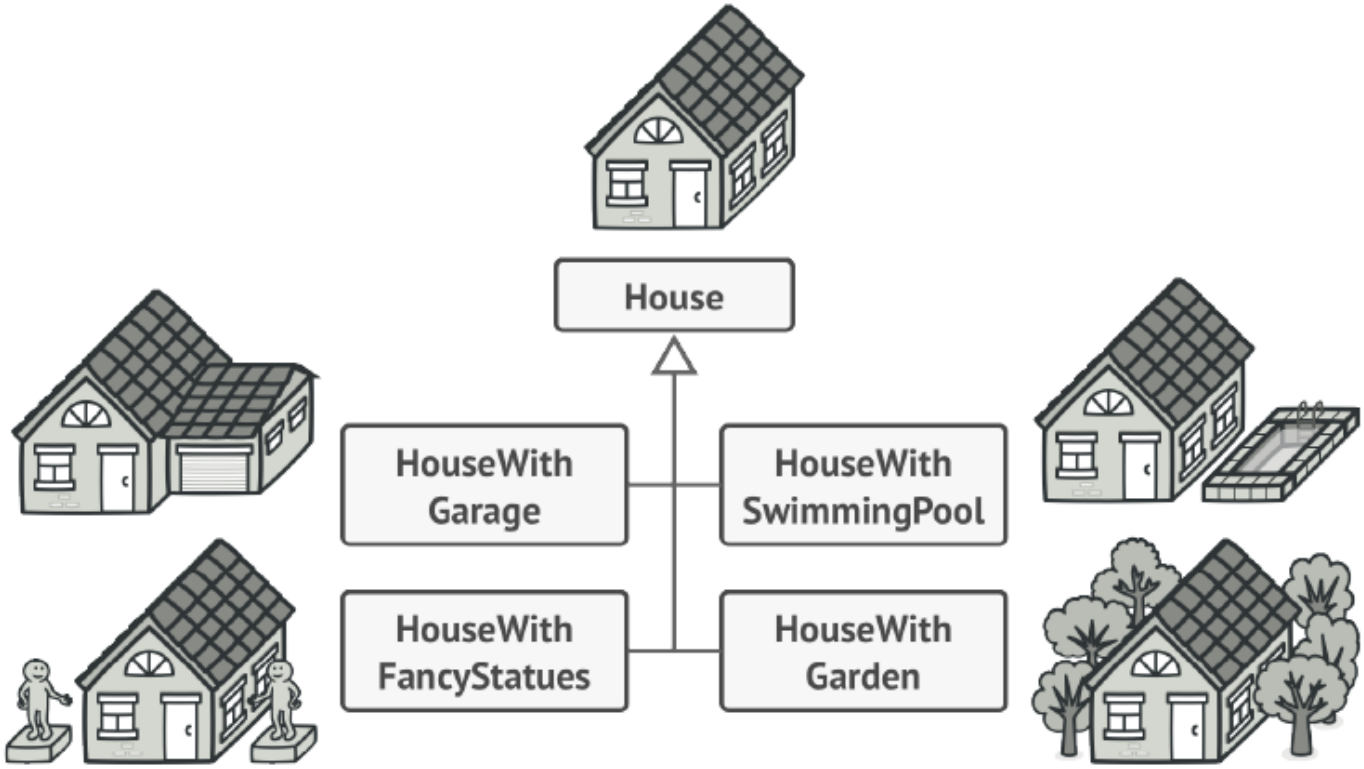
Builder

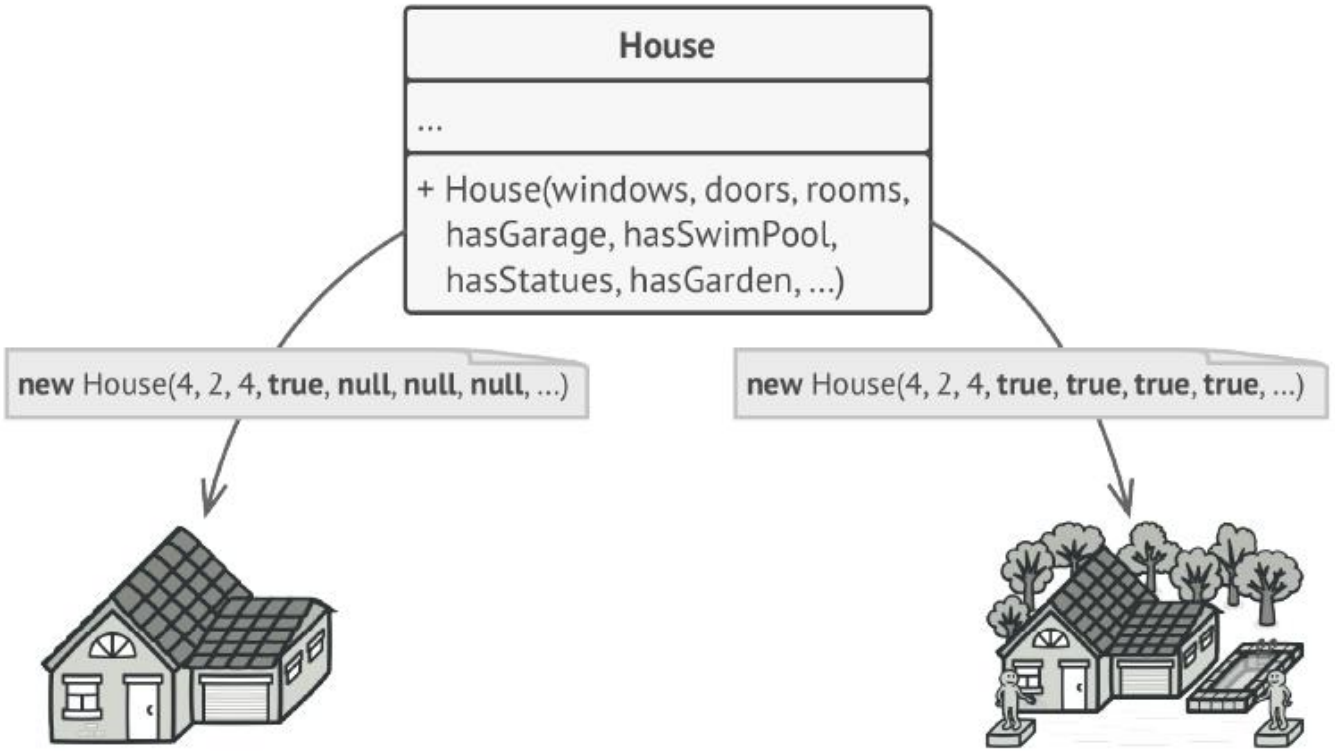
- **Builder** is a creational design pattern that lets you construct complex objects step by step. The pattern allows you to produce different types and representations of an object using the same construction code.

