Creational

Builder

Builder is a creational design pattern that lets you construct complex objects step by step. The pattern allows you to produce different types and representations of an object using the same construction code.

Solves:

Imagine a complex object that requires laborious, step-by-step initialization of many fields and nested objects. Such initialization code is usually buried inside a monstrous constructor with lots of parameters. Or even worse: scattered all over the client code.

Implement:

The Builder pattern suggests that you extract the object construction code out of its own class and move it to separate objects called *builders*.

The pattern organizes object construction into a set of steps (`buildWalls`, `buildDoor`, etc.). To create an object, you execute a series of these steps on a builder object. The important part is that you don't need to call all of the steps. You can call only those steps that are necessary for producing a particular configuration of an object.

Some of the construction steps might require different implementation when you need to build various representations of the product. For example, walls of a cabin may be built of wood, but the castle walls must be built with stone.

In this case, you can create several different builder classes that implement the same set of building steps, but in a different manner. Then you can use these builders in the construction process (i.e., an ordered set of calls to the building steps) to produce different kinds of objects.

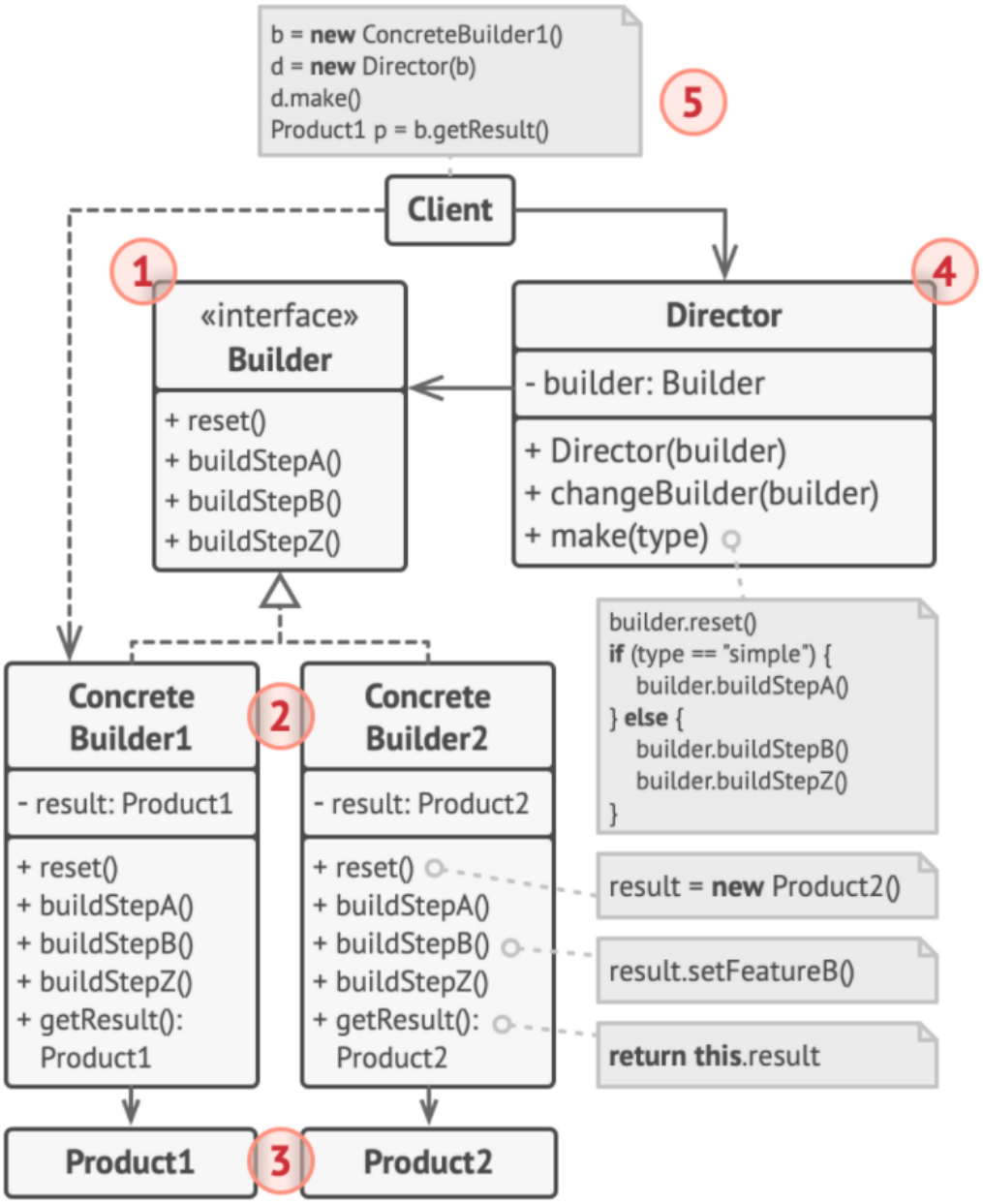
Director:

