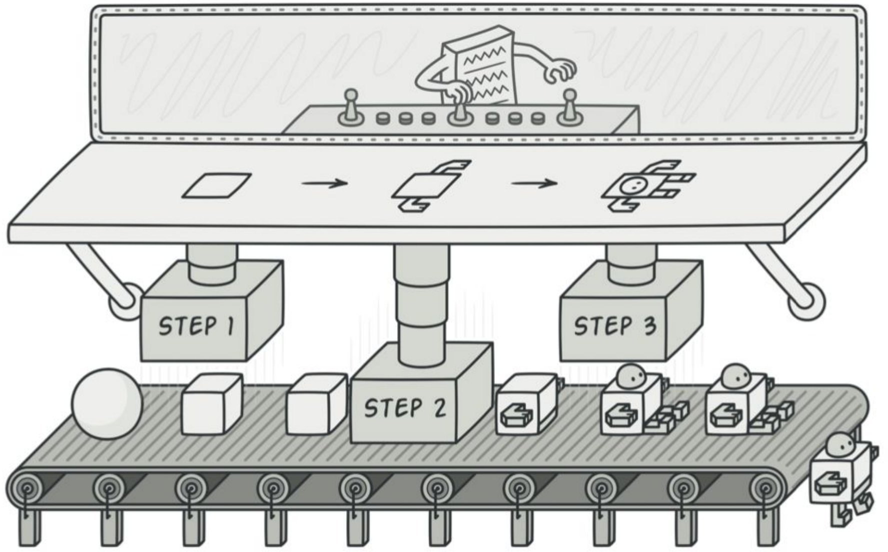
Lecture 5: Design Patterns Builder

•••

Builder Design 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.

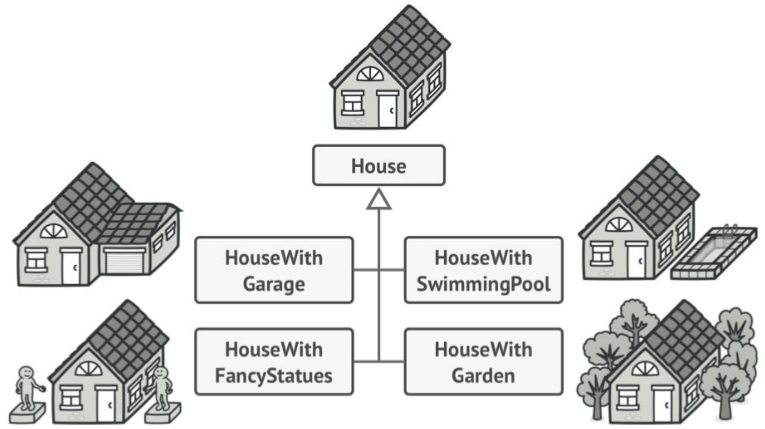


Builder: Problem

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

Builder: Problem

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)?



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

Builder: 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.

Builder: Problem

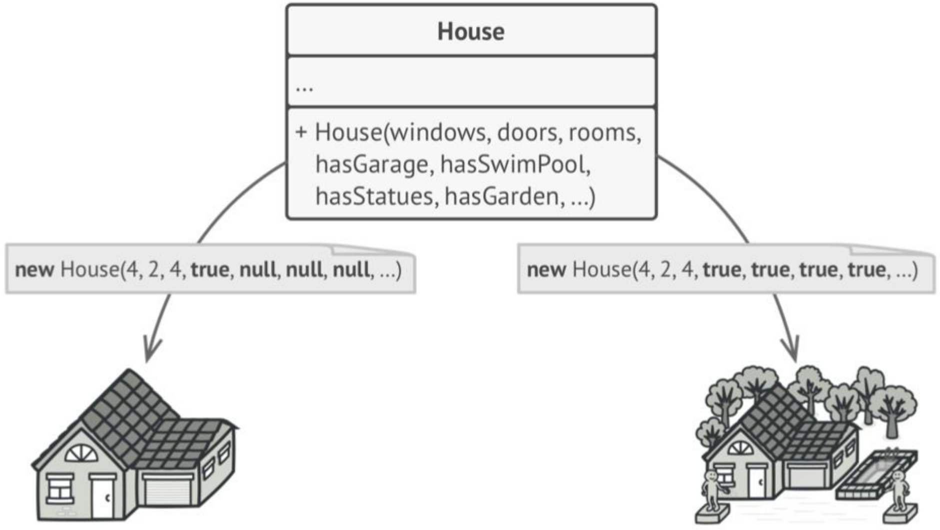
There’s another approach that doesn’t involve breeding sub- classes.

