HowToDoInJava

Builder Design Pattern

Last Updated: January 3, 2022 By: Lokesh Gupta Creational Patterns Design Patterns

The **builder pattern**, as the name implies, is an **alternative way to construct complex objects**. This pattern should be used when we want to build different immutable objects using the *same object building process*.

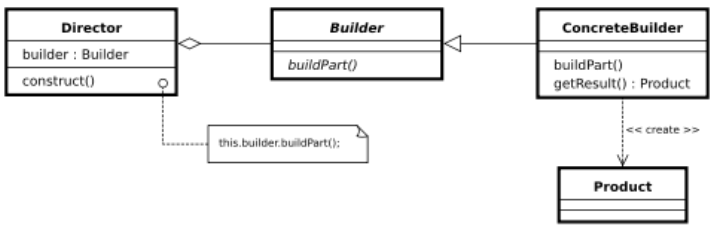
Table Of Contents

1. The GoF Builder Pattern

Before starting the discussion, I want to make it clear that the builder pattern which we are going to discuss in this post, is slightly different from what is mentioned in GangOfFour "Design Patterns" book. The book says:

> The builder pattern is a design pattern that allows for the step-by-step creation of complex objects using the correct sequence of actions. The construction is controlled by a director object that only needs to know the type of object it is to create.

And the book gives examples like below:



I really find it hard to make use of the above example in real-life programming and applications. The above process is very much similar (not exactly) to the abstract factory pattern, where we find a factory (or builder) for a specific type of object, and then the factory gives us a concrete instance of that object.

The only big **difference between the builder pattern and the abstract factory pattern** is that builder provides us more control over the object creation process and that's it. Apart from it, there are no major differences.

> *In one sentence, abstract factory pattern is the answer to* **"WHAT"** *and the builder pattern to* **"HOW"**.

Now from here, we will start discussing the builder pattern the way I find it useful especially in practical cases.

2. Definition of Builder Pattern

Let's start by giving a definition of builder pattern:

> Builder pattern aims to "Separate the construction of a complex object from its representation so that the same construction process can create multiple different representations."

A builder pattern should be more like a fluent interface. A fluent interface is normally implemented by using **method cascading** (or method chaining) as we see it in lambda expressions.

3. Where We Require Builder Pattern?

We already know the benefits of immutability and immutable instances in an application. If you have any questions about it, let me remind you of the `String` class in Java. And as I already said, the builder pattern **helps us in creating immutable classes with a large set of state attributes**.

Let's discuss a common problem in our application. In any user management module, the primary entity is `User`, let's say. Ideally and practically as well, once a *User* object is fully created, we will not want to change its state. It simply does not make sense, right?

Now, let's assume, our `User` object has the following 5 attributes i.e. `firstName`, `lastName`, `age`, `phone` and `address`.

In normal practice, if we want to make an immutable User class, then we must pass all five information as parameters to the constructor. It will look like this:

```
public User (String firstName, String lastName, int age, String phone, String address){
    this.firstName = firstName;
    this.lastName = lastName;
    this.age = age;
    this.phone = phone;
    this.address = address;
}
```

Very good. Now what if only `firstName` and `lastName` are **mandatory** and the rest 3 fields are optional. Problem !! We need more constructors. This problem is called the **telescoping constructors problem**.

```
public User (String firstName, String lastName, int age, String phone){ ... }
public User (String firstName, String lastName, String phone, String address){ ...  }
public User (String firstName, String lastName, int age){ ...   }
public User (String firstName, String lastName){ ...    }
```

We will need some more like above. Still can manage? Now let's introduce our sixth attribute i.e. salary. Now it is problem.

One way is to create more constructors, and another is to lose the immutability and introduce setter methods. You choose any of both options, you lose something, right?

Here, the builder pattern will help you to consume additional attributes while retaining the immutability of the `User` class.

4. Implementing Builder Pattern

> Lombok's @Builder annotation is a useful technique to implement the builder pattern.

Let's solve the above problem in code. The given solution uses an additional class `UserBuilder` which helps us in building desired `User` instance with all mandatory attributes and a combination of optional attributes, without losing the immutability.