Problem



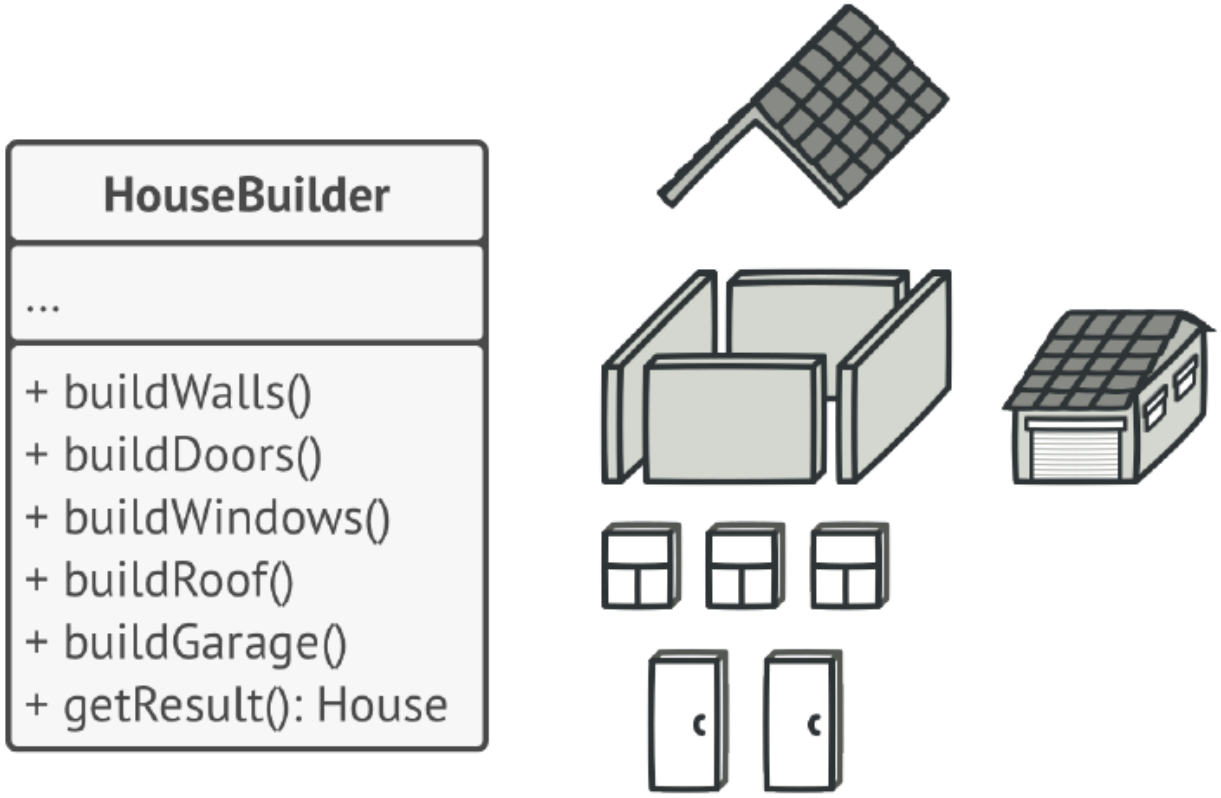
You might make the program too complex by creating a subclass for every possible configuration of an object.

Contd.



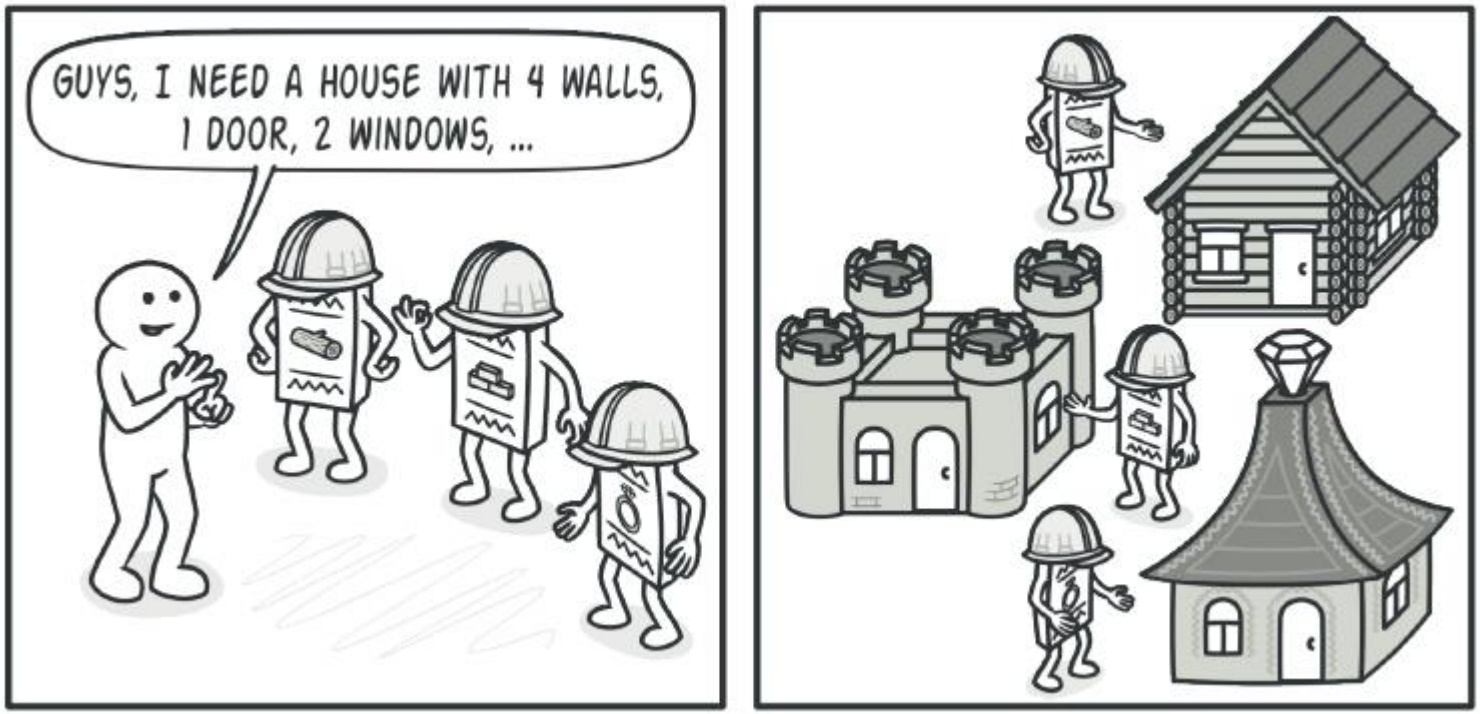
The constructor with lots of parameters has its downside: not all the parameters are needed at all times.

