BUILDER Design Pattern

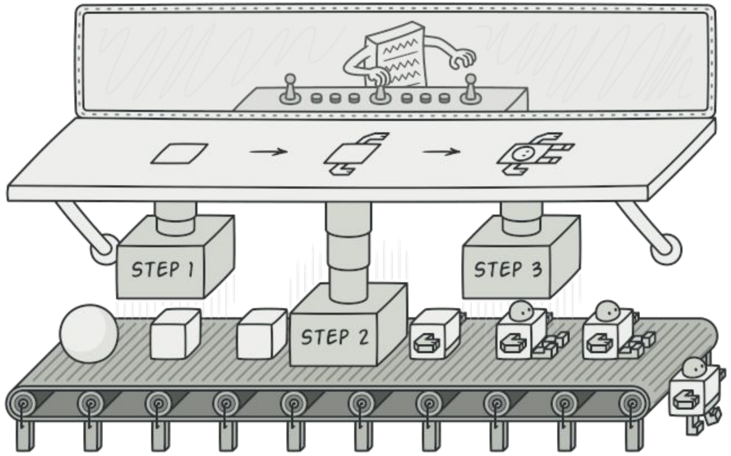
Upcode Software
Engineer Team

-2023-

CONTENT

1. What is BUILDER ?

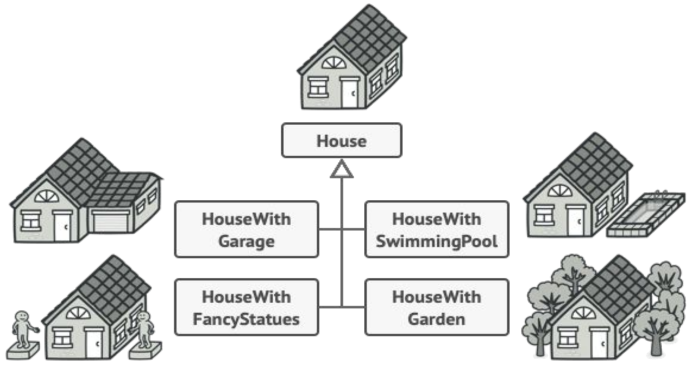
- The primary purpose of the Builder pattern is to help with the creation of a complex object.
- **The complexity of an object requires a step-by-step creation and cannot, or is hard to, be reduced to a list of parameters.**



2. WHY needs to BUILDER ? (1/n)

- When the complexity of creating object increases, the Builder pattern can separate out the instantiation process by using another object (a builder) to construct the object.
- This builder can then be used to create many other similar representations using a simple step-by-step approach.
- However, this pattern has two versions that differ dramatically in their goals and intentions.

1. **Simple Builder.**
2. **Classic Builder.**



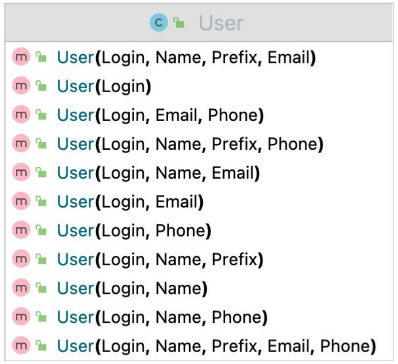
2. WHY needs to BUILDER ? (2/n)

- When the implementation of an interface or an abstract class is expected to change frequently
- When the current implementation cannot comfortably accommodate new change
- When the initialization process is relatively simple, and the constructor only requires a handful of parameters

