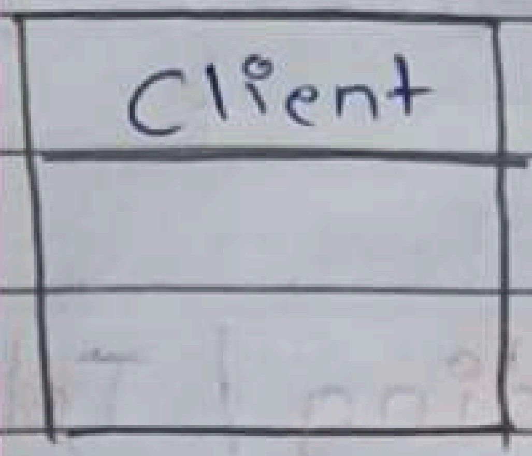
Lecture - 28 : Builder Design Pattern

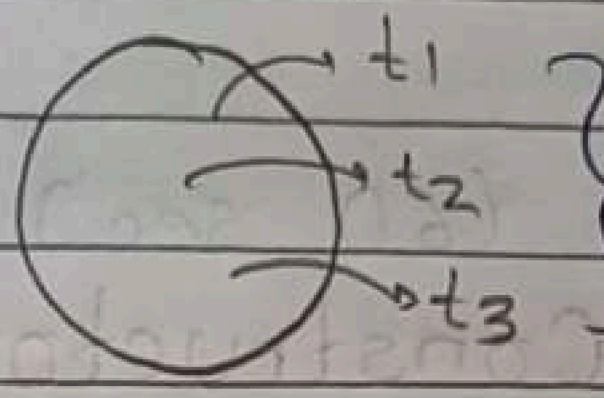
Introduction

- It is most used design pattern in industry and in real life application

- whenever it comes to creating objects, we use builder design pattern.

- Ex: client need a object of class





} Bahoot saare parameters that would be declared using constructor

```
Target * t = new Target();
```