Solution



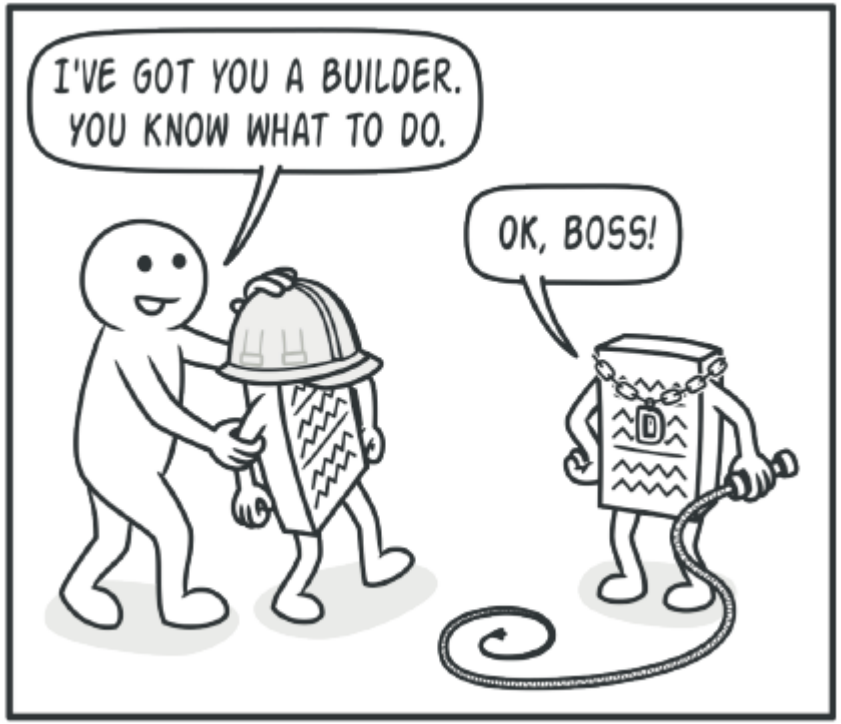
The Builder pattern lets you construct complex objects step by step. The Builder doesn't allow other objects to access the product while it's being built.

Contd.