User.java

```
public class User
{
    //All final attributes
    private final String firstName; // required
    private final String lastName; // required
    private final int age; // optional
    private final String phone; // optional
    private final String address; // optional

    private User(UserBuilder builder) {
        this.firstName = builder.firstName;
        this.lastName = builder.lastName;
        this.age = builder.age;
        this.phone = builder.phone;
        this.address = builder.address;
    }

    //All getter, and NO setter to provde immutability
    public String getFirstName() {
        return firstName;
    }
    public String getLastName() {
        return lastName;
    }
    public int getAge() {
        return age;
    }
    public String getPhone() {
        return phone;
    }
    public String getAddress() {
```

```
        return address;
    }

    @Override
    public String toString() {
        return "User: "+this.firstName+", "+this.lastName+", "+this.age+", "+this.phone+", "+
    }

    public static class UserBuilder
    {
        private final String firstName;
        private final String lastName;
        private int age;
        private String phone;
        private String address;

        public UserBuilder(String firstName, String lastName) {
            this.firstName = firstName;
            this.lastName = lastName;
        }
        public UserBuilder age(int age) {
            this.age = age;
            return this;
        }
        public UserBuilder phone(String phone) {
            this.phone = phone;
            return this;
        }
        public UserBuilder address(String address) {
            this.address = address;
            return this;
        }
        //Return the finally consrcuted User object
        public User build() {
            User user =  new User(this);
            validateUserObject(user);
            return user;
        }
        private void validateUserObject(User user) {
            //Do some basic validations to check
            //if user object does not break any assumption of system
        }
    }
}
```

And below is the way, we will use the `UserBuilder` in our code:

UserBuilder Example

```
public static void main(String[] args)
{
    User user1 = new User.UserBuilder("Lokesh", "Gupta")
```

```
        .age(30)
        .phone("1234567")
        .address("Fake address 1234")
        .build();

        System.out.println(user1);

        User user2 = new User.UserBuilder("Jack", "Reacher")
        .age(40)
        .phone("5655")
        //no address
        .build();

        System.out.println(user2);

        User user3 = new User.UserBuilder("Super", "Man")
        //No age
        //No phone
        //no address
        .build();

        System.out.println(user3);
    }
```

Please note that the above-created `User` object **does not have any setter method**, so its state can not be changed once it has been built. This provides the desired immutability.

Sometimes developers may forget to add a few attributes to the *User* class. While adding a new attribute and containing the source code changes to a single class (SRP), we should enclose the builder inside the class (as in the above example). It makes the change more obvious to the developer that there is a relevant builder that needs to be updated too.

*Sometimes I think there should be a **destroyer pattern** (opposite to builder) that should tear down certain attributes from a complex object in a systematic manner. What do you think?*

5. Existing Implementations in JDK

All implementations of java.lang.Appendable are infact good examples of the use of Builder pattern in java. e.g.

- java.lang.StringBuilder#append() [Unsynchronized class]
- `java.lang.StringBuffer#append()` [Synchronized class]
- java.nio.ByteBuffer#put() (also on CharBuffer, ShortBuffer, IntBuffer, LongBuffer, FloatBuffer and DoubleBuffer)
- Another use can be found in javax.swing.GroupLayout.Group#addComponent().

Look how similar these implementations look to what we discussed above.

```
StringBuilder builder = new StringBuilder("Temp");
```

```
String data = builder.append(1)
        .append(true)
        .append("friend")
        .toString();
```

6. Advantages

Undoubtedly, the number of lines of code increases at least to double in the builder pattern, but the effort pays off in terms of **design flexibility** and much more **readable code**.

The **parameters to the constructor are reduced** and are provided in **highly readable chained method calls**. This way there is **no need to pass in `null` for optional parameters** to the constructor while creating the instance of a class.

Another advantage is that an **instance is always instantiated in a complete state** rather than sitting in an incomplete state until the developer calls (if ever calls) the appropriate "setter" method to set additional fields.

And finally, we can build **immutable objects** without much complex logic in the object building process.

7. Disadvantages

Though the Builder pattern reduces some lines of code by eliminating the need for setter methods, still it **doubles up total lines** by introducing the builder object. Furthermore, although client code is more readable, the client code is also **more verbose**. Though for me, readability weighs more than lines of code.

That's the only disadvantage I can think of.

Happy Learning !!

