**Fluent Builder – a tasty approach to create complex objects**

The classic Builder design pattern is one of the creational design patterns used in object-oriented programming, as defined by the Gang of Four, and helps to separate the creation of complex objects from their implementation. This can be very useful when you want to create representations of the same object, with 3, 4 or even more attributes, over and over again.

As part of this pattern there is a *Director* class, that orchestrates the creation of something. The *Builder* is usually an abstract interface that specifies how to build that something in small incremental steps. A *Concrete Builder* will implement the builder interface to create different representations of your object and track them as they change. The concrete builder also has the ability to retrieve the final *Product*, which is the representation of the complex object that you want to build in the first place.

(builder diagram)

Although this design pattern reduces complexity and encapsulates the creation and assembly of different parts of a complex object, sometimes our code might become harder to read. Every so often you might want to make it more business readable and highlight the domain concepts. This is where a fluent interface comes into play. Not to be confused with method chaining, even though method chaining is a way to implement a fluent interface.

If we take a look at the creation of a *Pizza* object (sorry if this will make you hungry), it could look like this:

var capricciosa = new Pizza("large", "thin", "tomato", new List<string> { "mozzarella", "italian ham", "mushrooms", "olives" });

The constructor for *Pizza* forces you to pass all those attributes every time you want to create a new representation. If you want flexibility you might end up creating different constructors in your class. One that might not accept a sauce, one that might take an extra attribute when your pizza is special like a *calzone*, and the list goes on.

What if we could instead do:

var pizzaiolo = new PizzaBuilder();

var margherita = pizzaiolo.CreatePizzaWithSize(Size.Medium)

.WithCrust(Crusts.Thin)

.WithSauce(Sauces.Tomato)

.AddTopping(Toppings.Mozzarella)

.AddTopping(Toppings.Basil)

.Bake();

This interface emphasizes the different parts that can be added to your *Product* while making it super easy to read in a business context. Each of the methods being chained return your *Builder* and at the end you get your *Product*, with .Bake() in this case.

If we have a look at the code inside the *Builder*, we can see easy methods that set the properties on the object being created. Also you can spot your half-baked product (see what I did there) inside the class, ready to be returned by your Build() method, the .Bake() in our example.

public class PizzaBuilder : IPizzaBuilderCrust, IPizzaBuilderSauce, IPizzaBuilderToppings

{

private readonly Pizza pizza;

public PizzaBuilder()

{

pizza = new Pizza();

}

public PizzaBuilder CreatePizzaWithSize(Size selectedSize)

{

pizza.Size = selectedSize;

return this;

}

...

public Pizza Bake()

{

return pizza;

}

}

The next level, from an implementation perspective, would be to create some interfaces that would force the user to follow a specific path to create an object, in case there are some prerequisites that need to be met to build a specific part of your *Product*. You wouldn't want to add toppings to your pizza before adding the sauce. So your .WithSauce() might return an interface IPizzaBuilderToppings. So the method would look like this:

public IPizzaBuilderToppings WithSauce(Sauces selectedSauce)

{

pizza.Sauce = selectedSauce;

return this;

}

The interface would define the method AddTopping(), that in turn would force you to either add more toppings or just .Bake() it. Your *PizzaBuilder* would then inherit from all these interfaces.

public interface IPizzaBuilderToppings

{

IPizzaBuilderToppings AddTopping(Toppings selectedTopping);

}

Give the *Fluent Builder* a try the next time you are writing some unit tests, and you will see that building that complex database client will become much easier, verbose and clearer to understand and read.

Disclaimer: No pineapple was added in the making of this post.

Source code can be found at <https://github.com/afmguedes/DesignPatterns>