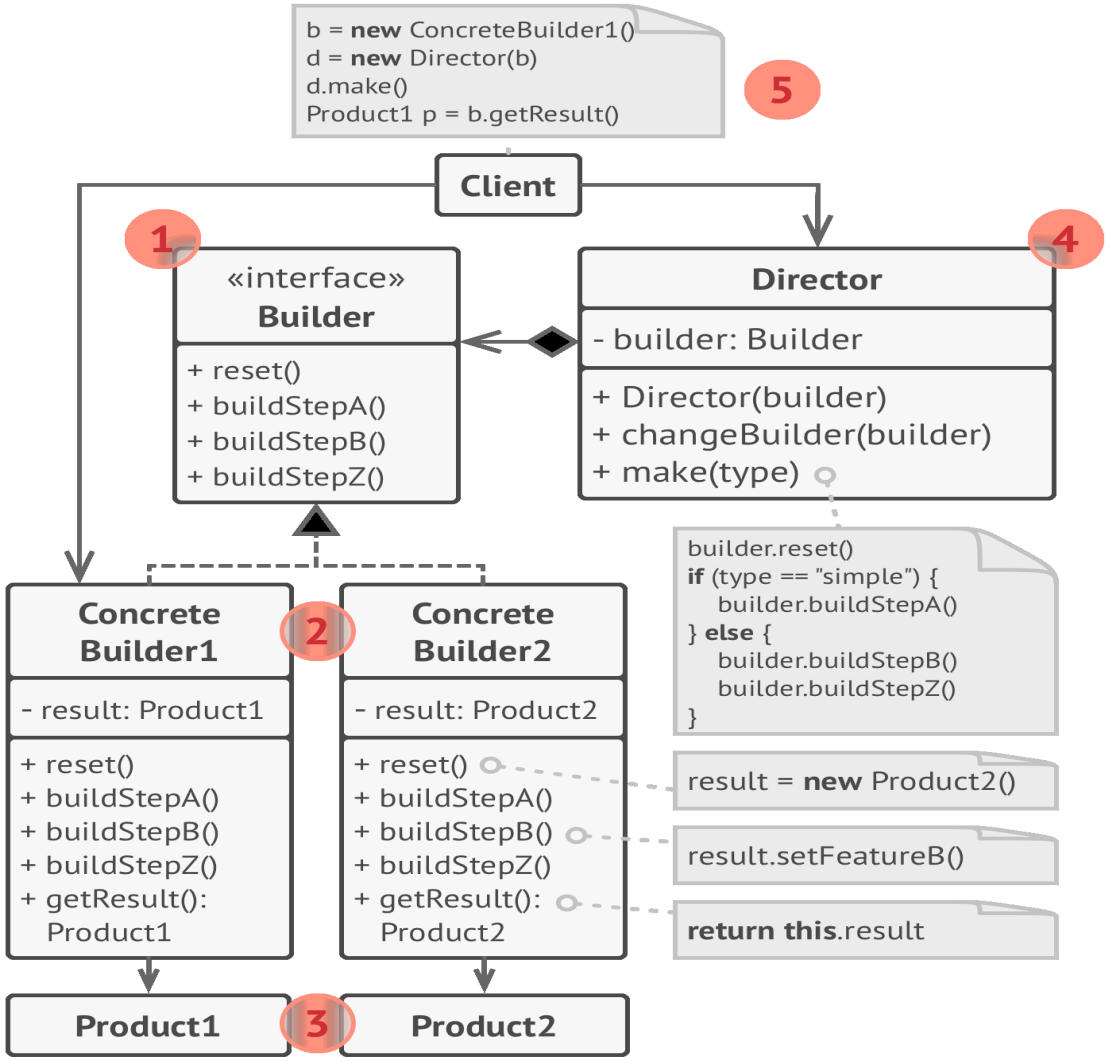
Different builders execute the same task in various ways.

Contd.