3. HOW to use BUILDER ? (1/n)

Simple Builder

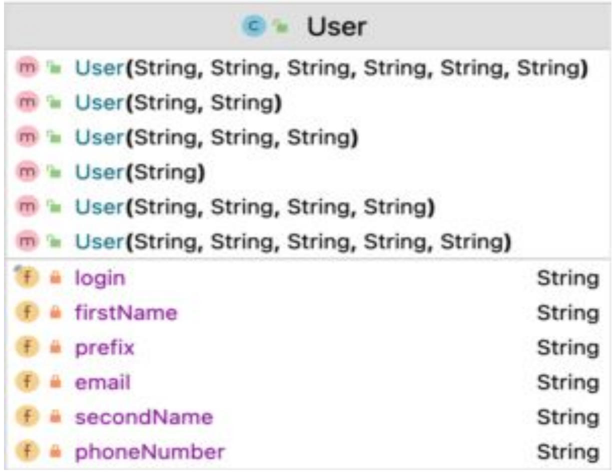
- This version of the Builder pattern mainly aims to make object creation more convenient. This pattern is useful when an object has optional fields, and there might be too many combinations.
- **Often this problem is resolved by telescopic constructors, which clutter the code and make it more error-prone.** Let's consider the following case.



```
User
User(Login, Name, Prefix, Email)
User(Login)
User(Login, Email, Phone)
User(Login, Name, Prefix, Phone)
User(Login, Name, Email)
User(Login, Email)
User(Login, Phone)
User(Login, Name, Prefix)
User(Login, Name)
User(Login, Name, Phone)
User(Login, Name, Prefix, Email, Phone)
```

3. HOW to use BUILDER ? (2/n)

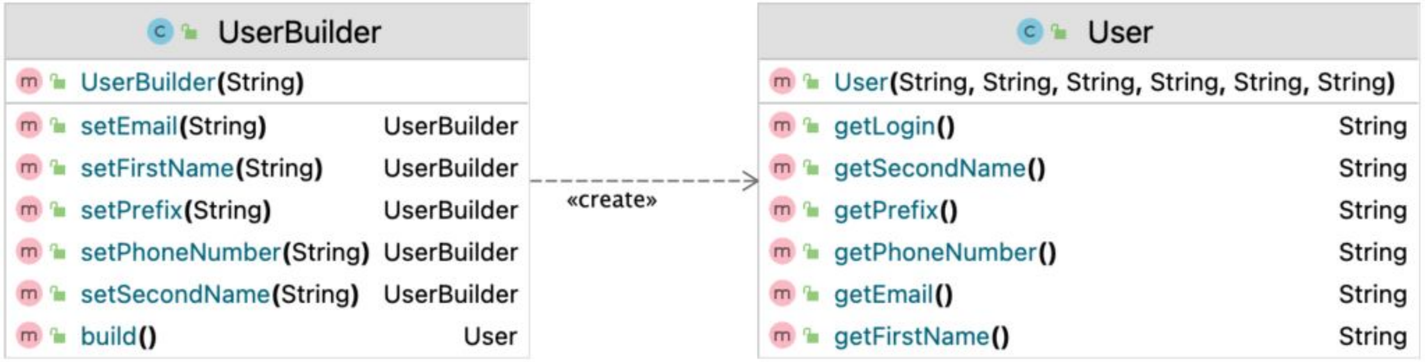
- The number of constructors would increase exponentially to cover all possible options. Also, if objects contain fields of the same type, it might be impossible to create all the variations because of overriding rules in most languages.
- **The last thing is that it might result in hard-to-debug issues if we confuse the order of the parameters:**



```
User
User(String, String, String, String, String, String)
User(String, String)
User(String, String, String)
User(String)
User(String, String, String, String)
User(String, String, String, String, String)
login          String
firstName      String
prefix         String
email          String
secondName     String
phoneNumber    String
```

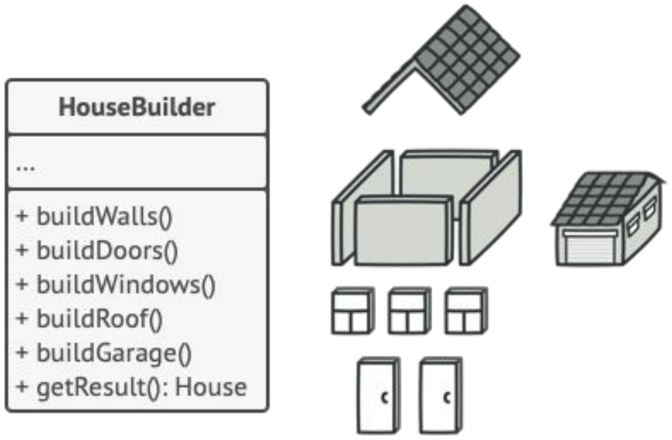
3. HOW to use BUILDER ? (3/n)