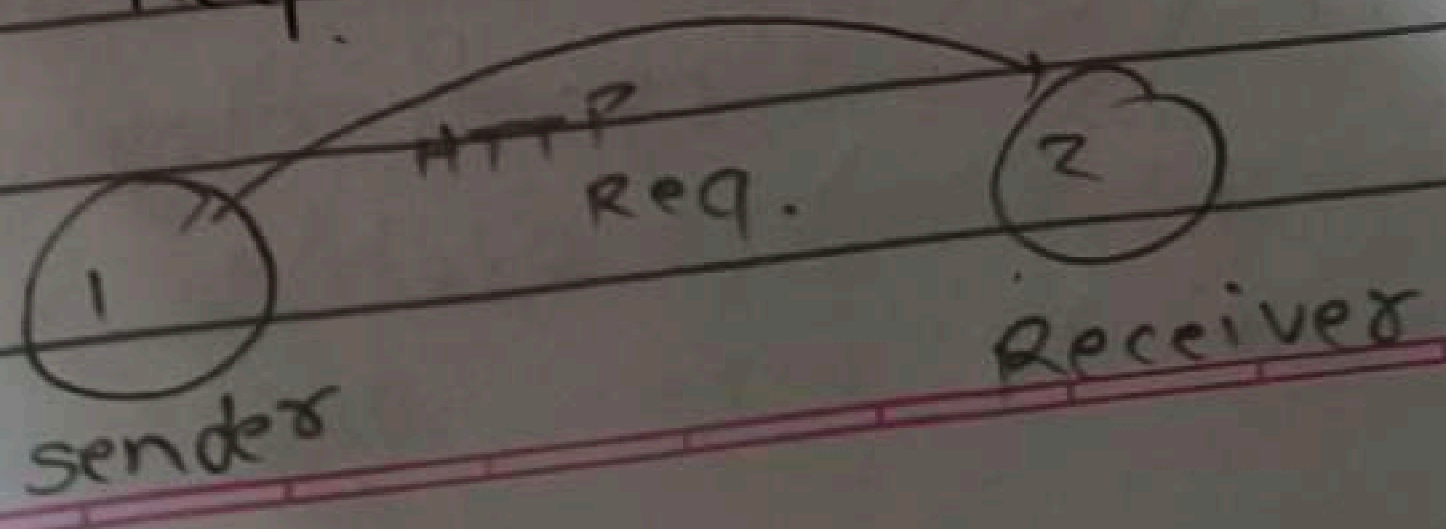
↳ declares memory in heap

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

↳ Normally hum yeh syntax use karte hai then why we need this class

↳ We will understand the use/need of this pattern with example of HttpRequest

- HTTP Request ?



- For sending message from sender to receiver, we use HTTP Request. It has various methods / parameters. But now here are few for understanding:

  → URL (https://~~exa~~ www.example.com/target)
  → methods (GET, POST, PUT, DELETE, ...)
  → headers ([content type : application/json)
  → Query Params (optional)
  → Body
  → timeout (60 sec)

Problem 1 : Constructor Overloading / Telescoping

- Now creating a class to have execute() as when we call this, a http request sent from client to server.

```
class HTTPReq {
    string url;
    string method;
    map <string, string> header;
    map <string, string> query Params;
    string body;
    int timeout;
                                                some
public:                                         code →
    HTTPReq(url, method, header, ...) {
        this → url = url    → //more stmt like this
                                for all variables
    }

    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() {
  HTTP Req * req = new HTTPReq(url, method);

  HTTPReq * req2 = new HTTPReq(url, method, headers);
     :
```

In this way the are diff. & multiple methods.

- This makes it complex as no. of arguments vary which leads to problem -
Constructor Telescoping.

Problem 2 : Immutable Object

- problem is here of setters. We want that once our object is created we should not be able to change them or its value i.e. Objects should be immutable
- We can't remove setters as it is important and it will create new problem

Problem 3 : Inconsistent state Problem

-Instead of passing all parameters through constructors and we can pass only the important ones in the constructor and use setters for the remaining values and then run execute() method. In this way we can solve constructor telescoping problem

```
class HTTPReq {
    string url;
    string method;
    map <string, string> header;
    map <string, string> queryParams;
    string body;
    int timeout;

    HTTPReq (url, method, header) {      // only contains
        this ->url = url;                   imp value /
        this ->method = method;             parameter
        this -> header = headers;
    }
    // Getter & Setters

    execute () {
        ≡
    }

}   main () {
    HTTPReq * req = new HTTPReq(url, method, headers);
    req -> setBody (-);
    req -> setTimeout (-);
}   req -> execute();
```

No overloading of constructor

- We have to make sure now that execute() will only run if all methods are passed or set.

- ex: we pass 3 obj as argument & set body only not timeout and queryParams then it shouldn't run

- But if we do this it will not give us compile time error but runtime error. which is worst if we come to know about error at runtime.
  ↳ Inconsistent state Problem.

- Ex: Even if we declare all variable but call execute() before some ~~e~~ set methods still it will give us run time error.

Problem 4: Validation

- Now let's assume I am a responsible developer who always ~~test~~ remembers to set all the required values and never leaves the object in inconsistent object. Even then, I still want a way to validate that everything has been properly set before execute() is called - especially when someone else is using my class.