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

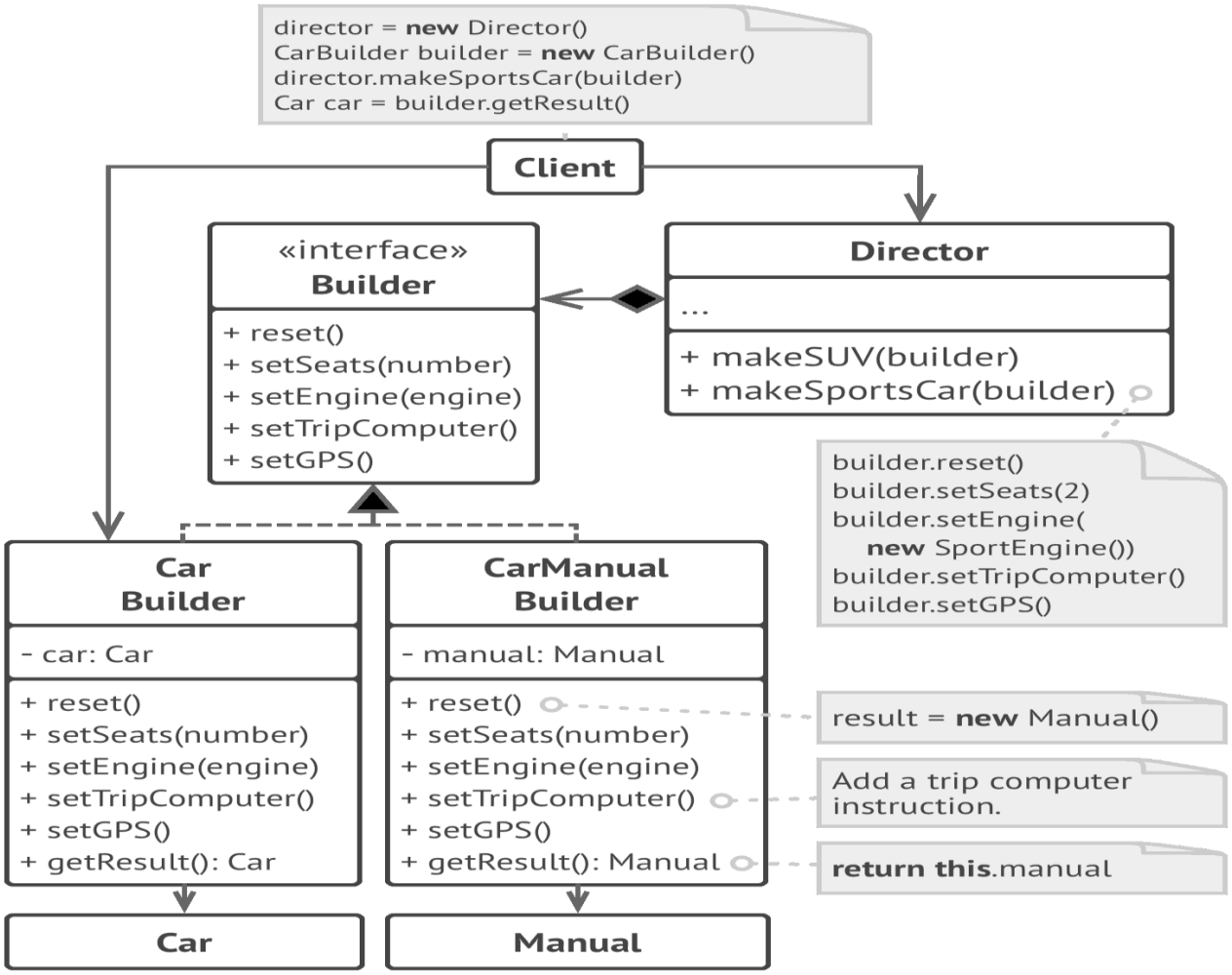
Structure


b = new ConcreteBuilder1()
d = new Director(b)
d.make()
Product1 p = b.getResult()
5
Client
1
«interface»
Builder
+ reset()
+ buildStepA()
+ buildStepB()
+ buildStepZ()
4
Director
- builder: Builder
+ Director(builder)
+ changeBuilder(builder)
+ make(type)
builder.reset()
if (type == "simple") {
builder.buildStepA()
} else {
builder.buildStepB()
builder.buildStepZ()
}
Concrete
Builder1
2
- result: Product1
+ reset()
+ buildStepA()
+ buildStepB()
+ buildStepZ()
+ getResult():
Product1
Concrete
Builder2
- result: Product2
+ reset()
+ buildStepA()
+ buildStepB()
+ buildStepZ()
+ getResult():
Product2
result = new Product2()
result.setFeatureB()
return this.result
Product1
3
Product2

Contd.

- The **Builder** interface declares product construction steps that are common to all types of builders.
- **Concrete Builders** provide different implementations of the construction steps. Concrete builders may produce products that don’t follow the common interface.
- **Products** are resulting objects. Products constructed by different builders don’t have to belong to the same class hierarchy or interface.
- The **Director** class defines the order in which to call construction steps, so you can create and reuse specific configurations of products.
- 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.

Implementation



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

Applicability

- 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.

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

Creating such a monster is only possible in languages that support method overloading, such as C# or Java.

- The Builder pattern lets you build objects step by step, using only those steps that you really need. After implementing the pattern, you don’t have to cram dozens of parameters into your constructors anymore.

Contd.

- 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.uses).
- 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. A builder doesn’t expose the unfinished product while running construction steps. This prevents the client code from fetching an incomplete result.

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.