- In this case, the Builder pattern would be a reasonable way to make this class more maintainable. Sometimes, creating a base object and finishing the configuration with setters is possible, which is a simple way to resolve the problem.
- **However, we cannot use this for immutable objects.** Also, if the object contains specific rules for the creation, we might not be able to check them, creating the object step-by-step:


UserBuilder
UserBuilder(String)
setEmail(String) UserBuilder
setFirstName(String) UserBuilder
setPrefix(String) UserBuilder
setPhoneNumber(String) UserBuilder
setSecondName(String) UserBuilder
build() User
«create»
User
User(String, String, String, String, String, String)
getLogin() String
getSecondName() String
getPrefix() String
getPhoneNumber() String
getEmail() String
getFirstName() String

3. HOW to use BUILDER ? (4/n)

- A *UserBuilder* is a dedicated object that will store all the elements and create an object in one go. In this case, we can talk about the illusion of object creation in steps.
- Although the interface will create this illusion, the main power of the pattern is to make the entire object at once and ensure all the rules and constraints.



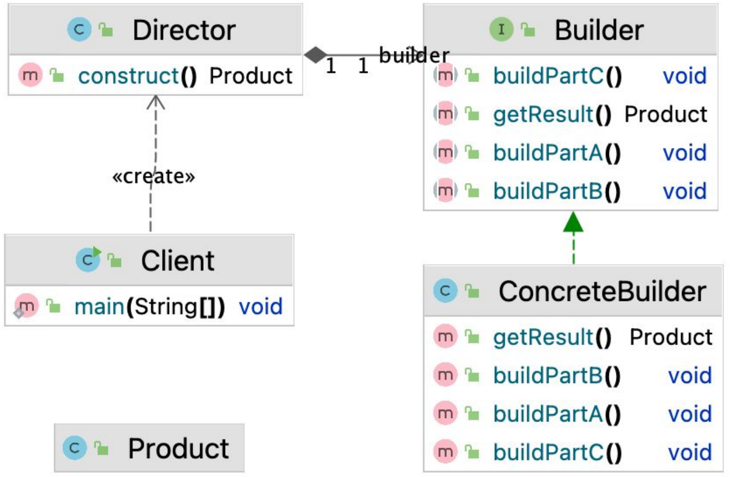
3. HOW to use BUILDER ? (5/n)

Classic Builder

- This version of the pattern aims to provide a way to create a complex object in steps.
- **The complexity emerges not from the number of parameters but from the object’s structure, which in most cases applies to** Composite **objects.**
- Objects with a complex inner structure often cannot be expressed with the number of parameters. A department with sub-departments is a good application for a Builder.

3. HOW to use BUILDER ? (6/n)

- The steps, or the rules, for creating an object are extracted to a *Director* class. This class contains complete knowledge of object creation.
- Usually, it involves graph traversing logic. If we want to translate an HTML web page into an object, we will use an algorithm, often recursive, rather than a recipe with several steps:



4. WHERE to use BUILDER ? (1/n)

- 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.

```
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;
	}
	code.
}
```

4. WHERE to use BUILDER ? (2/n)

```
        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;
                }
        code.
}
```

