The Builder Pattern in Dart

dev.to/inakiarroyo/the-builder-pattern-in-dart-efg

Everyone coming from OOP programming have heard about the builder pattern... "and its intent of separating the construction of a complex objets from its representation, providing a flexible solution for creating objects to programmers".

My first steps with OOP was with Java and trust me it was while ago. Nowadays, I pretty much code in Javascript (React & Typescript) and a week ago I decided to give a try Dart...!

Okay buddy, but...! I know, I know "Builder pattern"!

There is a great Dev article introducing cascade notation in Dart, which cover in somehow an approach to the Builder pattern, but in my desire to discover more about this new language this post explain my own implementation of the Builder pattern in Dart.

> My example is based on the 🎯 `class` from the mentioned article, in this way it will help to connect both articles and if you have read it before this one, you will be already familiar with the code
>
> If you prefer to go directly to the code and see what my brain coded, see this gist

First attempts

Thinking on a common implementation of a Builder pattern in *Java* my first attempt was to write a `class` that contained a `static` nested builder class such as:

```
class Pizza {
  Pizza._();

  static class Builder {
    Builder();
  }
}
```

Should be easy... but *"arg!!"* my first error came up: *classes can't be declared inside other classes*

So, researching about this error my surprise was that Dart doesn't allow to declare nested classes. Okay, well... maybe just use the cascade notation and get this sorted out.

NO WAY... I need to find a work around it!
but why? what are the reasons...?

- Immutability issues
  - cascade notation allows to modify/mutate the value of the `class Pizza` attributes ones it has been instantiated/"built" (classes in Dart auto-create *setters* methods for each of their attributes if they are not declared as `final`). So, using it breaks one of the purpose of the builder pattern: build it once and keep it as immutable across its life cycle.
  - Those attributes declared as `final` could not be modified by the cascade notation (there is not *setter* method), so they will need to be initialised into the constructor... *mmm this smells as another reason of why the builder pattern was created, isn't it?*
- Messy code

  there is not separation between those methods used for building the `Pizza` object and those which express what a pizza could "do" or what could be done with a Pizza.

Dart Builder pattern

There are two main code blocks:

- *pizza.dart* which contains the `class Pizza` & `class PizzaBuilder` code
- *test.dart* which contains code explaining how to use them and showing the output of the `print` statements from the *test.dart* code blocks

```dart
/// pizza.dart

class Pizza {
  final String sauce;
  final List<String> toppings;
  final bool hasExtraCheese;

  Pizza._builder(PizzaBuilder builder) :
    sauce = builder.sauce,
    toppings = builder.toppings,
    hasExtraCheese = builder.hasExtraCheese;
}

class PizzaBuilder {
  static const String neededTopping = 'cheese';

  final String sauce;

  PizzaBuilder(this.sauce);

  List<String> toppings;
  bool hasExtraCheese;

  void setToppings(List<String> toppings) {
    if (!toppings.contains(neededTopping)) {
      throw 'Really, without $neededTopping? :(';
    }

    this.toppings = toppings;
  }

  Pizza build() {
    return Pizza._builder(this);
  }
}
```

`class Pizza`

Ups:

- Clean code and concept separation
- `class Pizza` declares their attributes as `final`, so they can't be modified ones they have been instantiated or built
- `class Pizza` redefines its default constructor, declaring a named constructor `_builder`, which warns devs that this class must be built through the Builder pattern. In addiction, it is marked as private (library scoped) with the `_` and it does not allow to create direct instances of itself as there is not default constructor defined
- Injects the `PizzaBuilder` as attribute of the `Pizza` constructor initialing the `final` attributes invoking a superclass constructor using the initializer-list mode

- Because attributes are declared as `final` they don't need to be private, so we could access them directly ones the `Pizza` object is built with no extra *getters* code inside the class

`class PizzaBuilder`

Ups:

- Clean code and concept separation
- `PizzaBuilder` only knows about how to build a `Pizza`, nothing else
- It allows to build a `Pizza` object using specific methods as `setToppings` (for a fine customisation) or directly accessing the attributes by the cascade notation mode without the verbose *setters & getters* code
- Having it as external class allows to reuse a `PizzaBuilder` instance for building more than one `Pizza` object
- It combines the Builder pattern standards with the fast and flexible Dart cascade notation technique

```
/// test.dart
print('___PIZZA BBQ___');

Pizza pizza = (
  PizzaBuilder('bbq')
    ..setToppings(['tomato', 'cheese', 'chicken'])
    ..hasExtraCheese = true
  ).build();

print(pizza.sauce);          // bbq
print(pizza.toppings);       // [onion, cheese, chicken]
print(pizza.hasExtraCheese);  // true
```

Pizza BBQ was ordered with a nice bbq sauce, great toppings and an amazing extra of cheese.

The `PizzaBuilder` uses the cascade notation pattern to build the `Pizza`:

- uses the customise `setToppings` method which verify the needed toppings are part of the included ones
- set the `hasExtraCheese` accessing directly the attribute, because there is no need to create a verbose *setter* method

Fast, easy and flexible 🍕!

```
print('___PIZZA Carbonara___');

Pizza pizza2 = (
  PizzaBuilder('cream')
  ..hasExtraCheese = true
).build();

print(pizza2.sauce);          // cream
print(pizza2.toppings);       // null
print(pizza2.hasExtraCheese); // true
```

This Pizza Carbonara is a bit weird, don't you think it?... the employee forgot to add toppings to it!

...but for the coding world everything is good, nothing breaks, you can build your Pizza as you want!

```
print('___PIZZA Margherita___');

Pizza pizza3 = (
  PizzaBuilder('olive-oil')
  ..setToppings(['PINEAPPLE', 'tomato'])
  ..hasExtraCheese = false
).build();
                          // Uncaught Error: Really, without cheese? :(

print(pizza3.sauce);          // No output due to Uncaught Error
print(pizza3.toppings);       // No output due to Uncaught Error
print(pizza3.hasExtraCheese); // No output due to Uncaught Error
```

Hell yeah, the amazing and famous Neapolitan pizza Margherita has been ordered! Melted Mozzarella cheese... wait! what? Oh, God! the employee has added PINEAPPLE as topping instead of cheese!

...but building the Pizza by cascade notation pattern using a custom `setToppings` method allowed the restaurant to detect that the employee made a mistake showing `Really, without cheese? :(` on the system

Conclusion

The employee always said he was from Naples and the real Pizza was made with pineapple... :)

As I tried to explained above, combining different techniques and patterns such as Builder one along with the OOP standards and the cascade notation from Dart, we have been able to build and easy, flexible and powerful Dart Builder pattern.

...pineapple out! cheese forever!

Posted on Feb 28 by: