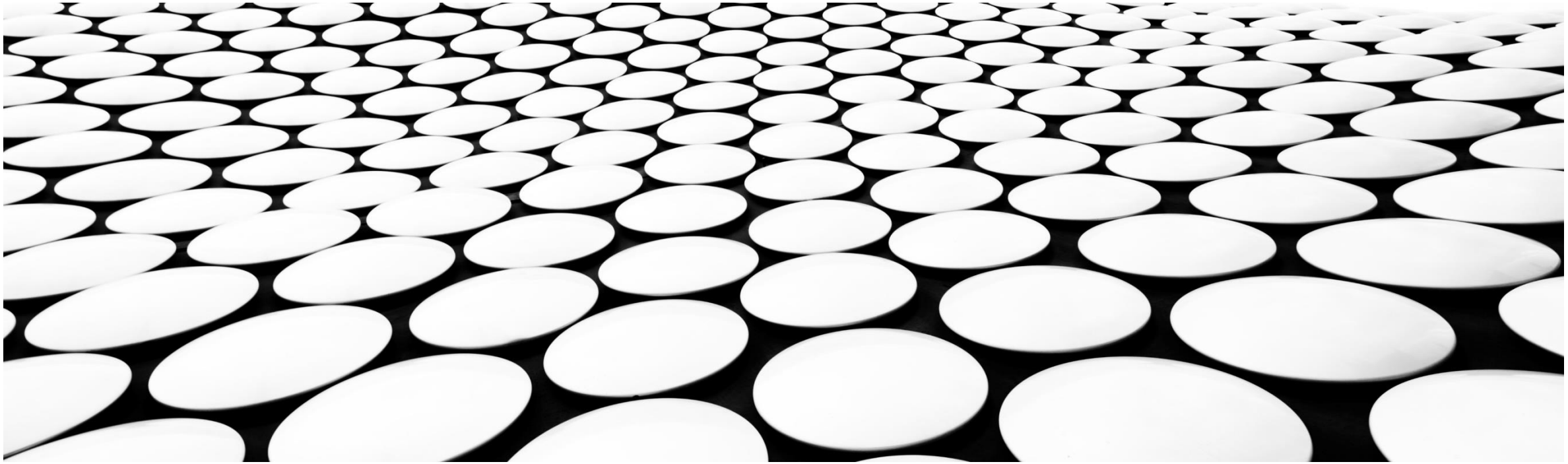
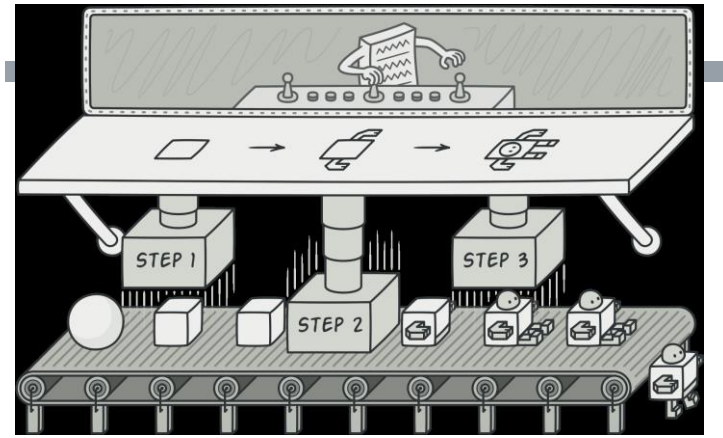
SOFTWARE DESIGN PATTERN (CREATIONAL PATTERN)

LEC2: BUILDER.



BUILDER

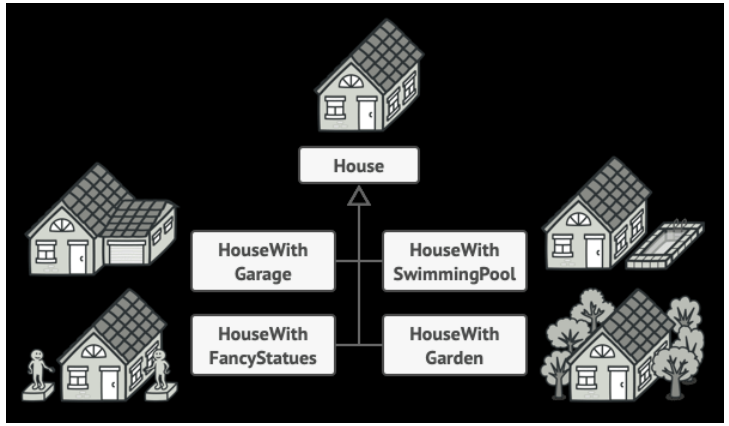


- The **builder pattern** is a type of creational pattern that helps in building complex objects using simpler objects. It provides a flexible and step-by-step approach towards making these objects and keeps the representation, and the process of creation shielded.