Recommended Reading:

1. Prototype design pattern in Java
2. Lombok @Builder
3. Guide to Retrofit.Builder API

4. Lombok Serialize and Deserialize @Builder Class
5. Chain of Responsibility Design Pattern
6. Adapter Design Pattern in Java
7. Template Method Design Pattern
8. Command Design Pattern
9. Flyweight Design Pattern
0. State Design Pattern

43 thoughts on “Builder Design Pattern”

Mragendra Singh

April 2, 2020 at 1:30 pm

Hey buddy, I never find best example of builder pattern then your's example. You explained in very easy way, while many tutorial make it complex & hard to understand. and ur way is superb. Keep going.

Reply

Monica Luo

December 5, 2019 at 6:48 am

Greate article.
Have a question of this block:
public UserBuilder(String firstName, String lastName) {
this.firstName = firstName;
this.lastName = lastName;
}
Should we return this as well? like other funcs

Reply

Monica luo

December 5, 2019 at 6:54 am

Never mind, I figured it out. it's constructor of UserBuilder, and we don't need to return.

Reply

Alwyn Schoeman

November 13, 2019 at 6:57 am

I like the idea of a Destroyer pattern, but it sounds like an impossible task.

Reply

Rishabh Sakhare

July 14, 2019 at 5:04 am

I have a class User (same as your User). When I try to inherit the User class, it gives compilation error that "no default constructor available".

```
public  class Electronic extends User{

    public enum ElecList{
        TV,
        MOBILE,
        REFRIGERATOR
    }

}
```

Reply

Roman Elizarov

March 2, 2020 at 3:13 pm

Yes because the constructor for User class is private and classes with private constructors cannot be inherited.

Reply

Kapil

December 27, 2021 at 11:31 am

its a immutable class, they are not meant to be extended.

Reply

pranit

December 13, 2017 at 10:15 am

Great article!
You have provided a great insight here about builder design pattern in java. Designer pattern allows to develop complicated object phase by phase and also makes sure a way to develop an object as a finished

object. It is very helpful for all java debvelopers to know about builder pattern in java, i have enjoyed this article, thanks a lot for sharing!

Reply

Prakriti

August 19, 2017 at 6:47 am

I'm not aligned on jdk example of Builder Pattern of StringBuilder#append and StringBuffer#append because these two are not immutable and also Builder pattern is best suitable for the classes that has large number of instance fields where as for StringBuilder#append and StringBuffer#append only single value is required. Please correct me if i'm missing anything in my understanding.

Reply

Lokesh Gupta

August 19, 2017 at 10:52 am

I will suggest you to look at the pattern from steps to create the full object.. rather number of fields. That will make things more clear.

Reply

Prakriti

August 20, 2017 at 10:16 am

Thank you, it resolves my doubt.

Reply

Prakriti

August 19, 2017 at 6:37 am

Very nicely explained.
I have a query in this pattern that as User class might have some fields as mandatory and some as optional so even if object is built using builder pattern that has only few optional fields set and others will be null then

how does it make difference to create object using constructor in the same class that will have required and applicable fields set with some value and not applicable fields as null. Please help me understand this.

Thanks.

Reply

Lokesh Gupta

August 19, 2017 at 10:50 am

Very good reasoning. You are correct that with mentioned optionality, there seems no difference. In last, it all boils down to readability and ability to create immutable objects, which is not possible using constructors.

Reply

Prakriti

August 20, 2017 at 10:18 am

That is the good point, thank you.

Reply

S. A. Oluoch

September 5, 2017 at 6:11 pm

Actually there is a clear difference. As stated before, once there is an optional object in the constructor, the developer is forced to initiate with a nullin cases where the value is not known. In the builder factory pattern, the optional fields are simply skipped and the code looks neater. The developer doesn't have to instantiate them.

Reply

Datt

July 11, 2019 at 3:23 pm

Hi Lokesh!

First of all, sincere thanks for your clear explanation! Can you please clarify my query below?

With the example code that you provided, the immutable User instance is created for no doubt. But if the User wants to update some detail, let's say the phone number, please explain clearly how the behaviour would be. Will it be again creating a new instance or update the existing instance.

Thank you

Regards
Datt

Reply

Lokesh Gupta
July 11, 2019 at 10:07 pm

In such cases, I would not like to change the existing user object. Rather, I will define a method `fromUser(User u)` in userbuilder which will take an existing user instance and populate create a new User object with same data and a chance to update desired fields before constructing the object.

Something like this:

```
User changedUser = new User.UserBuilder().fromUser(existingUser)
    .age(40)
    .phone(&amp;quot;5655&amp;quot;)
    .build();
```

Reply

Datt
July 12, 2019 at 12:16 pm

Thanks for your reply Lokesh!

