Atelier 1 – Patron de conception Builder / Monteur

Dans les différentes classes du SDK Spotify, le patron de conception Builder est souvent utilisé, on y retrouve donc des lignes de codes étranges comme celle-ci :

ConnectionParams connectionParams =  
 **new** ConnectionParams.Builder(***CLIENT\_ID***)  
 .setRedirectUri(***REDIRECT\_URI***)  
 .showAuthView(**true**)  
 .build();

Ce genre de code s’appuie sur le patron de conception Builder, de plus en plus utilisé. On l’a d’ailleurs déjà utilisé en créant des AlertDialog ( AlertDialog.Builder )A quoi sert-il ?

Exemple :

Je veux faire une classe permettant de représenter une Personne dans un programme de généalogie. Voici un exemple de la classe Personne :

public class Personne {

**private** String **prenom**;  
**private** String **deuxiemePrenom**;  
**private** String **nom**;  
**private int age**;  
**private** String **nomPere**;  
**private** String **nomMere**;  
**private String lieuNaissance**;  
**private Calendar dateNaissance**;

// constructeur

public Personne ( bien trop de paramètres à passer !! )

{

this.prenom = prenom;

this.deuxiemePrenom = deuxièmePrenom;

//… je suis tanné …

* Devient difficile à maintenir, difficile de se rappeler l’ordre des paramètres, entraîne erreurs

Le pattern Builder est utilisé pour la création d’objets complexes dont les différentes parties doivent être créées suivant un certain ordre ou algorithme spécifique. Une classe externe contrôle l’algorithme de construction

On ajouterait donc une classe statique interne à l’intérieur de notre classe Personne :

// suite de la classe Personne

**public static class** Builder {  
**private** String **prénom**;  
**private** String **deuxiemePrenom**;  
**private** String **nom**;  
**private int age**;  
**private** String **nomPere**;  
**private** String **nomMere**;  
**private String lieuNaissance**;  
**private Calendar dateNaissance**;

**public** Builder(){}

**public** Builder setPrenom(String prenom) {  
 **this**.**prenom** = prenom;  
 **return this**;  
 }  
  
 **public** Builder setDeuxiemePrenom(String deuxiemePrenom) {  
 **this**.**deuxiemePrenom** = deuxiemePrenom;  
 **return this**;  
 }  
  
 **public** Builder setNom(String nom) {  
 **this**.**nom** = nom;  
 **return this**;  
 }  
  
 **...**  
  
 **public** Personne build() {  
 **return new** Personne(**prenom**, **deuxiemePrenom**, **nom**, **age**, **nomPere**, **nomMere**, **lieuNaissance**, **dateNaissance**);  
 }  
 }

On peut donc à présent créer des objets de cette manière :

Personne personne = **new** Personne.Builder()  
.setAge(5)  
.setPrenom(**"Bob"**)  
.setNomMere(“Lucille Drapeau”)  
.setNom(“Charron”)  
.build();

# Avantages

1) Le code est plus gérable si le nombre de variable d’instance est plus grand que 4 ou 5..

2) La création des objets est moins sujet à erreur car l’usager appelle explicitement des méthodes

3) ce patron est robuste car on est certain que l’objet est réellement construit.

4) Pas besoin de passer des null dans des paramètres où on ne peut pas initialiser la variable tout de suite dans le constructeur

# Désavantages

On doit réécrire les variables d’instance dans la classe Builder également ( répétition de code )

Exercices ( créer des objets avec le patron Builder expliqué ci-haut )

1. Créer un objet Personne qui s’appelle Marie Antoinette Duchêne, 94 ans, née de parents inconnus le 2 janvier 1927 à St Jean.

Personne personne = **new** Personne.Builder()  
.setAge(94)  
.setPrenom(**"**Marie**"**)  
.setL**ieuNaissance(«St jean )**;  
**.setDateNaissance(«2 janvier 1927 »)**;   
.setNom(“Antoinette ”)  
.build();

1. Créer un objet Personne qui s’appelle Martin Lemieux, 30 ans, fils de Lucie Veillette et de Marcel Lemieux, né une date inconnue et on n’a pas d’informations sur son lieu de naissance.

Personne personne = **new** Personne.Builder()  
.setAge(50)  
.setPrenom(**"**Martin **"**)

.setNomMere(“Lucie Veillette ”)

.setNomPere(« Marcel Lemieux »);  
.setNom(“Lemieux ”)  
.build();