  void main() => runApp(App());
6
7  class App extends StatelessWidget {
8    @override
9    Widget build(BuildContext context) {
10      return MaterialApp(
11        theme: ThemeData.dark(), // Widget de tema
12        home: TransferList(), // Página inicial
13      ); // MaterialApp
14    }
15  }
```

A estruturação utilizada, neste projeto, das pastas e arquivos pode ser reutilizada em mais projetos, e foi baseada em um [artigo](#) cujo tema era exatamente esse. **Veja a imagem abaixo.**



A **pasta lib** contém o arquivo main.dart e outras 3 pastas: **a pasta screen**, que possui os widgets de páginas/telas do aplicativo; **a pasta models**, que possui classes que retornam objetos de importância geral no aplicativo; e **a pasta components**, que possui widgets/classes que são utilizadas em vários lugares diferentes, no aplicativo.