You can go further and extract a series of calls to the builder steps you use to construct a product into a separate class called *director*. The director class defines the order in which to execute the building steps, while the builder provides the implementation for those steps.

Having a director class in your program isn't strictly necessary. You can always call the building steps in a specific order directly from the client code. However, the director class might be a good place to put various construction routines so you can reuse them across your program.

In addition, the director class completely hides the details of product construction from the client code. The client only needs to associate a builder with a director, launch the construction with the director, and get the result from the builder.

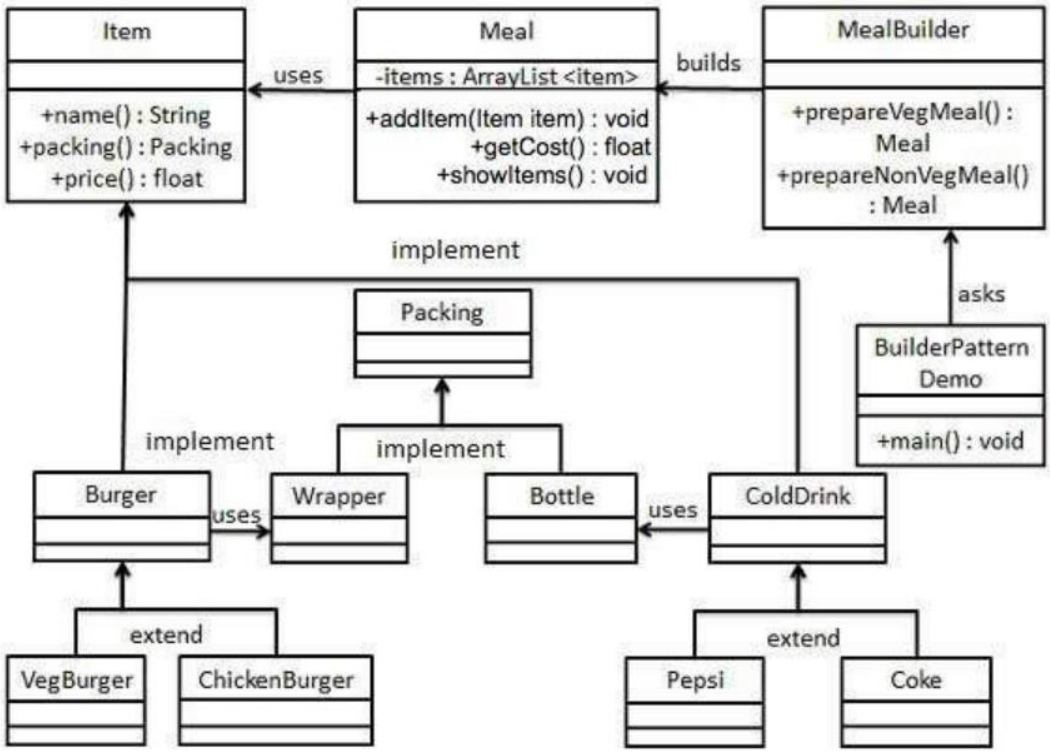
Structure



1. The **Builder** interface declares product construction steps that are common to all types of builders.
2. **Concrete Builders** provide different implementations of the construction steps. Concrete builders may produce products that don't follow the common interface.
3. **Products** are resulting objects. Products constructed by different builders don't have to belong to the same class hierarchy or interface.
4. The **Director** class defines the order in which to call construction steps, so you can create and reuse specific configurations of products.
5. The **Client** must associate one of the builder objects with the director. Usually, it's done just once, via parameters of the director's constructor. Then the director uses that builder object for all further construction. However, there's an alternative approach for when the client passes the builder object to

the production method of the director. In this case, you can use a different builder each time you produce something with the director.

Code Structure



Instances of Application

>Use the Builder pattern to get rid of a "telescopic constructor".

Say you have a constructor with ten optional parameters. Calling such a beast is very inconvenient; therefore, you overload the constructor and create several shorter versions with fewer parameters. These constructors still refer to the main one, passing some default values into any omitted parameters.

>Use the Builder pattern when you want your code to be able to create different representations of some product (for example, stone and wooden houses).

The Builder pattern can be applied when construction of various representations of the product involves similar steps that differ only in the details.The base builder interface defines all possible construction steps, and concrete builders implement these steps to construct particular representations of the product. Meanwhile, the director class guides the order of construction.

> Use the Builder to construct Composite trees or other complex objects.

The Builder pattern lets you construct products step-by-step. You could defer execution of some steps without breaking the final product. You can even call steps recursively, which comes in handy when you need to build an object tree.

