

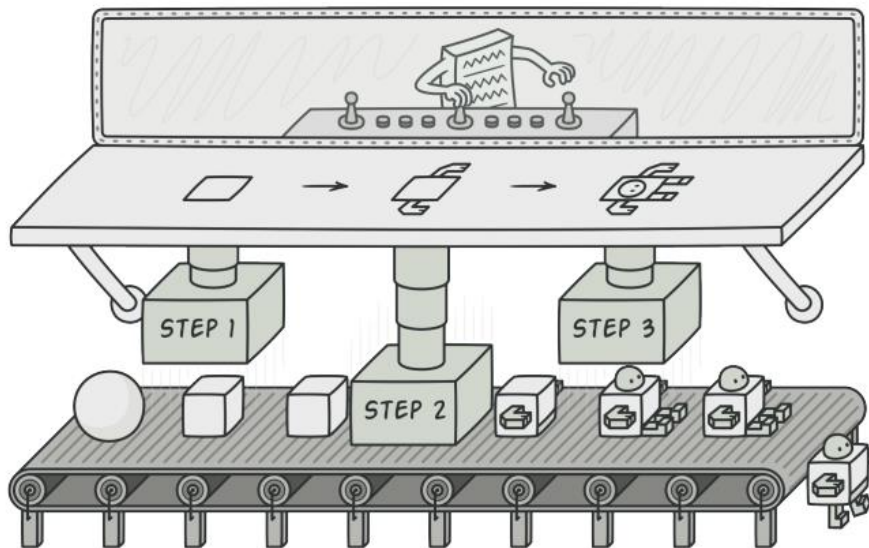
# Design Patterns(SE-408)

Course Teacher

Assistant Professor

Engr. Mustafa Latif

1



**BUILDER PATTERN**

2

# Builder Pattern

- ***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.*

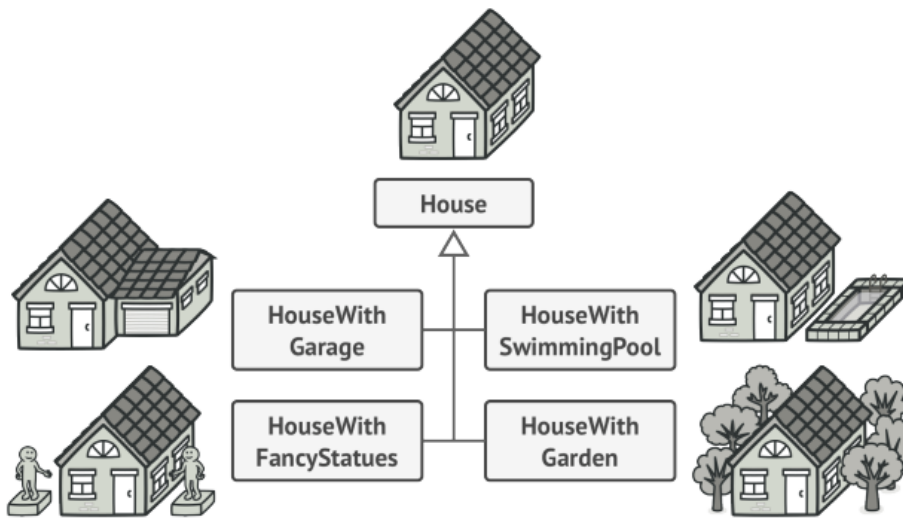
3

## Problem

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.

For example, let's think about how to create a House object. To build a simple house, you need to construct four walls and a floor, install a door, fit a pair of windows, and build a roof. But what if you want a bigger, brighter house, with a backyard and other goodies (like a heating system, plumbing, and electrical wiring)?

4



*You might make the program too complex by creating a subclass for every possible configuration of an object.*

