Kotlin - builder pattern

Donnerstag, 22. August 2024 17:03

Author: Damir Ljubic
e-mail: damirlj@yahoo.com

Introduction

Builder pattern belongs to the ***creational patterns***, and it's used for constructing the complex objects - complex in a way that consist usually from a lot of properties, which makes their initialization during the enclosing object instantiation kind of complex - since those properties are optional.

Instead of having the overloaded constructors set - with all possible combinations, the builder pattern offers the construction mechanism that enables the *configurable*, step-by-step construction of the user-defined type: giving the possibility to have multiple representation of the same type (separating creation of the type from its representation)

Implementation

Different languages employ different mechanisms (idioms), in order to implement the very same design patterns - as code best practices meant to solve a certain type of the problems.
In early days of C++, the design patterns are mostly implemented using dynamic polymorphism and inheritance.
Nowadays, the most patterns are reimplemented using mostly static polymorphism and advanced generic programming technics introduced with modern C++ standards.

<More on that>: https://github.com/damirlj/modern_cpp_tutorials/blob/main/docs/desing%20patterns/Builder.pdf

In Kotlin, the basic implementation is the classical one - pretty much the same as with Java.
We have the outer class for which we defined the static nested class - a builder, which will be used to configure the outer class - mirroring its properties, as nullable (optional) types.
The outer class (primary) constructor will be private - disabling the direct creation

```
class A private constructor(
    val id :Int? = null,
    val name : String? = null
) {

   companion object {// static block
       class Builder() {

           private var id : Int? = null
           private var name: String? = null

           // Setters

           fun setId(id: Int) = apply {this.id = id }

           fun setName(name: String) = apply {this.name = name }

           fun build() = A(id, name)
       }

   }


   override fun toString():String {
       return "[id=$id, name=$name]"
}
```

Where *apply* is one of the *scope functions* that references inside the function/lambda scope (as argument of apply) the object on which apply is invoked - the receiver, with "this", and returns at the end the very same reference of modified receiver.
This is just matter of convenience - for writing the setters: that set the property and return the reference to the modified object (Builder).

How about when we need **"dynamic" builder** - when we receive dynamically updates that alter only the subset of properties (or even a single one) that we need to inspect and aggregate?

For that, we need to update our design, by adding

- Additional constructor that takes the enclosing class instance as current aggregated value (copy-constructor)
- Additional setter that similarly takes and parses the dynamic updates - as instance of the enclosing class

In this simplified example, that would mean

```
// For aggregated updates - initialize with the current one (copy function)
  constructor(a: A) :this() {
     id = a.id
```

```
    name = a.name
  }

 /**
  * For aggregated updates:
  * - create the builder with secondary c-tor, that takes the current value of outer class instance
  * - update the aggregated value with the non-null delta-update
  * This will be our new aggregated value
  *
  * @param a The outer class instance, as currently aggregated value
  */
fun update(a: A) = apply {
  a.id?.let { id = it }
  a.name?.let { name = it }
}
```

As matter of convenience, we can add helper method within the static block (*companion objec*t)

```
fun update(curr :A, new: A) = Builder(curr).update(new).build()
```

@note Unlike C++ and Java, the "new" is not the reserved word (operator)

A better way

One of the obvious drawback with the classical implementation of a Builder is crowded interface with tedious to write setters - for all containing properties.
We can redesign the implementation in more Kotlin-idiomatic way, using this concepts:

- Builder as inner class with public mutable properties (var)
- Using the placeholder for anonymous (lambda) Extension Function on the Builder - to specify the properties on the fly, as part of the function block that will be consumed by the *apply*, altering the receiver object (the Builder itself)
  This will replace all individual setters at once

```
class CarConfiguration private constructor(
    // mandatory
    val id: Int,
    val brand: String,
    val engine: Engine,
    // optionals
    val camera: Camera?,
    val smartphone: Smartphone?,
    val adas: ADAS?
) {

      companion object {
          class Builder (
              // mandatory
              var id: Int,
              var brand: String,
              var engine: Engine ) {


              // Optionals
              var camera: Camera? = null
              var smartphone: Smartphone? = null
              var adas: ADAS ? = null

              fun build() = CarConfiguration(id, brand, engine, camera, smartphone, adas)

          /**
          * Building the outer class instance
          *
          * @param block This is lambda that extends Builder on fly - Builder is receiver,
          * which means the callable can access any non-private (public) properties of the
          * Builder (all of them), in order to properly configure the outer class
          */
          inline fun build(id: Int,
                           brand: String,
                           engine: Engine,
                           block: Builder.() -> Unit ) = Builder(id, brand, engine).apply(block).build()
        }

    }
}
```

