Builder Pattern using Fluent Interface

Below are some reasons arguing for the use of the pattern and example code in Java, but it is an implementation of the Builder Pattern covered by the Gang of Four in *Design Patterns*. The reasons you would use it in Java are also applicable to other programming languages as well.

As Joshua Bloch states in [Effective Java, 2nd Edition](http://www.amazon.co.uk/Effective-Java-Second-Joshua-Bloch/dp/0321356683):

The builder pattern is a good choice when designing classes whose constructors or static factories would have more than a handful of parameters.

We've all at some point encountered a class with a list of constructors where each addition adds a new option parameter:

Pizza(int size) { ... }

Pizza(int size, boolean cheese) { ... }

Pizza(int size, boolean cheese, boolean pepperoni) { ... }

Pizza(int size, boolean cheese, boolean pepperoni, boolean bacon) { ... }

**This is called the Telescoping Constructor Pattern.** The problem with this pattern is that once constructors are 4 or 5 parameters long it becomes **difficult to remember** the required **order of the parameters** as well as what particular constructor you might want in a given situation.

One **alternative** you have to the Telescoping Constructor Pattern is the **JavaBean Pattern** where you call a constructor with the mandatory parameters and then call any optional setters after:

Pizza pizza = new Pizza(12);

pizza.setCheese(true);

pizza.setPepperoni(true);

pizza.setBacon(true);

**The problem here is that because the object is created over several calls it may be in an inconsistent state partway through its construction.** This also requires a lot of extra effort to ensure thread safety.

**The better alternative is to use the Builder Pattern.**

**This results in code that is easy to write and very easy to read and understand.** In this example, the **build method could be modified** to check parameters after they have been copied from the builder to the Pizza object and **throw an IllegalStateException if an invalid parameter value has been supplied.** This pattern is flexible and it is easy to add more parameters to it in the future. It is really only useful if you are going to have more than 4 or 5 parameters for a constructor. That said, it might be worthwhile in the first place **if you suspect you may be adding more parameters in the future.**

**public** **class** Test {

**public** **static** **void** main(String[] args) {

Pizza pizza = **new** Pizza.Builder(12)

.cheese(**true**)

.pepperoni(**true**)

.bacon(**true**)

.build();

}

}

**public** **class** Pizza {

**private** **int** size;

**private** **boolean** cheese;

**private** **boolean** pepperoni;

**private** **boolean** bacon;

**public** **static** **class** Builder {

// required

**private** **final** **int** size;

// optional

**private** **boolean** cheese = **false**;

**private** **boolean** pepperoni = **false**;

**private** **boolean** bacon = **false**;

**public** Builder(**int** size) {

**this**.size = size;

}

**public** Builder cheese(**boolean** value) {

cheese = value;

**return** **this**;

}

**public** Builder pepperoni(**boolean** value) {

pepperoni = value;

**return** **this**;

}

**public** Builder bacon(**boolean** value) {

bacon = value;

**return** **this**;

}

**public** Pizza build() {

**return** **new** Pizza(**this**);

}

}

**private** Pizza(Builder builder) {

size = builder.size;

cheese = builder.cheese;

pepperoni = builder.pepperoni;

bacon = builder.bacon;

}

}

In case of Jersey, the Respponse is built like this.

Response.ok(200).entity(“Something”).build();

In case of Hibernate, the query can be formed like,

HQLQuery query = new HibernateQuery(session);

List<Stock> s = query.from(stock, bonus)

.where(stock.someValue.eq(bonus.id))

.list(stock);

One more example is given below.

**package** com.ddlab.rnd.type2;

**public** **class** Car {

**private** **int** quantity;

// Optional Parameters

**private** **boolean** isAVS;

**private** **boolean** isAutomaticGear;

**private** **boolean** isBulletProof;

**public** **static** **class** CarBuilder {

**private** **int** quantity;

**private** **boolean** isAVS;

**private** **boolean** isAutomaticGear;

**private** **boolean** isBulletProof;

**public** CarBuilder(**int** quantity) {

**this**.quantity = quantity;

}

**public** CarBuilder avsSystem(**boolean** flag) {

isAVS = flag;

**return** **this**;

}

**public** CarBuilder automaticGear(**boolean** flag) {

isAutomaticGear = flag;

**return** **this**;

}

**public** CarBuilder bulletProof(**boolean** flag) {

isBulletProof = flag;

**return** **this**;

}

**public** Car build() {

**return** **new** Car(**this**);

}

}

**private** Car(CarBuilder builder) {

isAVS = builder.isAVS;

isAutomaticGear = builder.isAutomaticGear;

isBulletProof = builder.isBulletProof;

}

}

**public** **class** TestCar {

**public** **static** **void** main(String[] args) {

Car car = **new** Car.CarBuilder(1).automaticGear(**true**).bulletProof(**true**).build();

System.***out***.println(car);

}

}