Builder

Resources

- Video → 27. All Creational Design Patterns | Prototype, Singleton, Factory, AbstractFactory, Builder Pattern
- Video → 23. Builder Design Pattern with Examples, LLD | Low Level Design Interview Question | System Design

Definition

> ***The Builder Pattern is a creational design pattern that constructs complex objects step by step.***

It's particularly useful when you need to create objects with many optional parameters or when the construction process should allow different representations of the same object.

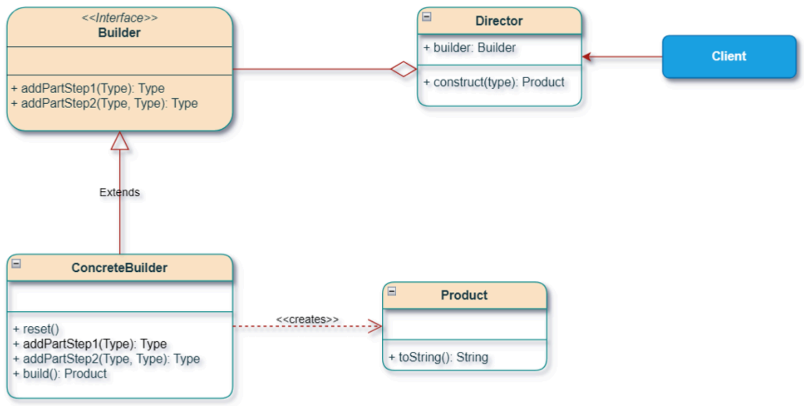
The Problem

```java
// Step 1: Product class - The complex Student object being built
public class Student {

    // mandatory fields
    int rollNumber;
    int age;
    String name;
    String branch;
    // optional fields
    String fatherName;
    String motherName;
    List<String> subjects;
    String mobileNo;
    String emailId;

    // Without Builder - Constructor overloading
    // Telescoping Constructor Problem - adding mandatory parameters
    public Student(int rollNumber, int age, String name, String branch) {
        this.rollNumber = rollNumber;
        this.age = age;
        this.name = name;
        this.branch = branch;
    }

    // Telescoping Constructor Problem - adding new parameter - fatherName(optional)
    public Student(int rollNumber, int age, String name, String branch, String fatherName) {
        this.rollNumber = rollNumber;
        this.age = age;
        this.name = name;
        this.branch = branch;
        this.fatherName = fatherName;
    }

    // Telescoping Constructor Problem - adding new parameter - motherName(optional)
    public Student(int rollNumber, int age, String name, String branch, String fatherName, String motherName) {
        this.rollNumber = rollNumber;
        this.age = age;
        this.name = name;
        this.branch = branch;
        this.fatherName = fatherName;
        this.motherName = motherName;
    }

    // Telescoping Constructor Problem - adding new parameter - emailId(optional)
    // Compilation error - Constructor signature is same as another constructor(below)
    public Student(int rollNumber, int age, String name, String branch, String fatherName, String motherName, String emailId) {
        this.rollNumber = rollNumber;
        this.age = age;
        this.name = name;
        this.branch = branch;
        this.fatherName = fatherName;
        this.motherName = motherName;
        this.emailId = emailId;
    }

    // Telescoping Constructor Problem - adding new parameter - emailId(optional)
    // Compilation error - Constructor signature is same as another constructor(above)
    /*public Student(int rollNumber, int age, String name, String branch, String fatherName, String motherName, String mobileNo) {
        this.rollNumber = rollNumber;
        this.age = age;
        this.name = name;
        this.branch = branch;
        this.fatherName = fatherName;
        this.motherName = motherName;
        this.mobileNo = mobileNo;
    }*/

    // Telescoping Constructor Problem - All arguments constructor
    // Big Constructor with long parameter list
    public Student(int rollNumber, int age, String name, String branch, String fatherName, String motherName, List<String> subjects, String mobileNo, String emailId) {
        this.rollNumber = rollNumber;
        this.age = age;
        this.name = name;
        this.branch = branch;
        this.fatherName = fatherName;
        this.motherName = motherName;
        this.subjects = subjects;
        this.mobileNo = mobileNo;
        this.emailId = emailId;
    }

    public void printDetails() {
        System.out.println("=== Student Details ===");
        System.out.print(this + ": ");
        System.out.println("Id: " + rollNumber +
                ", Name: " + name +
                ", Age: " + age +
                ", Branch: " + branch +
                ", Roll No: " + rollNumber +
                ", Father Name: " + fatherName +
                ", Mother Name: " + motherName +
                ", Subjects: " + subjects +
                ", Mobile No: " + mobileNo +
                ", Email Id: " + emailId);
    }
}
```

Here's a list of problems and issues that typically arise when we create similar and multiple complex objects, the traditional way.

