

## Lecture 28:- Builder design pattern

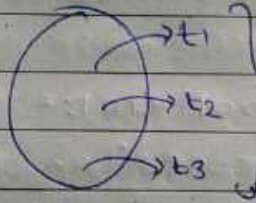
### # Introduction:-

It's most used design pattern in real life when it comes to creating objects we will use builder design pattern.  
ex:- client need a object of class.

Target t = new Target();

↑  
declare memory in Heap

multiple para that would be declared using constructor



```
class Target {
    int t1, t2, t3, ... tn;
    Target (-1 -1 -1 -)
    {
        // constructor
        this.t1 = t1;
        // ...
    }
}
```

∴ Normally we will use this syntax.  
∴ then why we need Builder pattern.

∴ We will understand the use of this pattern with Example of HttpRequest.

∴ for sending msg from sender to Receiver we use HttpRequest. It has various methods / parameters. But now here are few for understanding.





- ① URL (https://www.example.com/target)
- ② methods (GET, POST, PUT, DELETE).
- ③ Headers (Content-type; application/json).
- ④ Query params (optional).
- ⑤ Body.
- ⑥ Timeout. (60 sec).

### # problem:- constructor Overloading.

- Now create a class having execute() method when we call this a httpRequest sent from client to server.

```
class HttpRequest {
    String url;
    String method;
    Map<String, String> header;
    Map<String, String> queryParams;
    String body;
    int timeout;
```

public:-

```
HttpRequest (url, method, ...)
{
    this.url = url;
    // more variables like this
}
```

```
void execute () {
    // HTTP call
}
```

∴ we can make more constructors as all fields are not necessary.



If we do so there will be multiple constructor such as---

- ① one who takes url, method,
- ② other who takes url, method, headers.
- ③ url, method, queryparams.

So as the no. of optional fields increases you would need to write multiple constructors to entertain the request.

Client code:-

```
main()
{
    HTTPRequest *req = new HTTPRequest(url, mtd)
    _____ req2 = _____ (url, mtd, header);
}
```

∴ In this way there are diff & multiple methods.

}

This makes it complex as No. of Argument vary which leads to problem;

Problem 2:- Immutable objects.

once the problem object is created we should not be able to change object and its value i.e. objects should be immutable we can't remove setters as it is important to create new problem.



### problem 3:- Inconsistent state problem

Instead of passing all parameters through constructors, we can pass only important one and for other optional parameters we should use setter method. then when executed method in this way we can solve constructors telescoping problem.

We have to make sure that executed method will only run if pass all parameters and all values set through methods. If we do this we won't get compile time error but at runtime we can't know about error which is worst. It's called inconsistent state problem.

### problem 4:- validation

A responsible developer always validate that everything has been properly set before executed is called. If all fields are not set it leads to inconsistent state problem. developer gonna add the validation checks inside executed method to ensure req fields are present before proceeding.

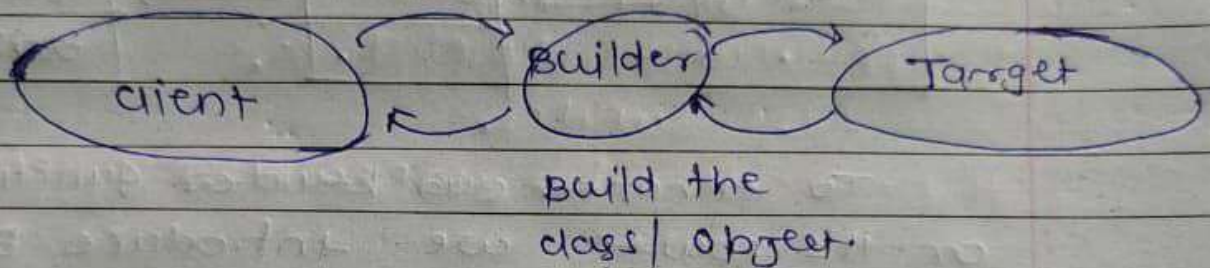


## # Introduction to Builder pattern

to Resolve the all problems such as.