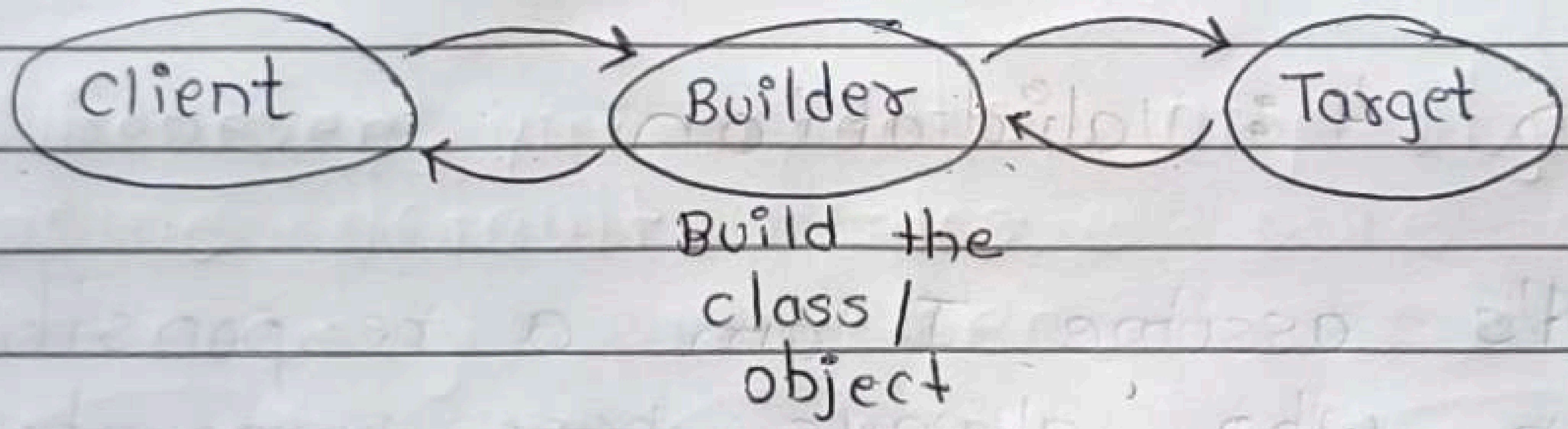
- To ensure this, we are gonna add validation checks inside execute() method to ensure required fields are present before proceeding.

```
void execute () {
    if (req —> getURL () == null)
            throw error;
    if (req —> getheader() == null)
            throw error;
              :
}
```

- We have to perform this validation everywhere where we have used this req object This problem is known as Scattered Validations

Introduction to Builder Design Pattern

- To resolve all the above problems we introduced a new class called 'builder'.



- To understand this part / pattern : check code.

- Firstly, let us understand about this keyword

```
class A {
  int i1, i2;
  m1() {
    this;
  }
}
```

→ this keyword return kr rha h A ka current reference kisko return kar h → client ko jisne A ka obj banaya hoga

Ex-2

- class A {
```
class A {
    int x1, x2;
    A() {
        this.x1 = x1        // this mtlb current value
        this.x2 = x2        ko reference of in
    }                       values ko set krdo.
}
```

Analyzing Diagram of Builder Pattern

- To build a object step by step

```
Request ()
    ↳ .withURL (——)
    ↳ . withmethod ( — )
      ⋮
    ↳ . build () ⟶ Terminating method
                   as it terminates the
                   chaining of build
```

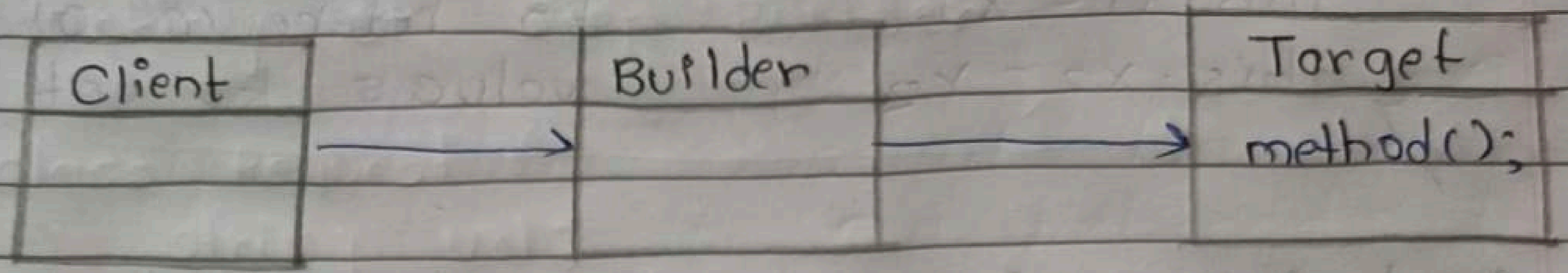
- other method known as <u>Intermediate method</u> as they do chaining.