5

## Problem

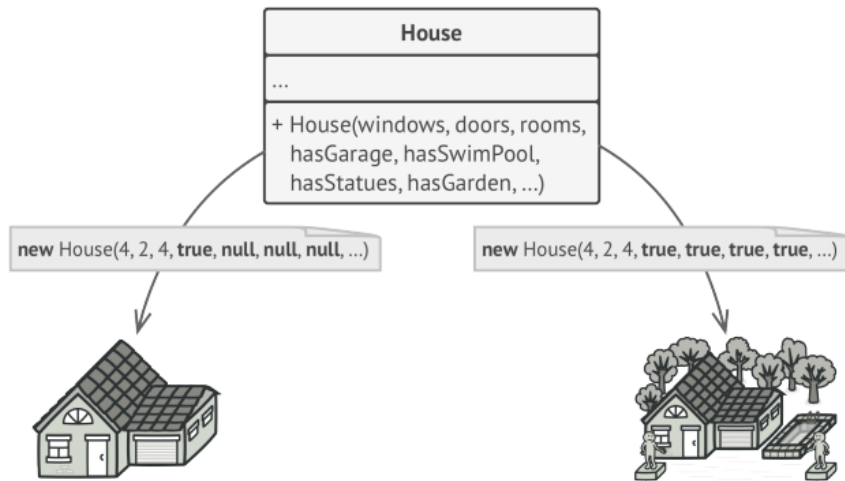
The simplest solution is to extend the base House class and create a set of subclasses to cover all combinations of the parameters. But eventually you'll end up with a considerable number of subclasses. Any new parameter, such as the porch style, will require growing this hierarchy even more.

There's another approach that doesn't involve breeding subclasses. You can create a giant constructor right in the base House class with all possible parameters that control the house object. While this approach indeed eliminates the need for subclasses, it creates another problem.

In most cases most of the parameters will be unused, making the constructor calls pretty ugly. For instance, only a fraction of houses have swimming pools, so the parameters related to swimming pools will be useless nine times out of ten.

6

## Problem



*The constructor with lots of parameters has its downside: not all the parameters are needed at all times.*

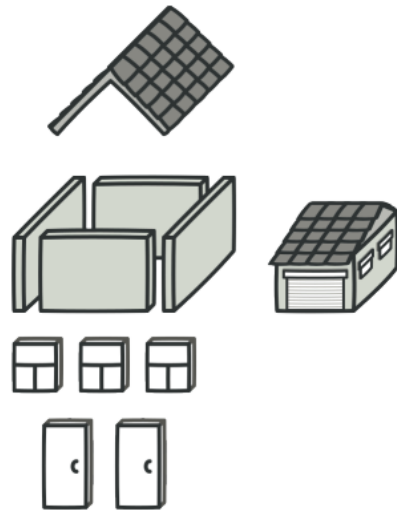
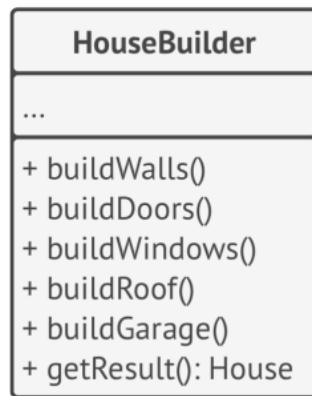
7

## Solution

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.

8



*The Builder pattern lets you construct complex objects step by step. The Builder doesn't allow other objects to access the product while it's being built.*

9

## Solution

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.

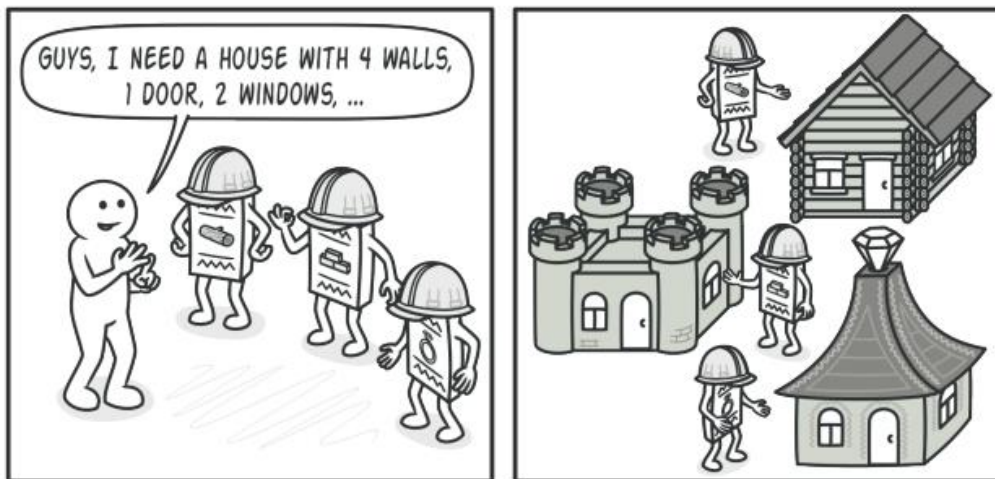
In this case, you can create several different builder classes that implement the same set of building steps, but in a different manner. Then you can use these builders in the construction process (i.e., an ordered set of calls to the building steps) to produce different kinds of objects.