At the client side, we can configure the outer class instance now as

```
val car = CarConfiguration.build(123, "Audi", Engine.DIESEL) {
    camera = Camera.FRONT
    smartphone = Smartphone.CarPlay
```

```
}
```

@note As in this example, we may have a mandatory parameters - not only the optional ones.

But this whole approach now, with Builder, where we try to get rid of the setters, reveals obvious - we don't need the Builder inner class at all, since it essentially only mirrors properties of the outer class.
Is that correct?

The Kotlin, with named parameters, default values and *apply* scope function eliminates need for the nested Builder class.
We can simplify the entire implementation having a single helper method: build, within the static block of the composing class

```
class CarConfiguration(
    // mandatory
    var id: Int,
    var brand: String,
    var engine: Engine,
    // optionals
    var camera: Camera? = null,
    var smartphone: Smartphone? = null,
    var adas: ADAS? = null
) {


        companion object {
         /**
         * Helper method, for building the outer class, as wrapper around the apply()
         * Instead of using the apply() directly - this (naming) gives a better sense of purpose
         *
         * @param block This is lambda that extends the receiver - instance of the outer class,
         * which means the callable can access any non-private (public) properties of the
         * receiver (all of them), in order to proper configure it
         */
             inline fun build(
                 id: Int,
                 brand: String,
                 engine: Engine,
                 block: CarConfiguration.() -> Unit
                ) = CarConfiguration(id, brand ,engine).apply(block)

        }
}
```

Having public constructor brakes yet the **encapsulation** (unfortunately, there is no *friend* concept in Kotlin as with C++) and gives the client absolute freedom to construct the object either at once - through the constructor, or to combine with step-by-step approach with helper method build().
It can access and modify each property afterwards.
If this is not desirable behavior (**Builder antipattern**) - one can fall back on one of previous variants.

At the client side, we can now construct the type as

```
var car = CarConfiguration2(123,"Audi",Engine.DIESEL)
println(car)
// Somewhere in the code, as conditional setup
car = CarConfiguration2.build(car) {
   camera = Camera.REAR
   adas = ADAS.Copilot
}
println(car)
```

Our "dynamic builder" - for handling the dynamic updates of a given type at subscriber side (Publisher-Subscriber scenario),
for having aggregated updates, we can, fallowing this antipattern path, rewrite as

```
class CarConfiguration(
    // mandatory
    var id: Int,
    var brand: String,
    var engine: Engine,
    // optionals
    var camera: Camera? = null,
    var smartphone: Smartphone? = null,
    var adas: ADAS? = null
) {

      // For aggregated updates - initialize with previous configuration (copy-constructor )
      constructor(car: CarConfiguration) :this(car.id, car.brand, car.engine) {
          camera = car.camera
          smartphone = car.smartphone
          adas = car.adas
     }
```

```
    // For aggregated updates - the new (partial) configuration that will be applied on
    // top of the current configuration
    fun update(car: CarConfiguration) = apply {
        id = car.id
        brand = car.brand
        engine = car.engine

        car.camera?.let {camera = it }
        car.smartphone?.let {smartphone = it }
        car.adas?.let {adas = it }
    }


    companion object {

        inline fun build(
            id: Int,
            brand: String,
            engine: Engine,
            block: CarConfiguration.() -> Unit) =
                   CarConfiguration(id, brand, engine).apply(block)

        // For aggregated updates
        inline fun build(
            curr: CarConfiguration,
            block: CarConfiguration.() ->Unit) =
                   CarConfiguration(curr).apply(block)

        fun build(curr: CarConfiguration, new: CarConfiguration) =
                   CarConfiguration(curr).update(new)
    }
}
```

since for update object this approach actually has a sense.

The entire code can be found at:
https://github.com/damirlj/modern_cpp_tutorials/tree/main/src/Kotlin/src/main/java/com/example/practice_kotlin/design_patterns/builder