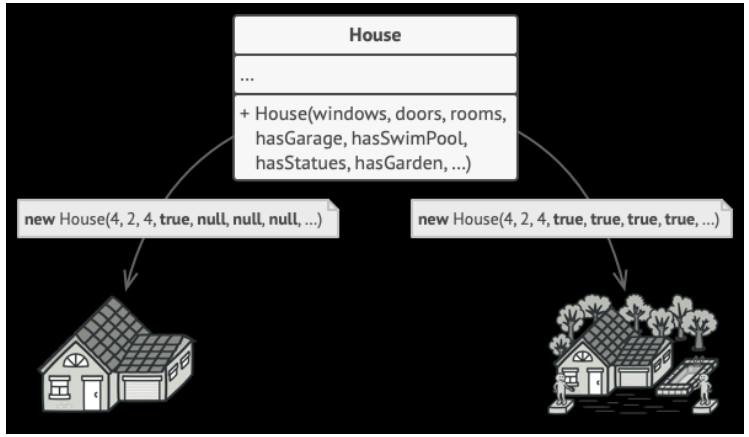
- As mentioned in the Gang of four “Separate the construction of a complex object from its representation so that the same construction process can create different representations.”

THE CONCEPT

- ☹ 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)?
- 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.



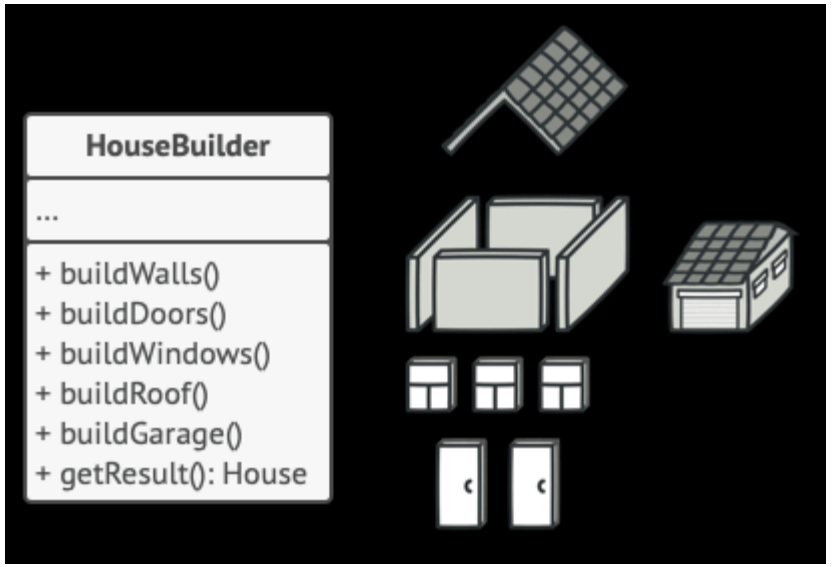
- There's another approach that doesn't involve breeding subclasses. You can create a giant constructor right in the base House class with all possible parameters that control the house object. While this approach indeed eliminates the need for subclasses, it creates another problem.



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

☺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 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.*



WHEN TO USE BUILDER PATTERN

- Builder pattern was introduced to solve some of the problems with Factory and Abstract Factory design patterns when the Object contains a lot of attributes.
- There will be issues with Factory and Abstract Factory design patterns when the Object contains a lot of attributes.
  - Some of the parameters might be optional but in Factory pattern, we are forced to send all the parameters and optional parameters need to send as NULL.
  - If the object is heavy and its creation is complex, then all that complexity will be part of Factory classes that is confusing.
- We can solve the issues with large number of parameters by providing a constructor with required parameters and then different setter methods to set the optional parameters. The problem with this approach is that the Object state will be **inconsistent** until unless all the attributes are set explicitly.
- Builder pattern solves the issue with large number of optional parameters and inconsistent state by providing a way to build the object step-by-step and provide a method that will actually return the final Object.

STRUCTURE & PARTICIPANTS

Builder

specifies an abstract interface for creating parts of a Product object.

ConcreteBuilder

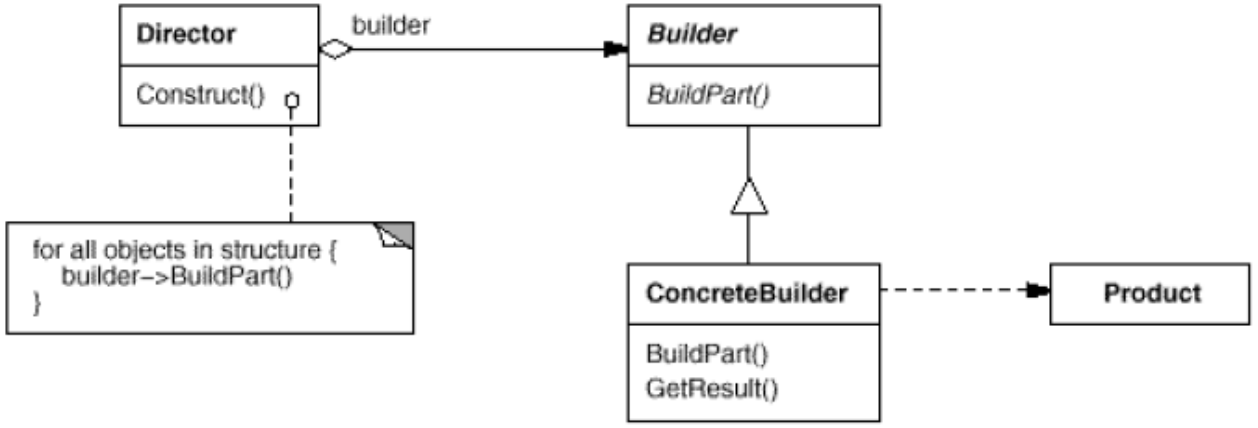
constructs and assembles parts of the product by implementing the Builder interface.
provides an interface for retrieving the product.

Director (Executor)

constructs an object using the Builder interface.

Product

represents the complex object under construction.



WORK FLOW:

- The client creates the director object and configures it with the desired builder object.
- Director notifies builder whenever a part of product should be built.
- Builder handles requests from the director and adds parts to the product.
- The clients retrieves the product from builder.

EXAMPLE:

- In this example, the participants are **IBuilder**, **Car**, **MotorCycle**, **Product**, and **Director**. Car and MotorCycle are implementing the IBuilder interface. IBuilder is used to create parts of the Product object where Product represents the complex object under construction.
- The assembly process is described in Product. We can see that we have used the Linked List data structure in Product for this assembly operation.
- Car and MotorCycle are the concrete implementations. They have implemented IBuilder interface. BuildBody(), InsertWheels(), and AddHeadlights() are used to build the body of the vehicle, insert the number of wheels into it, and add headlights to the vehicle. GetVehicle() will return the ultimate product.
- Finally, Director will be responsible for constructing the ultimate vehicle using the IBuilder interface. Director is calling the same Construct() method to create different types of vehicles.

PRODUCT

```
class Product {

    private LinkedList<String> parts;

    public Product() {
        parts = new LinkedList<String>();
    }

    public void Add(String part) {
        //Adding parts
        parts.addLast(part);
    }

    public void Show() {
        System.out.println("\n Product completed as below :");
        for (int i = 0; i < parts.size(); i++) {
            System.out.println(parts.get(i));
        }
    }
}
```

IBUILDER

```
// Builders common interface
interface IBuilder
{
        void BuildBody();
        void InsertWheels();
        void AddHeadlights();
        Product GetVehicle();

}
```

CAR & MOTORCYCLE

```
// Car is ConcreteBuilder
class Car implements IBuilder {

    private Product product = new Product();

    @Override
    public void BuildBody() {
        product.Add("This is a body of a Car");
    }

    @Override
    public void InsertWheels() {
        product.Add("4 wheels are added");
    }

    @Override
    public void AddHeadlights() {
        product.Add("2 Headlights are added");
    }

    @Override
    public Product GetVehicle() {
        return product;
    }
}
```

```
// Motorcycle is a ConcreteBuilder
class MotorCycle implements IBuilder {

    private Product product = new Product();

    @Override
    public void BuildBody() {
        product.Add("This is a body of a Motorcycle");
    }

    @Override
    public void InsertWheels() {
        product.Add("2 wheels are added");
    }

    @Override
    public void AddHeadlights() {
        product.Add("1 Headlights are added");
    }

    @Override
    public Product GetVehicle() {
        return product;
    }
}
```

DIRECTOR

```
class Director {

    IBuilder myBuilder;
    // A series of steps—for the production

    public void Construct(IBuilder builder) {
        myBuilder = builder;
        myBuilder.BuildBody();
        myBuilder.InsertWheels();
        myBuilder.AddHeadlights();

    }
}
```

EXAMPLE:

```
1
2   class BuilderPatternEx {
3
4       public static void main(String[] args) {
5           System.out.println("***Builder Pattern Demo***\n");
6
7           Director director = new Director();
8
9           IBuilder carBuilder = new Car();
10          IBuilder motorBuilder = new MotorCycle();
11          // Making Car
12          director.Construct(carBuilder);
13          Product p1 = carBuilder.GetVehicle();
14          p1.Show();
15
16          //Making MotorCycle
17          director.Construct(motorBuilder);
18          Product p2 = motorBuilder.GetVehicle();
19          p2.Show();
20      }
21  }
```