- All intermediate method returns object of builder then our terminating method at last provide us ~~remaining~~ request object by performing validations.

→ Efficient than setters due to
- provide immutability
- improves readability

→ Main task of builder is to provide is a req object slowly first made are then method & so on. It will not stop until we call final build() method

UML Diagram of Simple Builder

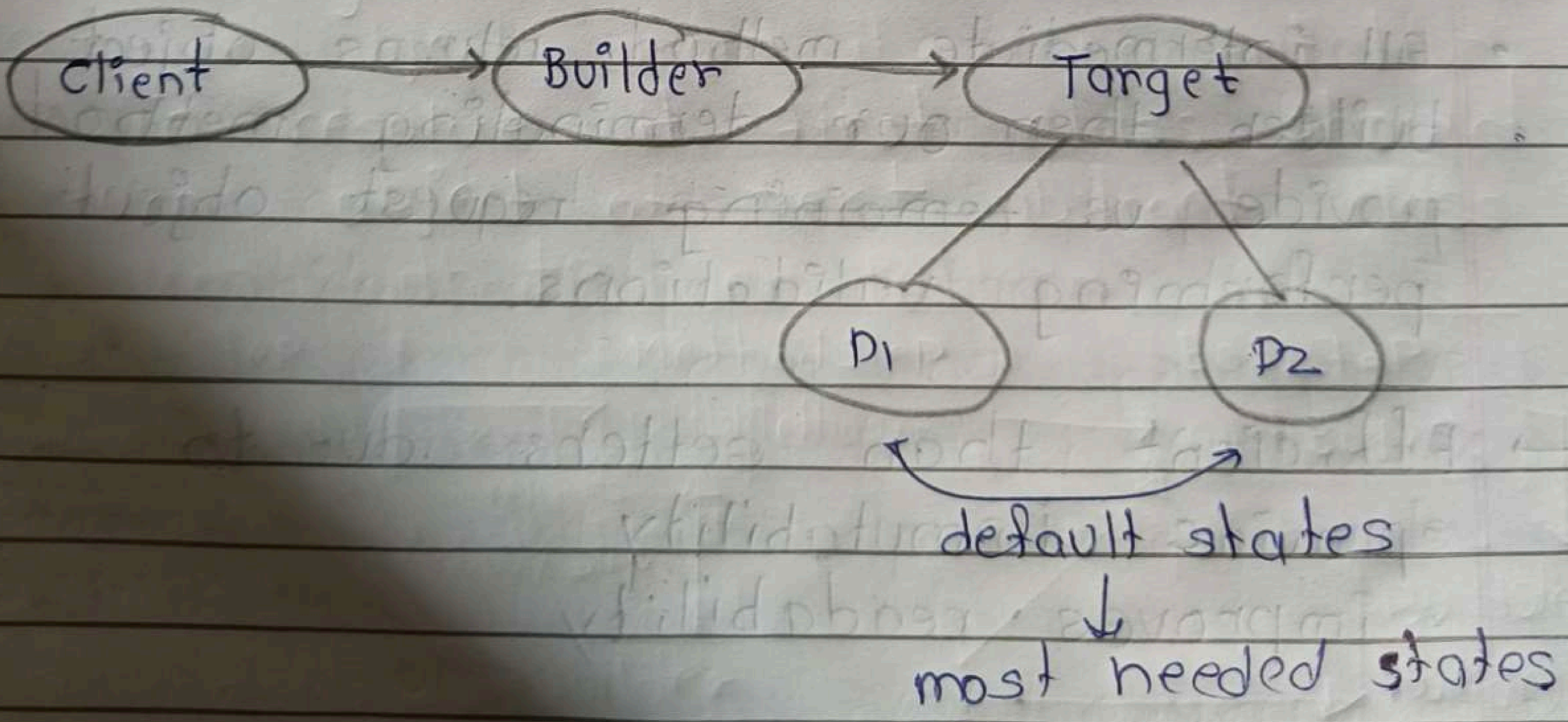


Builder with Director