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.

Builder Problem

In most cases most of the parameters w
be unused, making

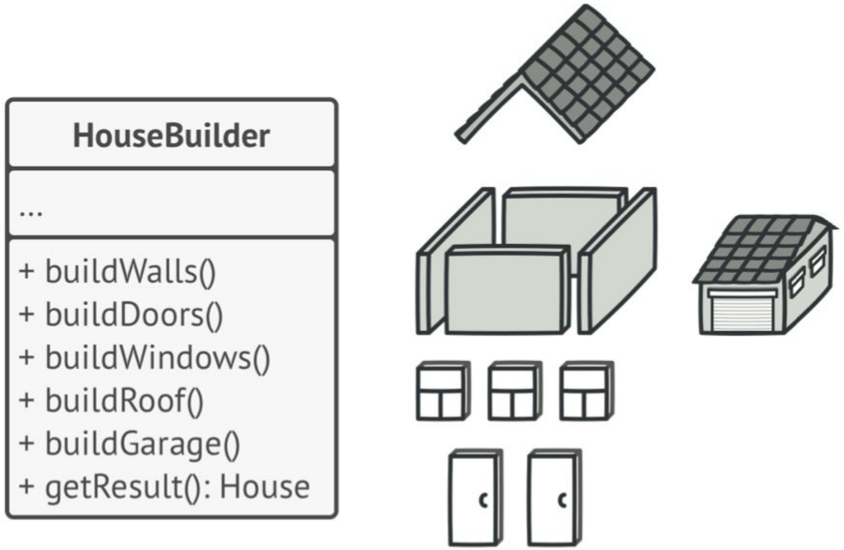
the constructor calls pretty ugly. For instance, only a fraction of houses have swimming pools, so the parameters related to swimming pools will be usele
nine times out of ten.


House
...
+ House(windows, doors, rooms, hasGarage, hasSwimPool, hasStatues, hasGarden, ...)
new House(4, 2, 4, true, null, null, null, ...)
new House(4, 2, 4, true, true, true, true, ...)

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

Builder: Solution

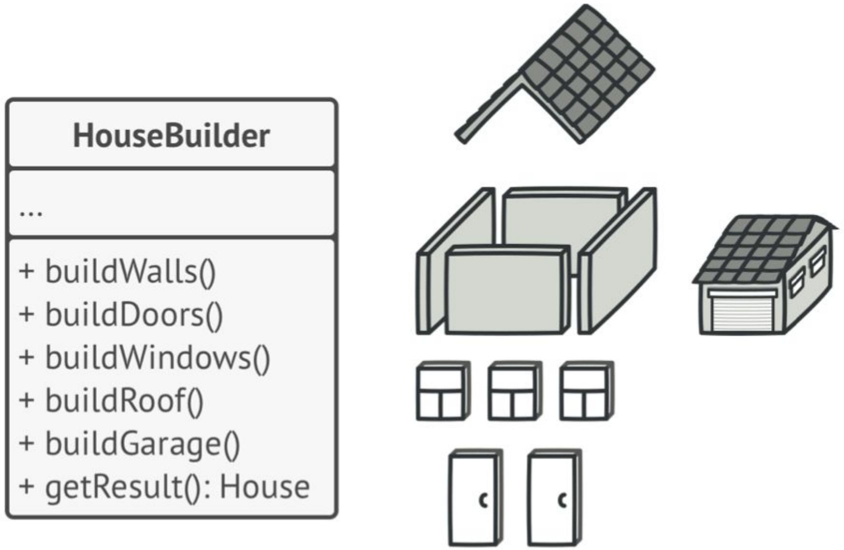
The Builder pattern suggests that you extract the object con- struction code out of its own class and move it to separate objects called builders.



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.

Builder: Solution

The pattern organizes object construction into a set of steps (buildWalls , buildDoor , etc.). To create an object, you exe- cute 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.