If we follow the above approach,
1. How about the first user object that was created before modification?
2. Also, if the new object is created with modified data as mentioned, there could be chance for two identical objects to exist with first name and last name as common in both of them and other details in different way. If this point is correct, Won't it be a memory loss? Please correct if am going wrong in understanding.

If possible please put this code patch in main program for better reference.

Thanks

Lokesh Gupta
July 13, 2019 at 7:04 am

In that case, you can provide methods for firstName and lastName fields also. Don't make them final and treat as other fields.

Christopher Barrett
March 7, 2017 at 11:40 pm

I am studying for my Java professional Exam and I really can't see the huge advantage of this pattern.

For example let's say I have an Animal class and an AnimalBuilder class:
Animal needs a new mandatory parameter. Not only do I have to change it in the Animal class, but in the AnimalBuilder class, plus all the classes using AnimalBuilder.

To be honest all I can see in terms of advantages is more readable code, but even then it is a stretch.

Please tell me I am wrong and show me the light 🙂

Reply

Lokesh Gupta
March 8, 2017 at 12:38 pm

Adding a new mandatory parameter is a big design change. To accommodate this change, you may follow below steps:

1) Add attribute to Animal class, assign some default value to it, AND one more constructor which sets its value from parameter.
2) Modify AnimalBuilder. Add new builder method to use this new constructor.

Till now, no change is needed in any other class. Now, you may use new builder method in new classes OR may use in older classes where its absolutely necessary.

Reply

Anjith

December 7, 2015 at 12:23 pm

The GOF book actually says “Separate the construction of a complex object from its representation so that the same construction process can create different representations.”. Please correct your post.

Reply

Lokesh Gupta

December 13, 2015 at 7:17 am

Hi Anijith, I clearly mentioned that this definition is slightly different from what is mentioned in GangOfFour.

Reply

Anjith

December 13, 2015 at 4:11 pm

Your post says “The book says:
The builder pattern is a design pattern that allows for the step-by-step creation of complex objects using the correct sequence of actions. The construction is controlled by a director object that only needs to know the type of object it is to create.”

But, I don’t see the above description in the GOF book.

Instead the book says”
“Separate the construction of a complex object from its representation so that
the same construction process can create different representations.”

Reply

f3tekkencode

October 1, 2015 at 9:33 am

Brilliant! Thank you for sharing. Been Inspired, so I ventured to borrow your example and do another with the same goal but slightly different idea.

```
public interface User
{
  //User now been declared as an interface, thus is immutable
  //All getters need to be implemented
    public String getFirstName();
    public String getLastName();
    public int getAge();
    public String getPhone();
    public String getAddress();

    public static class UserBuilder
    {
        private final String firstName;
        private final String lastName;
        private int age;
        private String phone;
        private String address;

        public UserBuilder(String firstName, String lastName) {
            this.firstName = firstName;
            this.lastName = lastName;
        }
        public UserBuilder age(int age) {
            this.age = age;
            return this;
        }
        public UserBuilder phone(String phone) {
            this.phone = phone;
            return this;
        }
        public UserBuilder address(String address) {
            this.address = address;
            return this;
        }
        //Return the finally consrcuted User object
        public User build() {
            User user =  newUser();
            validateUserObject(user);
            return user;
        }
        private void validateUserObject(User user) {
            //Do some basic validations to check
            //if user object does not break any assumption of system
        }
        //Way to construct
        private User newUser() {
          //Introduce anonymous class
          return new User() {
            //All final attributes
              private String firstName; // required
              private String lastName; // required
              private int age; // optional
              private String phone; // optional
              private String address; // optional

              private User basedON(UserBuilder builder) {
                  this.firstName = builder.firstName;
                  this.lastName = builder.lastName;
                  this.age = builder.age;
                  this.phone = builder.phone;
                  this.address = builder.address;
                  return this;
              }

              public String getFirstName() {
                  return firstName;
              }
              public String getLastName() {
```

```
                return lastName;
            }
            public int getAge() {
                return age;
            }
            public String getPhone() {
                return phone;
            }
            public String getAddress() {
                return address;
            }

            @Override
            public String toString() {
                return "User: "+this.firstName+", "+this.lastName+", "+this.age+", "+this.phone+", "+
            }
        }.basedON(this);
    }
}
```

Reply

Lokesh Gupta

October 1, 2015 at 9:50 am

Thanks for sharing this.