- 1) constructor overriding,
- 2) constructor telescoping.
- 3) Inconsistent state problem.
- 4) Validation.

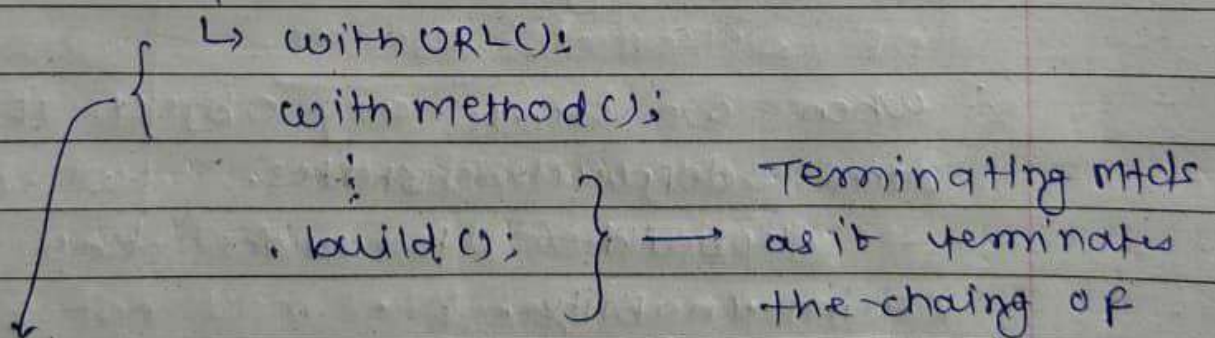
We introduced new class called 'Builder'.



## # Analyze diagram of builder pattern.

- To build the object step by step.

Request ()



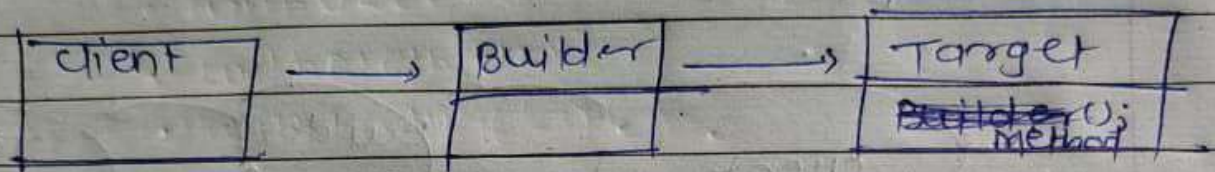
- Intermediate methods build.

returns object of builder then terminate method at last provide us request object by performing validation.



∴ main task of builder is to provide req object slowly first made one then method & so on. It will not stop until we call ~~final~~ <sup>builder</sup> method.

## # UML Diagram of simple Builder.



To enhance our builder functionalities or its power we introduce Builder with director.

It stores provides reusable builds mean it stores the preexisting default states as method. whenever anyone asks for it to give it to them.

∴ when we create any object it has some default states.

Stepbuilder provides the order maintainability.

## # Two key points.

- ① create objects step-by-step.
- ② If required, validate whenever you declare all parameters or not (optimal).