→ To enhance our builder functionalities or its power, we introduce Builder with Director

→ It provides reusable builds means it stores the preexisting default states as method whenever anyone ask for it then give it to them.

→ Whenever you create any object it has some default states.



Step Builder

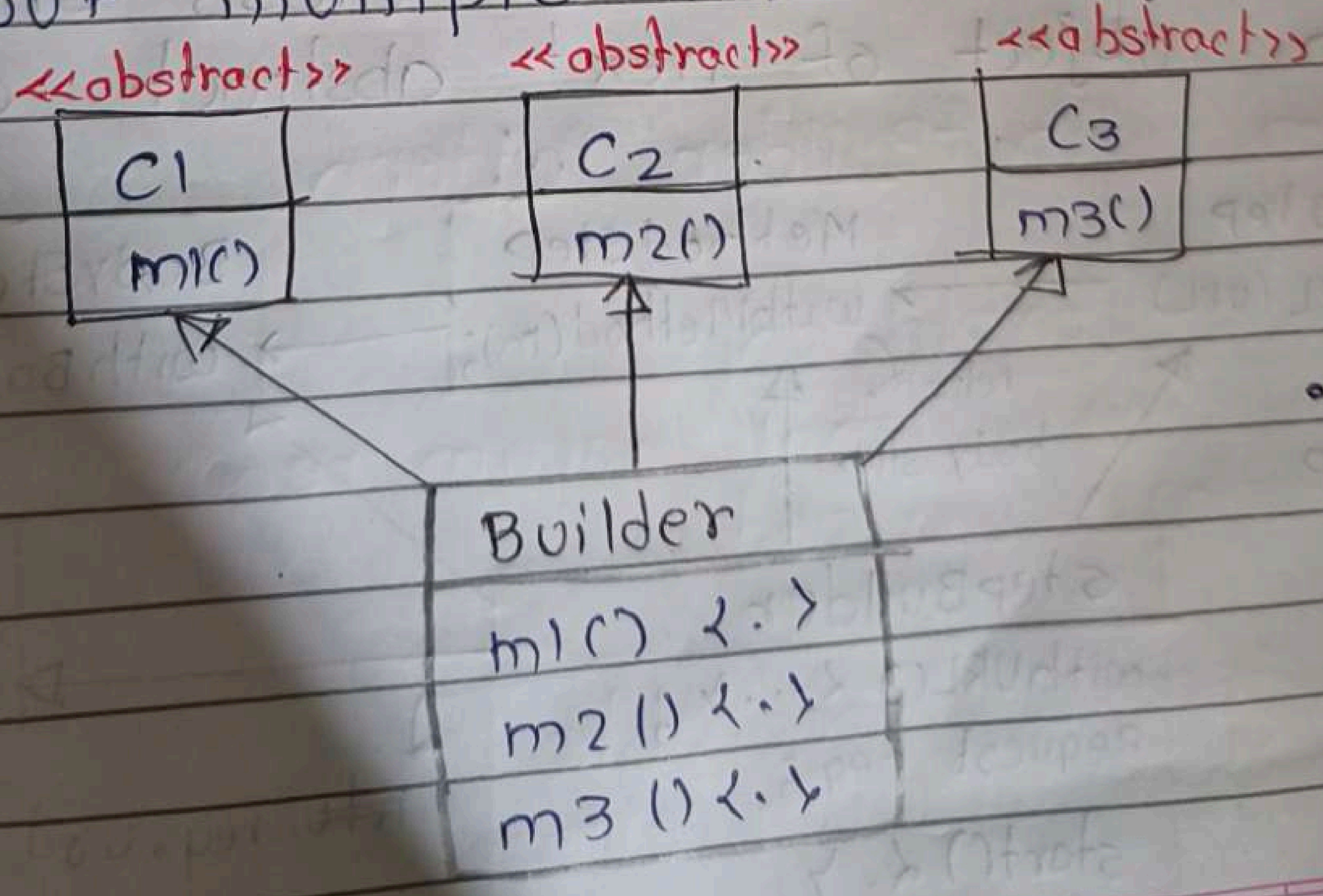
- Let's take example of pizza when we ask for pizza, they asks us some questions about pizza in an order like
  - → Crust
  - → Sauce
  - → Toppings
  - → Cheese