4. WHERE to use BUILDER ? (3/n)

- And below is the way we will use the **UserBuilder** in our code:

```
public static void main(String[] args)
{
        User user1 = new User.UserBuilder("Jahongir", "Nodirxo’ja")
        .age(30)
        .phone("1234567")
        .address("Fake address 1234")
        .build();

        System.out.println(user1);

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

        System.out.println(user2);

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

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

4. WHERE to use BUILDER ? (4/n)

- 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.

5. SOURCE code. (1/n)

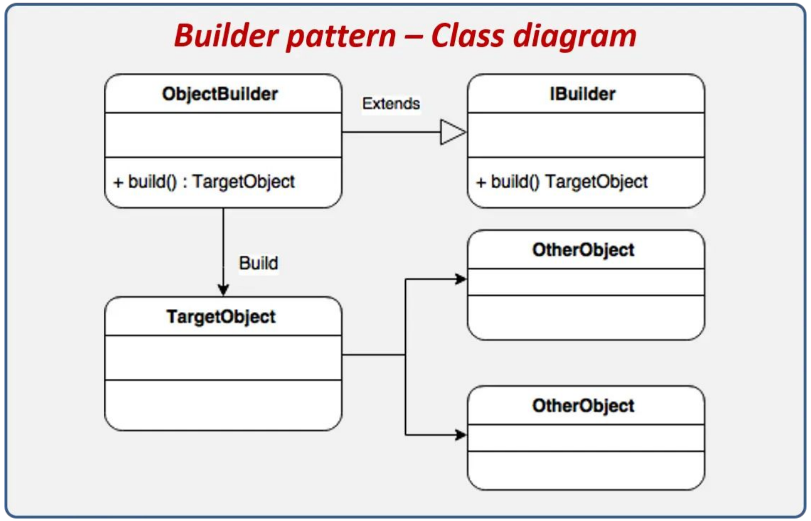
```
public class Computer {

        //required parameters
        private String HDD;
        private String RAM;

        //optional parameters
        private boolean isGraphicsCardEnabled;
        private boolean isBluetoothEnabled;

        public String getHDD() {
                return HDD;
        }
        public String getRAM() {
                return RAM;
        }
        public boolean isGraphicsCardEnabled() {
                return isGraphicsCardEnabled;
        }
        public boolean isBluetoothEnabled() {
                return isBluetoothEnabled;
        }
        private Computer(ComputerBuilder builder) {
                this.HDD=builder.HDD;
                this.RAM=builder.RAM;
                this.isGraphicsCardEnabled=builder.isGraphicsCardEnabled;
                this.isBluetoothEnabled=builder.isBluetoothEnabled;

        }
```

5. SOURCE code. (2/n)

```
//Builder Class
        public static class ComputerBuilder{

                // required parameters
                private String HDD;
                private String RAM;

                // optional parameters
                private boolean isGraphicsCardEnabled;
                private boolean isBluetoothEnabled;

                public ComputerBuilder(String hdd, String ram){
                        this.HDD=hdd;
                        this.RAM=ram;
                }
                public ComputerBuilder setGraphicsCardEnabled(boolean isGraphicsCardEnabled) {
                        this.isGraphicsCardEnabled = isGraphicsCardEnabled;
                        return this;
                }
                public ComputerBuilder setBluetoothEnabled(boolean isBluetoothEnabled) {
                        this.isBluetoothEnabled = isBluetoothEnabled;
                        return this;
                }
                public Computer build(){
                        return new Computer(this);
                }
        }

  }
```

5. SOURCE code. (3/n)

```
public class TestBuilderPattern {

        public static void main(String[] args) {
                //Using builder to get the object in a single line of code and
                 //without any inconsistent state or arguments management issues
                Computer comp = new Computer.ComputerBuilder(
                                "500 GB", "2 GB").setBluetoothEnabled(true)
                                .setGraphicsCardEnabled(true).build();
        }

     }
```



Summary.

- Thus, we have two types of this pattern. However, their goals and implementation are different.
- We can distinguish these patterns because the first one is used only to make it easier to create an object with many fields.
- The second is used for *Composite* objects to provide a way to create them step-by-step.

Reference Resources

1. What is BUILDER ?
2. WHY needs to BUILDER ?
3. HOW to use BUILDER ?
4. Dive into design pattern compression (book).

Thank You.

Sherjonov Jahongir
jakhongirsherjonov@gmail.com

Tursunov Nodirxo`ja
ameriqano@gmail.com