Reply

anu

March 26, 2015 at 10:52 am

awesome!! really I have gone through many sites but couldnt understand the builder pattern this much clearly… Thanks 🙂

Reply

aniket

January 14, 2015 at 9:49 pm

I am trying to implement a similar problem,except that it has some nested builders.I have the builders created separately. Now,the problem is how do I write the client code . Also,I have seen implementations using

Interfaces,when are they used? is it like interfaces give us the ability to have a control over the sequence in which we want to call out the methods?
I have a user class:
with similar fields as to what you have,
Then I have an Address class with usual fields such as city,state,zip etc.
How do I call the builder in Address inside the builder for User?
And while creating the user I want to give the option of creating the address or not. Can this be done? DO you need the builder code here? its pretty big,hence did not add.But,assuming its on the similar lines of yours how would go ahead and do it? What I have tried is this..

public HomeAddressBuilder havingHomeAddressAs() {
//HomeAddress is the other class,and it contains the static class for the builder inside it.
return new HomeAddress.HomeAddressBuilder();
}
is that the correct way?
so if I had the user builder will it look like this:

```
User user1 = new User.UserBuilder("Lokesh", "Gupta")
            .age(30)
            .phone("1234567")
            .havingHomeAddressAs() //the above method
            .createMyUserHomeAddress() //the method inside my address builder.
            .withCity("ccc")
            .withState("sss")
            .createAndReturnMyAddress() //this marks the end of address builder
            .build();//this marks the end of the user builder
```

Reply

Lokesh Gupta

January 15, 2015 at 3:56 am

If I will be in your place, I will not build address using userbuilder reference. Address is another example of complex data having lot's of possible fields with multiple permutations and combinations internationally. So I will create a separate builder object for address as well.

//Build the address using addressbuilder
//Build the user using userbuilder AND set homeaddress obtained in above step.

This approach is good for two reasons:

1) Your code remain clean and easy to extend and modify. Always prefer simplicity in you code. Never make it unnecessarily complex.
2) There is no dependency between user class and address class builders. So any change in one class will not affect other class. **Single Responsibility Principle**.

Reply

aniket

January 15, 2015 at 4:56 am

Hi Lokesh,
Thank you very much for the reply.I got around that problem.I actually had created 2 separate classes for the purposes you mentined,sorry for mentioning the nested word.I also had 1 more problem. My test team wants to create a user even though no field has been set,so my approach was,when I am telling my builder to return the user object back,before that check if any fields have been set to null or are empty strings.If they are ,then recreate the user object with default-values.And then return the user.Is this approach correct?
Also,another problem I see with this approach is,I will be getting the same method name appearing twice,meaning,if set the name of the user for the first time,and I want to set some other properties, i will still be able to select the setName method and set the name again,i.e. i am not able to restrict the fields/reduce the fields that will now be set,I have an approach of creating the interfaces and making return type of the next field,i.e. make each field mandatory.Is that a good approach?
Thank you very much for the response.

Reply

Lokesh Gupta

January 15, 2015 at 6:35 am

I will suggest to create a constructor with all mandatory fields as arguments. This way you can force the coder to pass all mandatory fields even before getting the builder object. This will also remove unnecessary checks for mandatory fields inside build() method. You can remove the setter methods for these mandatory fields to avoid overriding them wrongly.
Note: Keep the number of mandatory fields to minimum.

Reply

No

December 4, 2014 at 6:19 pm

“java.lang.Appendable are in fact good example of use of Builder pattern” ???
😕

Reply

Lokesh Gupta

December 5, 2014 at 6:27 am

"All implementations of java.lang.Appendable are infact good example".. Can you please tell me, why you disagree?