10

## Solution

For example, imagine a builder that builds everything from wood and glass, a second one that builds everything with stone and iron and a third one that uses gold and diamonds. By calling the same set of steps, you get a regular house from the first builder, a small castle from the second and a palace from the third. However, this would only work if the client code that calls the building steps is able to interact with builders [using a common interface](#).

11



*Different builders execute the same task in various ways.*

12

## Solution

### 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.

13

## Solution

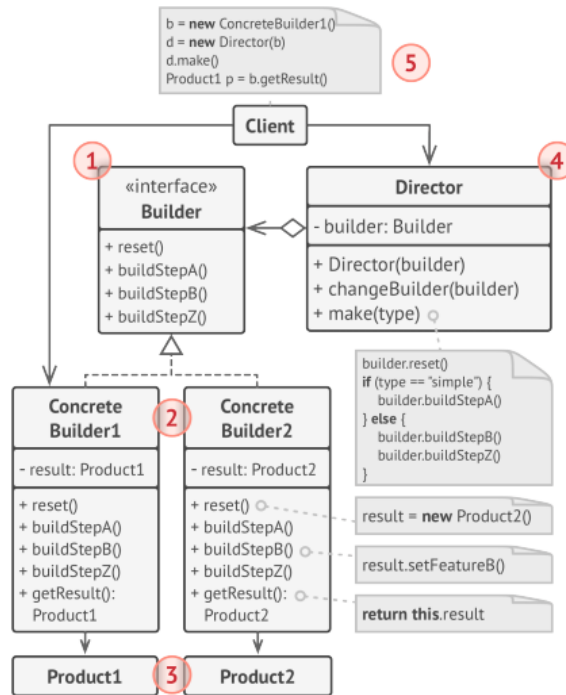
In addition, the director class completely hides the details of product construction from the client code. The client only needs to associate a builder with a director, launch the construction with the director, and get the result from the builder.



*The director knows which building steps to execute to get a working product.*

14

# Structure




15

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.

16



- 
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.

17

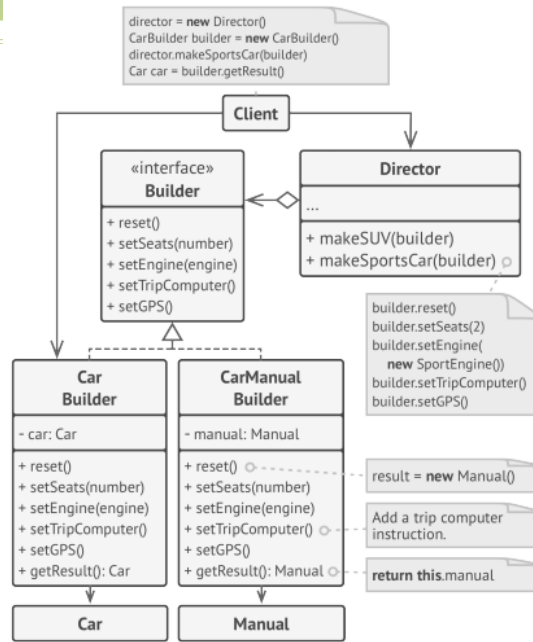


## Pseudocode

This example of the Builder pattern illustrates how you can reuse the same object construction code when building different types of products, such as cars, and create the corresponding manuals for them.

18

# Pseudocode




The example of step-by-step construction of cars and the user guides that fit those car models.

