Builder pattern

In creational design patterns, the main focus is on making a separate method for object creation.

In some types of these patterns, it may spawn different sets of connected subclasses which are really hard to track.

To avoid this problem you may use the builder pattern.

What is the builder?

Builder pattern is a creational design pattern that allows you to create objects step by step.

This approach allows creating fundamentally the same objects using one constructor, but they can be modified to depict different types of objects.

The main goal of the builder pattern is to separate the construction of a complex object from its representation so that the same construction process can create different representations.

For example, you have a restaurant that serves different types of dishes to guests.

All dishes need to be cooked and they all consist of food ingredients that should be prepared and added.

The basic way to recreate this situation in form of code will be to create a dish class that will be extended to a set of different subclasses to cover all combinations of the parameters.

This solution will spawn a large number of subclasses which will make a really complex hierarchy that you need to extend every time when you need to add a new parameter. So what will change if we'll use the builder pattern?

Builder as a solution

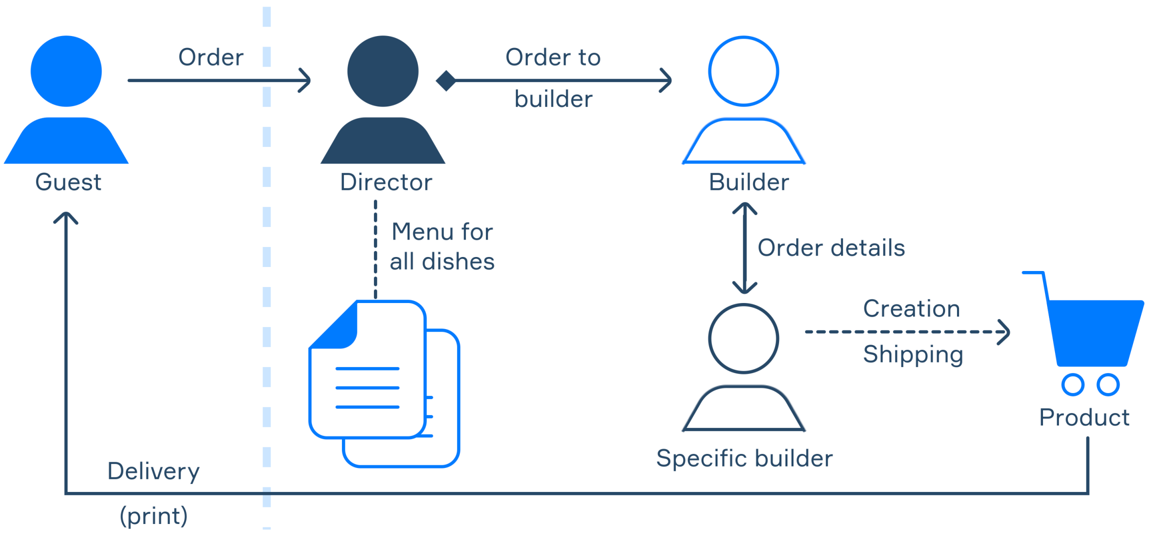
The point of builder is to organize separate construction classes called **builders.**

These classes will create separate objects with their set of parameters. They all work with pre-made steps that are defined in a **builder interface.** In our example, these steps are associated with food, so they will be something like cook(), salt(), pepper(), etc.

You can also have a separate class called director. It defines the order in which to execute the building steps, while the builder implements those steps.

This isn't necessary, but this class might be a good place to put various construction routines so you can reuse them in your program. It also hides the construction process from the client.

So when you need a specific dish, you should call a specific builder that creates it. Here's a scheme of how it could look:



In our system, a guest will make an order for a dish to the director. Then the director will send an order to the builder interface which will call a specific builder class that will make an order for this guest.

Builder class example

Let's try to depict this pattern in form of pseudocode. First, we define our builder interface, which will be called Dish:

```
interface Dish is
  method serve()
  method fry()
  method stew()
  method addIngredient()
  method addSalt()
  method addPepper()
  method addSpice()
  method addOil()
```

Now let's specify builders for some dishes. Here's an example for a fried dish:

```
class FriedDish implements Dish is
  method addIngredient() is
    ...

  method addOil() is
    ...

  method addSalt() is
    ...

  method addPepper() is
    ...

  method fry() is
    ...

  method serve() is
    ...
```

Here we just need to add some ingredient, oil, salt, pepper and fry it.
And now let's describe the builder for a stewed dish. In this builder we'll need to add ingredient and stew it:

```
class StewDish implements Dish is
  method addIngredient() is
    ...

  method addSpice() is
    ...

  method addPepper() is
    ...

  method stew() is
    ...

  method serve() is
    ...
```

Builder implementation

To use our builder class, we will define our director class that will allow us to stew or fry a dish:

```
class Director is
  method fry(Dish dish) is
   dish.addIngredient()
   dish.addOil()
   dish.addSalt()
   dish.addPepper()
   dish.fry()
   dish.serve()

  method stew(Dish dish) is
   dish.addIngredient()
   dish.addSpice()
   dish.addPepper()
   dish.stew()
   dish.serve()
```

Now we will use these builders to create our dishes:

```
method makeDish() is
  Director director = new Director()

  StewDish stewedChicken = new StewDish()
  director.stew(stewedChicken)

  FriedDish friedChicken = new FriedDish()
  director.fry(friedChicken)
```

You should use this pattern when you want your code to be able to create different representations of some product. Builder will allow you to reuse the same construction for them. However, it will require creating multiple new classes which might make your code more complex.

Conclusion

Builder pattern is a great way to create many similar objects with minor (or not so minor) differences. It will allow you to reduce the complexity of your code hierarchy, but it could also increase the number of your classes.