Reply

sohaib

July 24, 2014 at 11:54 am

```
public class RegistrationDoaHelper {

    Registration register = null;

    public RegistrationDoaHelper(Context context) {
        register = new Registration(context);
    }

    public long insertData(String username, String password, String email) {

        SQLiteDatabase db = register.getWritableDatabase();
        ContentValues contentValues = new ContentValues();
        contentValues.put("username", username);
        contentValues.put("password", password);
        contentValues.put("email", email);
        long id = db.insert(Registration.TABLE_NAME, null, contentValues);
        return id;
    }

    public Cursor getAllForListView() {

        SQLiteDatabase db = register.getWritableDatabase();
        String columns[] = { Registration.UID, Registration.USERNAME, Registration.PASSWORD, Registrati
        Cursor cursor = db.query(Registration.TABLE_NAME, columns, null, null,
                null, null, null);

        if(cursor!=null) {
            cursor.moveToNext();
        }
        return cursor;
    }

    public String getAllData() {

        //Map&lt;String,String&gt; mapData = new HashMap&lt;String, String&gt;();

        SQLiteDatabase db = register.getWritableDatabase();
        String columns[] = { Registration.UID, Registration.USERNAME,
                Registration.PASSWORD, Registration.EMAIL };
        Cursor cursor = db.query(Registration.TABLE_NAME, columns, null, null,
                null, null, null);

        StringBuffer buffer = new StringBuffer();
        while (cursor.moveToNext()) {
```

```
            int uid = cursor.getInt(0);
            String username = cursor.getString(1);
            String password = cursor.getString(2);
            String email = cursor.getString(3);

            /*mapData.put("username", username);
            mapData.put("password", password);
            mapData.put("email", email);*/

            buffer.append(uid + " " + username + " " + password + " " + email
                    + "\n");
        }
        //return mapData;
        return buffer.toString();
    }

    public static class Registration extends SQLiteOpenHelper {

        private static final String TAG = "DATABASE";

        private static final String DATABASE_NAME = "demo";
        private static final String TABLE_NAME = "REGISTER";
        private static final int DATABASE_VERSION = 1;

        private static final String UID = "_id";
        private static final String USERNAME = "username";
        private static final String PASSWORD = "password";
        private static final String EMAIL = "email";

        private static final String CREATE_TABLE = "CREATE TABLE " + TABLE_NAME
                + "(" + UID + " INTEGER PRIMARY KEY AUTOINCREMENT, " + USERNAME
                + " VARCHAR(255), " + PASSWORD + " VARCHAR(255), " + EMAIL
                + " VARCHAR(255))";

        private static final String DROP_TABLE = "DROP TABLE " + TABLE_NAME
                + " IF EXISTS";

        private Context context;

        public Registration(Context context) {
            super(context, DATABASE_NAME, null, DATABASE_VERSION);
            this.context = context;
            Log.d(TAG, "Constructor called");

            ToastMessage.toastMessage(context, "Constructor Called");
        }

        @Override
        public void onCreate(SQLiteDatabase db) {

            try {

                db.execSQL(CREATE_TABLE);
                Log.d("onCreate", "onCreate called");

                ToastMessage.toastMessage(context, "On Create Called");

            } catch (SQLException e) {

                e.printStackTrace();
            }

        }

        @Override
        public void onUpgrade(SQLiteDatabase db, int oldVersion, int newVersion) {
```

```
            try {

                db.execSQL("DROP TABLE IF EXISTS" + TABLE_NAME);
                Log.d(TAG, "onUpdate called");

                ToastMessage.toastMessage(context, "DATABASE DROPED");

                onCreate(db);
                Log.d(TAG, "onCreate called inside onUpdate");

                ToastMessage.toastMessage(context,
                        "onCreate called after onUpdate");

            } catch (SQLException e) {
                e.printStackTrace();
            }
        }
    }
```

Is above code is example of builder pattern?

Reply

Lokesh Gupta

July 24, 2014 at 12:03 pm

I am not able find any reason to say any piece of above code as builder pattern. Perhaps I am missing something you want to say. Can you please highlight, what and why in above code you see a possibility of builder pattern.

Reply

sohaib

July 24, 2014 at 2:05 pm

I am just new to learn design pattern and I want to learn design pattern when in real world scenario I cannot think how to solve real world problem using design patterns.

Please do not mind on my stupid question. I post above question because of I have Registration inner class and RegistrationDalHelper outer class. In registration class I have not create any getter and setter. I access these value inside the RegistrationDaoHelper class because of that I have create the Outer class and make inner registration class.I can create the getter for the registation private members varialbe to accesss these outside the class.

I just post question because I have found that Inner and outer class inside the code.
Sorry again for stupid questions.
Please tell me why you have not find any reason for above code to be a builder patter.Please make just

bullet points
Thanks

Reply

Lokesh Gupta
July 24, 2014 at 6:58 pm

Hi Sohaib, no need to say sorry ever. And there are no stupid questions in discussions.
Further, Builder pattern is used to build an object using simple steps; which otherwise will take a very complex logic to build. Breaking building process in steps keep it simpler and easy to use. An example I have already covered into the tutorial.
Your example can be more related to Proxy pattern where a class functioning as an interface to something else.