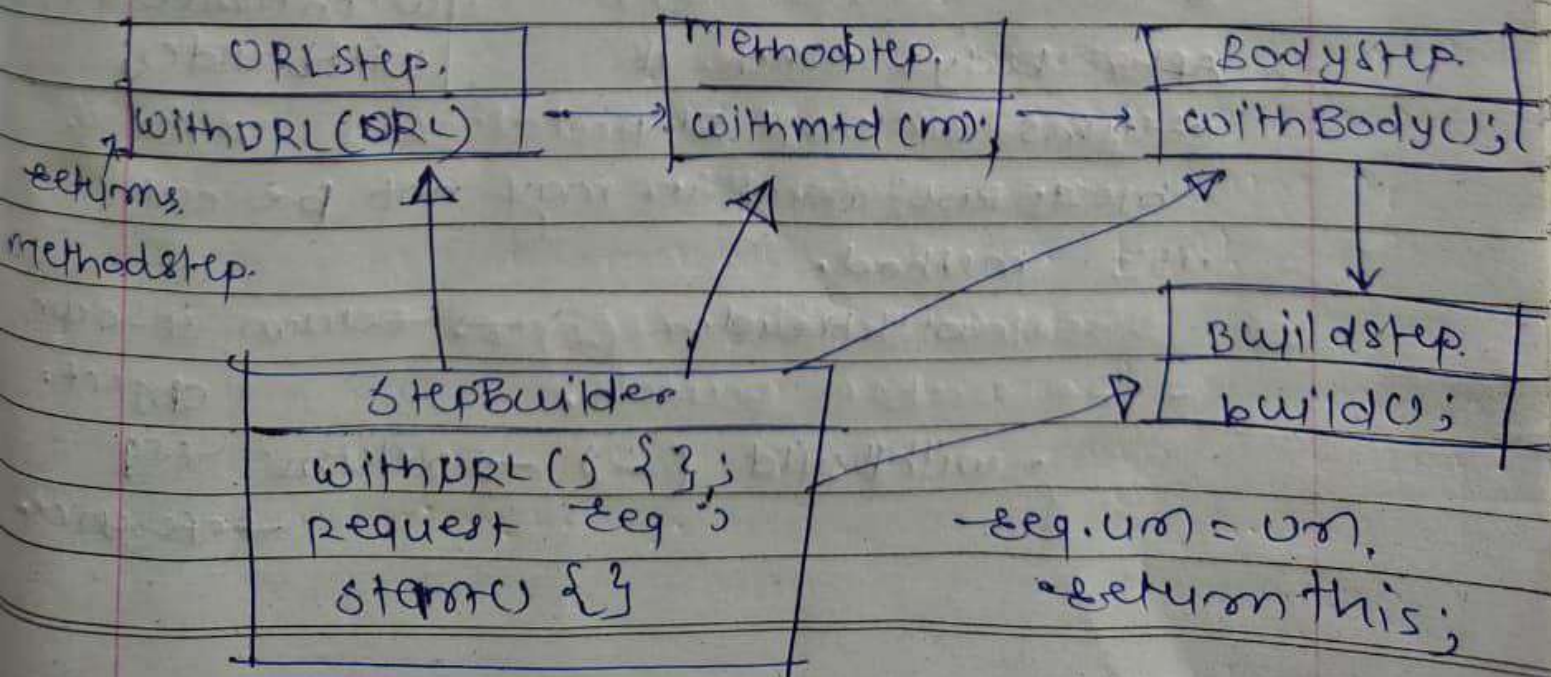
## # working of step Builder.

```
class Request {  
    String url;  
    String methods;  
    String body;  
    Map <> headers;  
}
```

we created builder class when we call withURL then it return a object of builder class which also use to access any of the method in any order.

But for step by step execution that made a separate pure abstract class of every parameter.

TO <sup>so</sup> return it first class should return object of next abstract class.





reference of

- ① we gave a `stepbuilder` class to a client.
- ② It says call `start()` {} to start object creation.
- ③ client call `start()` → `start()` return object client get an `StepObject`! or `Step`
  - `withURL(-)`
  - `withMethod()`
  - `withBody(-)`
  - `Build()`↳ stops - client gets request reference.

If we want some optional parameter we can make optional step in place of `build`.