How to Implement

1. Make sure that you can clearly define the common construction steps for building all available product representations. Otherwise, you won't be able to proceed with implementing the pattern.
2. Declare these steps in the base builder interface.
3. Create a concrete builder class for each of the product representations and implement their construction steps.

   Don't forget about implementing a method for fetching the result of the construction. The reason why this method can't be declared inside the builder interface is that various builders may construct products that don't have a common interface. Therefore, you don't know what would be the return type for such a method. However, if you're dealing with products from a single hierarchy, the fetching method can be safely added to the base interface.

4. Think about creating a director class. It may encapsulate various ways to construct a product using the same builder object.
5. The client code creates both the builder and the director objects. Before construction starts, the client must pass a builder object to the director. Usually, the client does this only once, via parameters of the director's constructor. The director uses the builder object in all further construction. There's an alternative approach, where the builder is passed directly to the construction method of the director.
6. The construction result can be obtained directly from the director only if all products follow the same interface. Otherwise, the client should fetch the result from the builder.

Pros and Cons

Pros:

- You can construct objects step-by-step, defer construction steps or run steps recursively.
- You can reuse the same construction code when building various representations of products.
- *Single Responsibility Principle*. You can isolate complex construction code from the business logic of the product.

Cons:

- The overall complexity of the code increases since the pattern requires creating multiple new classes.

Relations with other pattern

- Many designs start by using **Factory Method** (less complicated and more customizable via subclasses) and evolve toward **Abstract Factory**, **Prototype**, or **Builder** (more flexible, but more complicated).
- **Builder** focuses on constructing complex objects step by step. **Abstract Factory** specializes in creating families of related objects. *Abstract Factory* returns the product immediately, whereas *Builder* lets you run some additional construction steps before fetching the product.
- You can use **Builder** when creating complex **Composite** trees because you can program its construction steps to work recursively.
- You can combine **Builder** with **Bridge**: the director class plays the role of the abstraction, while different builders act as implementations.
- **Abstract Factories**, **Builders** and **Prototypes** can all be implemented as **Singletons**.