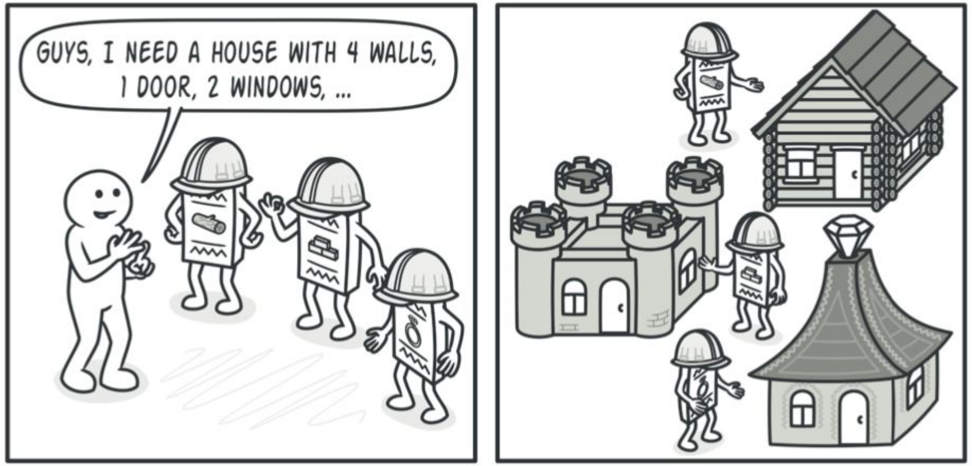


The Builder pattern lets you construct complex objects step by step. The Builder doesn't allow other objects to access the product while it's being built.

Builder Solution

Some of the construction steps might require different imple- mentation 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 differ- ent manner. Then you can use these builders in the construc- tion process (i.e., an ordered set of calls to the buildin steps) to produce different kinds of objects.



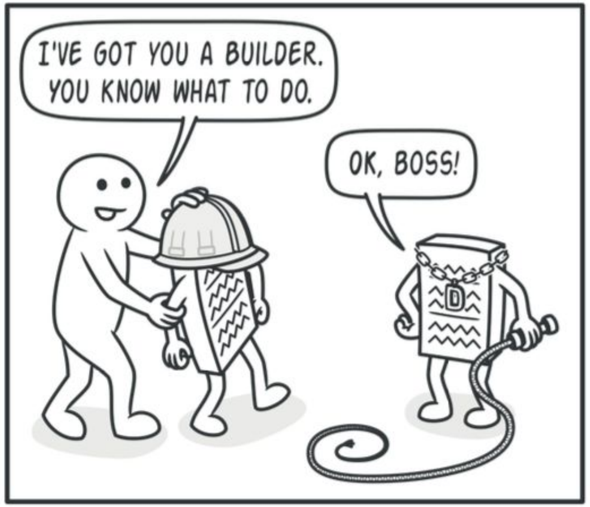
Different builders execute the same task in various ways.

Builder: Solution

Director

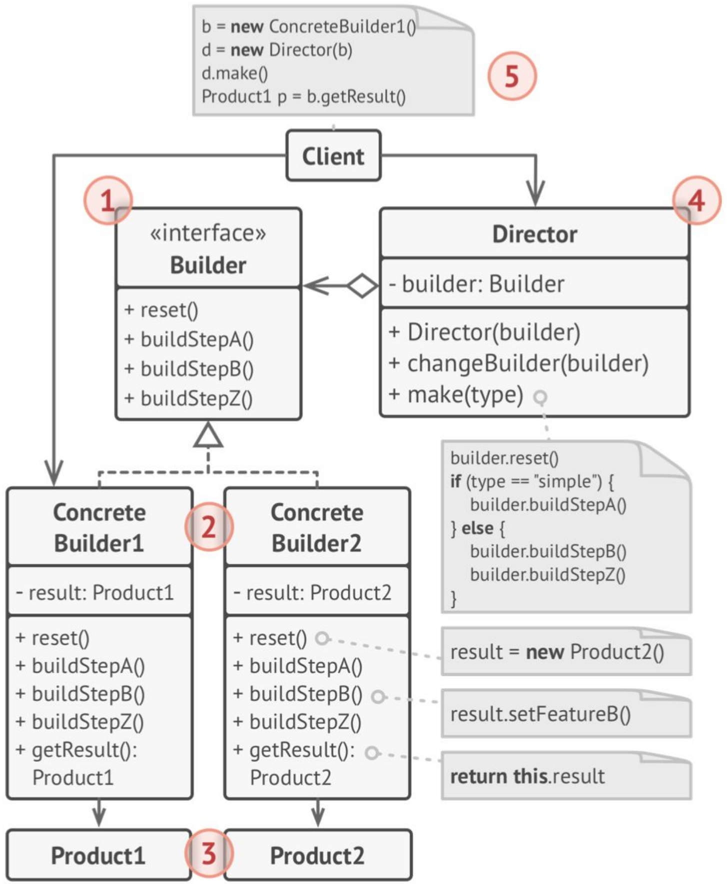
You can go further and extract a series of calls to the builder steps you use to construct a product into a separate class called director. The director class defines the order in which to execute the building steps, while the builder provides the implementation for those steps.