Reply

sohaib
July 25, 2014 at 12:12 am

Thanks

Nadim Sheikh
June 5, 2014 at 6:29 am

Really Awesome Article …… Getting lot of knowledge from your article Lokesh.

Reply

vinodbeli
May 16, 2014 at 8:25 pm

Hi Lokesh,

After your super duper blogs it made me to read more artcle ..after chetan bhagat ..I felt to read is only your blogs 🙂 …

I liked your StringBuilder with any data type append method implemented using builder pattern , but in your case
I am using inner class for object creation and it divert me instead of building my own class using some inner class…is not possible like StringBuilder.

```
public class Pizza {
private final int size;
private final boolean cheese;
private final boolean corn;
public Pizza(int size){
//check the state
if(0==size){
throw new IllegalStateException("oop! size zero pizza not exist in world :)");
}
this.size = size;
}
public Pizza cheese(final boolean cheese){
this.cheese = cheese;
return this;
}

public Pizza corn(final boolean corn){
this.corn = corn;
return this;
}

}
```

May I wrong in this case?

Thanks
Vinod

Reply

Lokesh Gupta
May 19, 2014 at 7:10 am

Vinod, thanks for the kind words.
Next, your question about Pizza example. I strongly believe that design patterns are just concepts, and they do not enforce any particular implementation (literally any). So, if you are making the pizza building process easy by breaking the whole process into multiple small steps such that you can call PizzaBuilder().createBase().addStuff1().addStuff2()… or any such pattern, then you have implemented builder pattern.

Whether you have used inner classes or not, doesn't make any difference. It's only a design solution, not implementation guideline.

Reply

Ujjawal
May 12, 2014 at 9:24 am

Hi Lokesh,

Nicely explained. But I have one question. You stated the following line before explaining builder pattern: "Now what if only firstName and lastName are mandatory and rest 3 fields are optional."

How is this problem solved using Builder as I still need to include all fields and their values while object creation in the line:
new User.UserBuilder("Lokesh", "Gupta").age(30).phone("1234567").address("Fake address 1234").build();

Also, if additional fields are added, they also need to be included.

Reply

Lokesh Gupta
May 12, 2014 at 9:33 am

Ujjawal, If you write new User.UserBuilder("Lokesh", "Gupta").build(); then also you will get a User object with only first name and lastname set. That means other fields are optional. If want to set age then call ".age(24)" OR if you don't want to set it, simply leave it. So essentially they are optional.

Reply

Leave a Comment

Name *

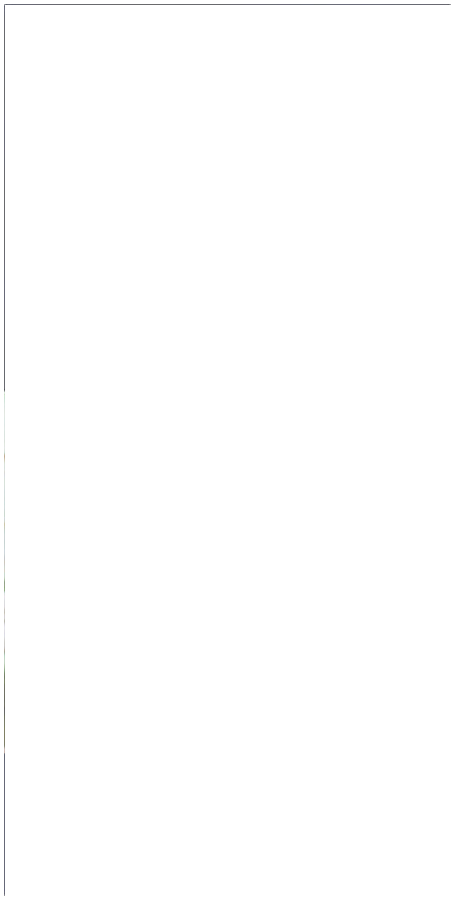
Email *

Website

Add me to your newsletter and keep me updated whenever you publish new blog posts

Post Comment

Search …



HowToDoInJava

A blog about Java and related technologies, the best practices, algorithms, and interview questions.

Meta Links

- About Me
- Contact Us
- Privacy policy

- Advertise
- Guest Posts

Blogs

REST API Tutorial