Now `BodyStep` returns obj;  
or `OptionalStep`;

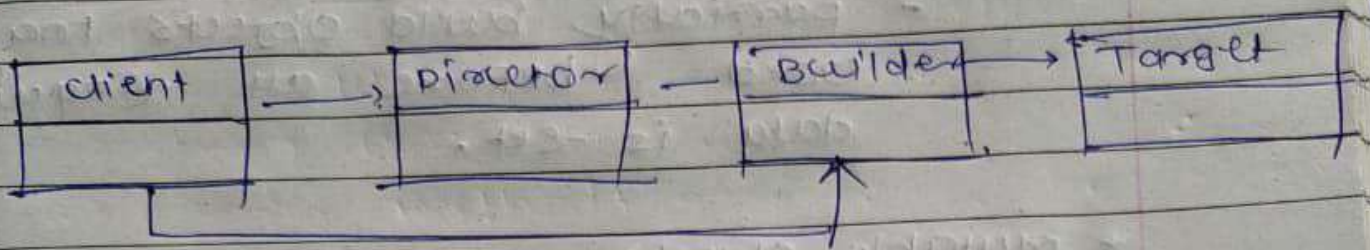
optional step.
<code>withHeader()</code> ;
<code>withTimeout()</code> ;
<code>withBuild()</code> ;

↳ After body, when it returns optional step object we may or may not be call it's method,

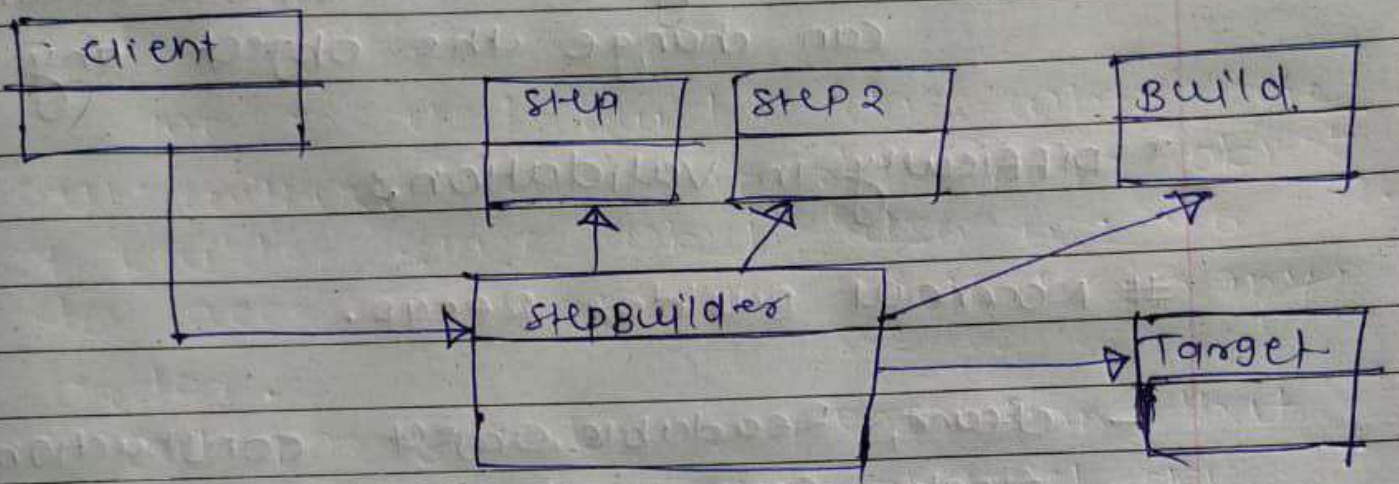
- `withHeader(-)` → return its own object.
- `withBuild(-)` → return ref reference.



## # standard UML for Builder with Director



## # standard UML step Builder



## # standard Definition

- Builder separates the construction of a complex object from its representation.

## # problem without Builder pattern

### \* constructors Expansion

- Every new optional param requires a new constructor overload.
- Calls become unreadable as you pass empty / dummy values for skipped fields.



\* Inconsistent object values.

- partially build objects may be used before all required data is set.

\* mutable objects

- exposing setter methods means client can change the object any time.

\* difficulty in validations

# Normal Builder Recap.

- clear, Readable, object construction.
- single centralised validation.
- immutable objects.
- no constructor overload.

# Step Builder Recap.

- compile time enforcement of all required fields.
- separation of mandatory vs optional.
- IDE friendly.

# Director Builder Recap.

- Reusable builds.