- Some objects are needed to be created in an specific order.

- This **Order Maintainability** is provided by stepBuilder.

⊙ Two key points
- Create objects step by step
- If required, validate whether you declare all parameters or not (optimal)

- Before going further, let us understand about multiple inheritance



- only pattern to use multiple inheritance

- Use tradeoff ⟶ more strictness ⟶ use multiple inheritance.

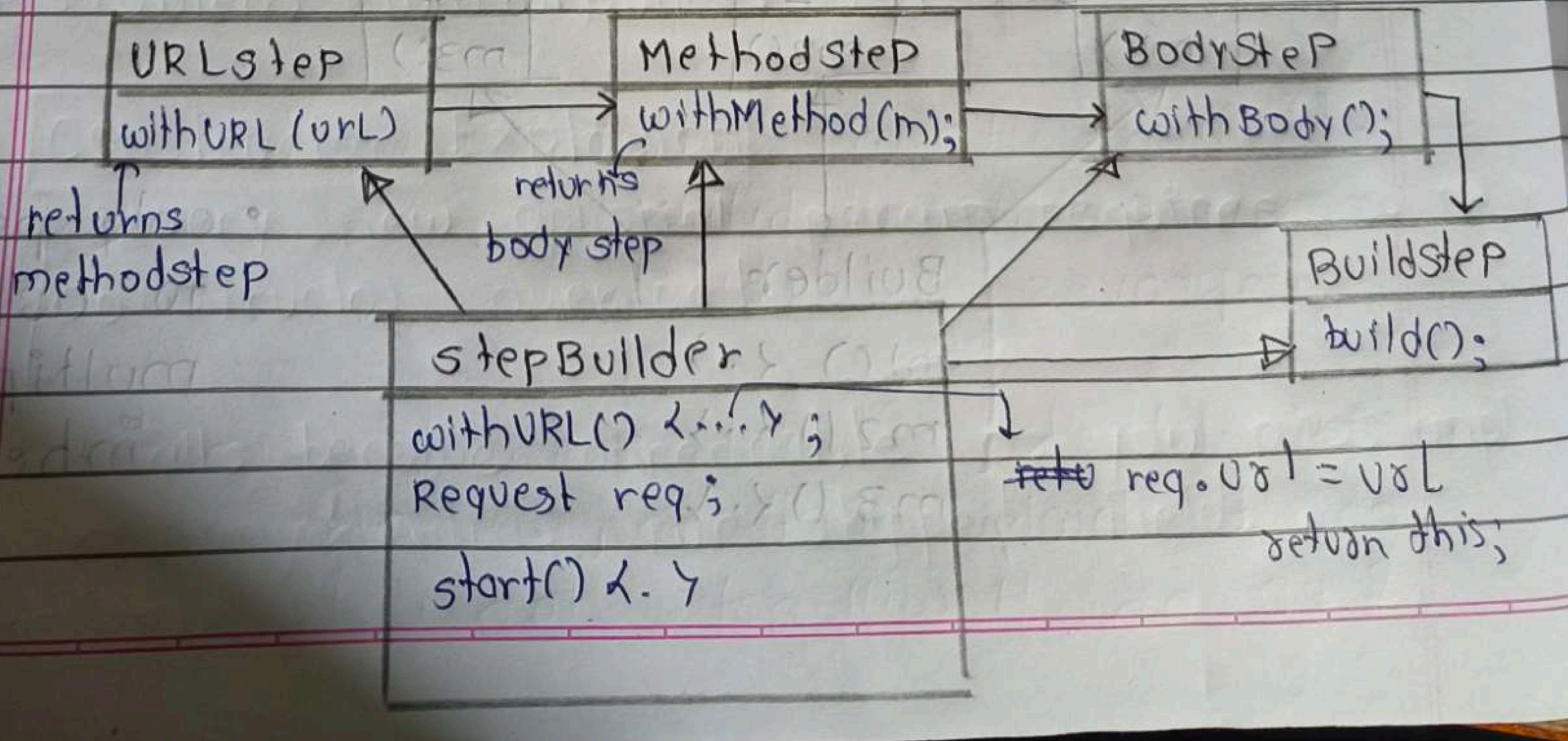
Working of step Builder

-
```
class Request {
        string url;
        string method;
        string body;
        map <> headers;
}
```