Not necessary, sometimes overkill



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

Builder: Structure



```
b = new ConcreteBuilder1()
d = new Director(b)
d.make()
Product1 p = b.getResult()
```

```
builder.reset()
if (type == "simple") {
    builder.buildStepA()
} else {
    builder.buildStepB()
    builder.buildStepZ()
}
```

```
result = new Product2()
```

```
result.setFeatureB()
```

```
return this.result
```

Builder: Parts

1. The Builder interface declares product construction steps that are common to all types of builders.

Builder: Parts

2. Concrete Builders provide different implementations of the construction steps. Concrete builders may produce products that don't follow the common interface.

Builder: Parts

3. Products are resulting objects. Products constructed by differ- ent builders don’t have to belong to the same class hierarchy or interface.

Builder: Parts

4. The Director class defines the order in which to call construc- tion steps, so you can create and reuse specific configurations of products.

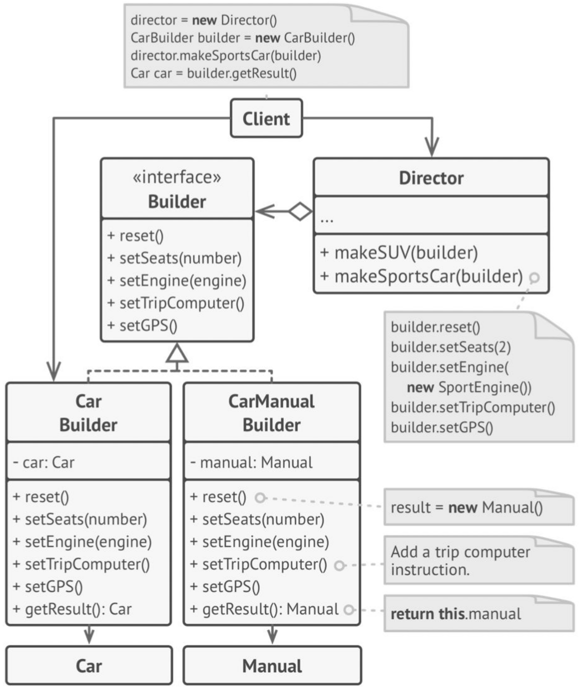
Builder: Parts

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 alterna- tive 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.

Builder: Concrete Example

We want to create a CarBuilder.

Builder : Concrete Example



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

When to Use Builder Pattern

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

Say you have a constructor with ten optional parameters. Call- ing such a beast is very inconvenient; therefore, you over- load 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.

When to use the builder pattern

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 vari- ous representations of the product involves similar steps that differ only in the details.

A real-world example of this is varied machine learning pipelines!

Builder: Pros

You can construct objects step-by-step, defer construction

steps or run steps recursively.

You can reuse the same construction code when building vari- ous representations of products.

Single Responsibility Principle. You can isolate complex con- struction code from the business logic of the product.

Builder: Cons

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

Builder: In-class exercise TODO make better

You find some code for a PizzaMaker that looks like this:

```
class Pizza:
    def __init__(size: int, cheese: bool, pepperoni: bool, chile: bool, basil: bool,
extra_sauce: bool, )
```

Change this into a builder pattern!