19

```


1 // Using the Builder pattern makes sense only when your products
2 // are quite complex and require extensive configuration. The
3 // following two products are related, although they don't have
4 // a common interface.
5 class Car is
6 // A car can have a GPS, trip computer and some number of
7 // seats. Different models of cars (sports car, SUV,
8 // cabriolet) might have different features installed or
9 // enabled.
10
11 class Manual is
12 // Each car should have a user manual that corresponds to
13 // the car's configuration and describes all its features.
14
15
    
```

20




```
16 // The builder interface specifies methods for creating the
17 // different parts of the product objects.
18 interface Builder is
19 method reset()
20 method setSeats(...)
21 method setEngine(...)
22 method setTripComputer(...)
23 method setGPS(...)
24
25 // The concrete builder classes follow the builder interface and
26 // provide specific implementations of the building steps. Your
27 // program may have several variations of builders, each
28 // implemented differently.
29 class CarBuilder implements Builder is
30 private field car:Car
31
32
```

21




```
33 // A fresh builder instance should contain a blank product
34 // object which it uses in further assembly.
35 constructor CarBuilder() is
36 this.reset()
37
38 // The reset method clears the object being built.
39 method reset() is
40 this.car = new Car()
41
42 // All production steps work with the same product instance.
43 method setSeats(...) is
44 // Set the number of seats in the car.
45
46 method setEngine(...) is
47 // Install a given engine.
```

22




```
48
49 method setTripComputer(...) is
50 // Install a trip computer.
51
52 method setGPS(...) is
53 // Install a global positioning system.
54
55 // Concrete builders are supposed to provide their own
56 // methods for retrieving results. That's because various
57 // types of builders may create entirely different products
58 // that don't all follow the same interface. Therefore such
59 // methods can't be declared in the builder interface (at
60 // least not in a statically-typed programming language).
61 //
62 // Usually, after returning the end result to the client, a
```

23




```
63 // builder instance is expected to be ready to start
64 // producing another product. That's why it's a usual
65 // practice to call the reset method at the end of the
66 // `getProduct` method body. However, this behavior isn't
67 // mandatory, and you can make your builder wait for an
68 // explicit reset call from the client code before disposing
69 // of the previous result.
70 method getProduct():Car is
71 product = this.car
72 this.reset()
73 return product
74
```

24




```
75 // Unlike other creational patterns, builder lets you construct
76 // unrelated products that don't follow a common interface.
77 class CarManualBuilder implements Builder is
78 private field manual:Manual
79
80 constructor CarManualBuilder() is
81 this.reset()
82
83 method reset() is
84 this.manual = new Manual()
85
86 method setSeats(...) is
87 // Document car seat features.
88
```

25




```
89 method setEngine(...) is
90 // Add engine instructions.
91
92 method setTripComputer(...) is
93 // Add trip computer instructions.
94
95 method setGPS(...) is
96 // Add GPS instructions.
97 method getProduct():Manual is
98 // Return the manual and reset the builder.
99
100
```

26



```
101 // The director is only responsible for executing the building
102 // steps in a particular sequence. It's helpful when producing
103 // products according to a specific order or configuration.
104 // Strictly speaking, the director class is optional, since the
105 // client can control builders directly.
106 class Director is
107   private field builder:Builder
108
109 // The director works with any builder instance that the
110 // client code passes to it. This way, the client code may
111 // alter the final type of the newly assembled product.
112 method setBuilder(builder:Builder)
113   this.builder = builder
114
```

27



```
115 // The director can construct several product variations
116 // using the same building steps.
117 method constructSportsCar(builder: Builder) is
118   builder.reset()
119   builder.setSeats(2)
120   builder.setEngine(new SportEngine())
121   builder.setTripComputer(true)
122   builder.setGPS(true)
123
124 method constructSUV(builder: Builder) is
125   // ...
126
127
128
```

28

```
129 // The client code creates a builder object, passes it to the
130 // director and then initiates the construction process. The end
131 // result is retrieved from the builder object.
132 class Application is
133
134 method makeCar() is
135   director = new Director()
136
137   CarBuilder builder = new CarBuilder()
138   director.constructSportsCar(builder)
139   Car car = builder.getProduct()
140
141   CarManualBuilder builder = new CarManualBuilder()
142   director.constructSportsCar(builder)
143
144 // The final product is often retrieved from a builder
145 // object since the director isn't aware of and not
146 // dependent on concrete builders and products.
147   Manual manual = builder.getProduct()
```

29

## Java Code Example

30

# Applicability

- Use the Builder pattern to get rid of a “telescopic constructor”

```
1  class Pizza {  
2      Pizza(int size) { ... }  
3      Pizza(int size, boolean cheese) { ... }  
4      Pizza(int size, boolean cheese, boolean pepperoni) { ... }  
5      // ...
```

- 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).

31

# 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.

32



## How to Implement

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.

33

## Pros and Cons

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

34



# Review