- Ab humne builder class banaya Till now, jb bhi hum withUrl then it return a object of builder class which also us to access any of the method in any order

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

- To resolve it first class should return object of next abstract class



① We gave a reference of stepBuilder class to client.

② It says call start ()<> to start object creation

③ Client call start () ⟶ start () return URLstep object — client gets URL step object.

- withURL (—)
- withMethod (—)
- withBody (—)
- Build ()
  └→ stops → client get Request reference

If we want some optional parameter we can make Optionalstep inplace of build

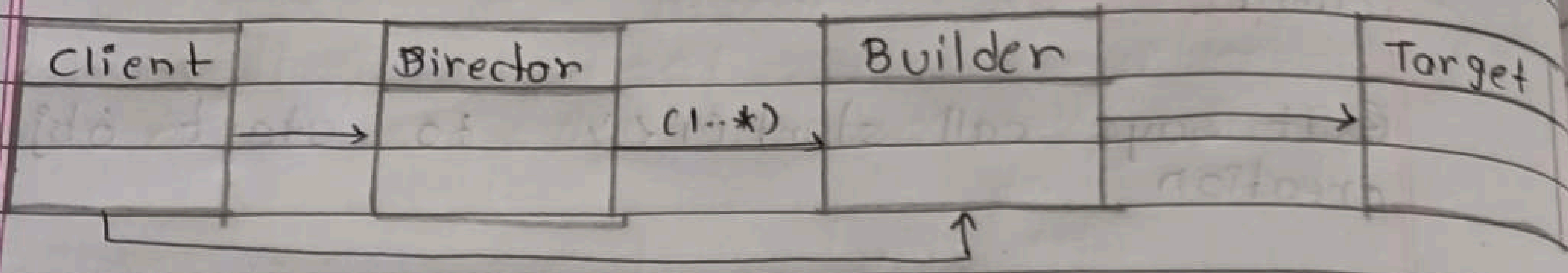
Now bodystep ~~has~~ returns obj of optionalstep

Optionalstep
withheader();
withtimeout();
with build();