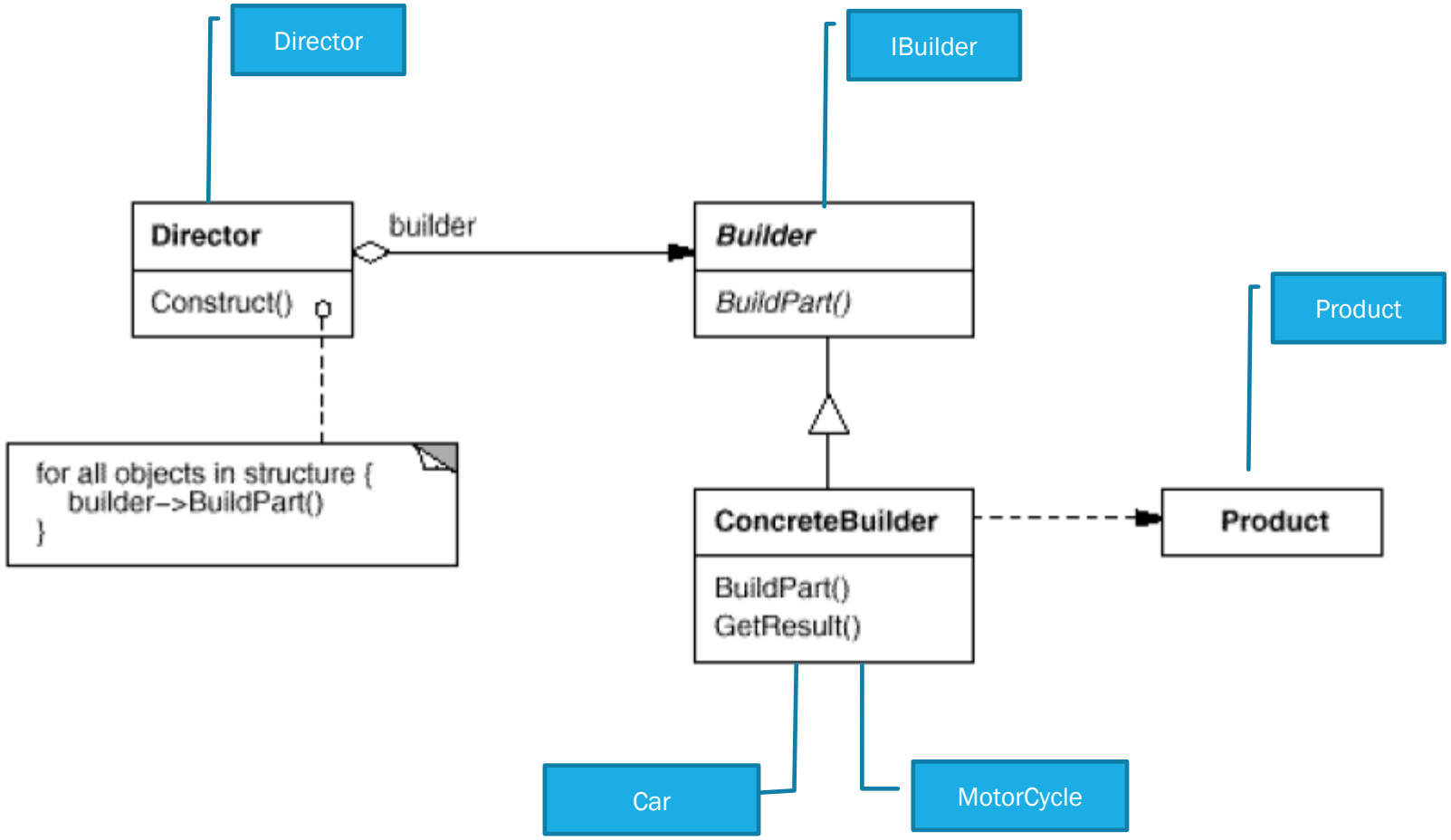
OUTPUT:

```
run:
***Builder Pattern Demo***


 Product completed as below :
This is a body of a Car
4 wheels are added
2 Headlights are added

 Product completed as below :
This is a body of a Motorcycle
2 wheels are added
1 Headlights are added
```

PARTICIPANTS:



PROS & CONS

- Pros
  - Allows you to vary a product's internal representation.
  - Encapsulates code for construction and representation.
  - Provides control over steps of construction process.
- Cons
  - Any change in concrete class requires same change in Builder as well. If its required parameter then change in client as well.