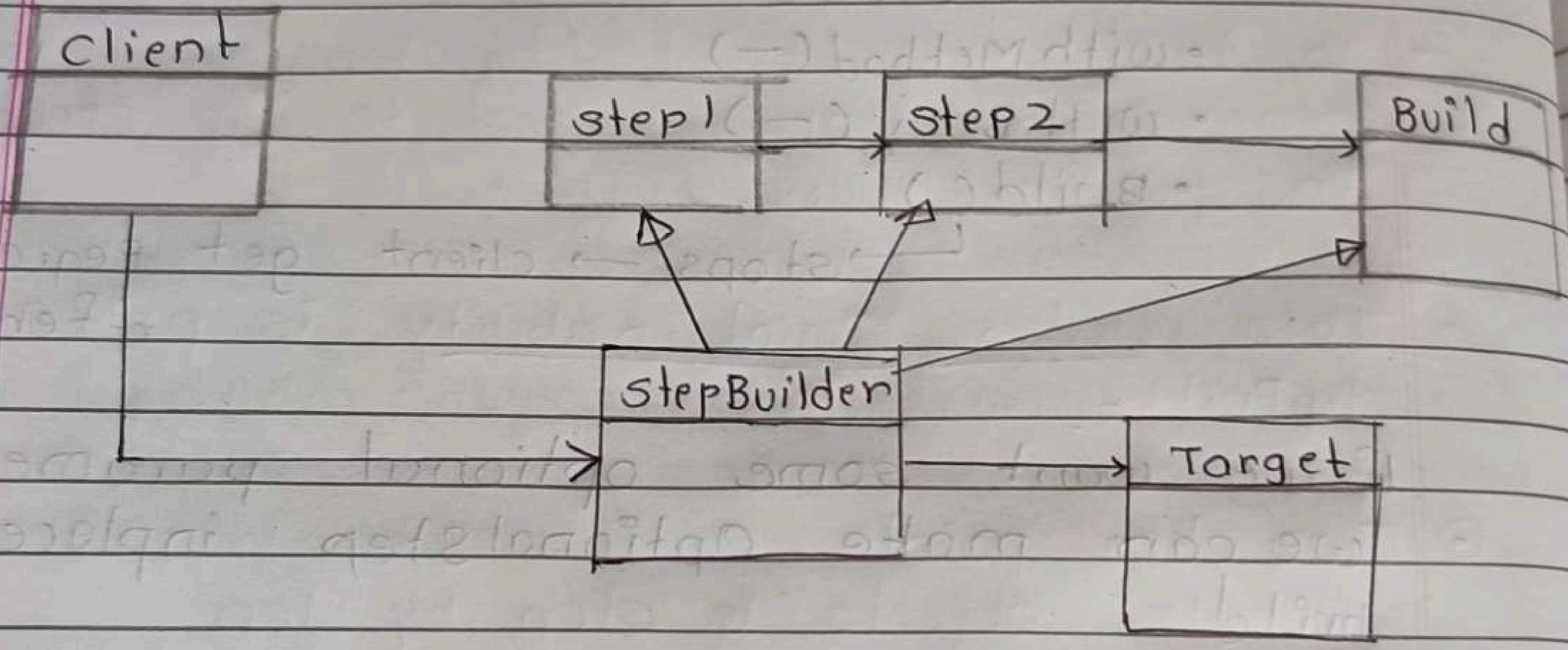
→ After body, when it return optional step object, we may or may not call its method.

- withheader(—) → return its own object
- with build(—) → returns req reference.

Standard UML for Builder with Director



Standard UML for step Builder



Standard Definition

↳ Builder separates the construction of a complex object from its representation

Problem without Builder Pattern

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

- Inconsistent Object status
  - ↳ Partially built objects may be used before all required data is set.
- Mutable Objects
  - ↳ Exposing setters means client can change the object any time.
- Difficulty in Validations

Normal Builder Recap

- → clear, readable object construction
- → single centralized validation
- → Immutable Objects.
- ↳ No constructor overload

Director Builder Recap

- ↳ Reusable Builds

Step Builder Recap

- → Incremental / compile time enforcement of all required fields.
- → Separation of mandatory vs optimal
- ↳ IDE friendly.