- When a class has many optional parameters, you end up with a large number of constructors. This is called Telescoping Constructors (Constructor Overload Explosion).
- It is difficult to maintain, read, and prone to errors when parameters are of the same type, particularly when constructors have identical signatures.
- Reduced readability and maintainability, as we can't tell what each parameter means unless we look at the constructor definition. Chances of wrong assignment of parameter values due to the same types.
- Clients should pass `null` for any optional parameters that they do not choose to set, which increases the risk of bugs.
- If you want to make a class immutable (i.e., its fields are final), you cannot use setters.
- Creating test objects becomes increasingly complex and verbose without builders, particularly when many fields are optional.
- Violates the Single Responsibility Principle (SRP) because the class manages both data storage and construction logic.

Class Diagram



<<Interface>>
Builder
+ addPartStep1(Type): Type
+ addPartStep2(Type, Type): Type
Director
+ builder: Builder
+ construct(type): Product
Client
Extends
ConcreteBuilder
+ reset()
+ addPartStep1(Type): Type
+ addPartStep2(Type, Type): Type
+ build(): Product
<<creates>>
Product
+ toString(): String

Example: Student Registration

1. **Product (** `Student` **)**
   a. The `product` is a complex object that is being constructed.
   b. Contains many optional and required components.
   c. Has package-private setters to ensure only builders can modify it.
2. **Abstract Builder Interface (** `StudentBuilder` **)**

- Defines all possible construction steps.
- Provides a fluent interface (for method chaining).
- Includes a `build()` method to return the final product(`Student`).

3. **Concrete Builders (`EngineeringStudentBuilder` and `MBAStudentBuilder`)**
   - Implement the builder interface.
   - Each builder creates a specific type of product.
   - Contain the logic for how each component should be built for that specific product type.
4. **Director (`StudentRegistrationDirector`) - Optional**
   - Defines common construction sequences.
   - Encapsulates the order of construction steps.
   - Provides convenience methods for standard configurations.

Implementation

```java
// Step 1: Product class - Complex Student object
public class Student {

    // mandatory fields
    int rollNumber;
    int age;
    String name;
    String branch;
    // optional fields
    String fatherName;
    String motherName;
    List<String> subjects;
    String mobileNo;
    String emailId;

    // Constructor - package private, only the builder can create
    Student(StudentBuilder builder) {
        this.rollNumber = builder.rollNumber;
        this.age = builder.age;
        this.name = builder.name;
        this.branch = builder.branch;
        this.fatherName = builder.fatherName;
        this.motherName = builder.motherName;
        this.subjects = builder.subjects;
        this.mobileNo = builder.mobileNo;
        this.emailId = builder.emailId;
    }

    public String toString() {
        return " roll number: " + this.rollNumber +
                " age: " + this.age +
                " name: " + this.name +
                " branch: " + this.branch +
                " father name: " + this.fatherName +
                " mother name: " + this.motherName +
                " subjects: " + subjects.get(0) + "," +
subjects.get(1) + "," + subjects.get(2) +
                " mobile no: " + this.mobileNo +
                " email id: " + this.emailId;
    }

}
```

```java
// Step 2: Abstract Builder interface
public abstract class StudentBuilder {

    // mandatory fields
    int rollNumber;
    int age;
    String name;
    String branch;
    // optional fields
    String fatherName;
    String motherName;
    List<String> subjects;
    String mobileNo;
    String emailId;

    public StudentBuilder setRollNumber(int rollNumber) {
        this.rollNumber = rollNumber;
        return this;
    }

    public StudentBuilder setAge(int age) {
        this.age = age;
        return this;
    }

    public StudentBuilder setName(String name) {
        this.name = name;
        return this;
    }

    public StudentBuilder setBranch(String branch) {
        this.branch = branch;
        return this;
    }

    public StudentBuilder setFatherName(String fatherName) {
        this.fatherName = fatherName;
        return this;
    }

    public StudentBuilder setMotherName(String motherName) {
        this.motherName = motherName;
        return this;
    }

    public StudentBuilder setMobileNo(String mobileNo) {
        this.mobileNo = mobileNo;
        return this;
    }

    public StudentBuilder setEmailId(String emailId) {
        this.emailId = emailId;
        return this;
    }

    abstract public StudentBuilder setSubjects();

    // Build method
    public Student build() {
        return new Student(this);
    }

}
```

```java
// Step 3: Concrete Builder for Engineering Students
public class EngineeringStudentBuilder extends StudentBuilder {

    // Engineering-specific methods
    @Override
    public StudentBuilder setSubjects() {
        List<String> enggSubjectsList = new ArrayList<>();
        enggSubjectsList.add("Operating Systems");
        enggSubjectsList.add("Computer Architecture");
        enggSubjectsList.add("Data Structures");
        enggSubjectsList.add("DBMS");
        this.subjects = enggSubjectsList;
        return this;
    }
}

// Step 4: Concrete Builder for MBA Students
public class MBAStudentBuilder extends StudentBuilder {

    // MBA-specific methods
    @Override
    public StudentBuilder setSubjects() {
        List<String> mbaSubjectsList = new ArrayList<>();
        mbaSubjectsList.add("Micro Economics");
        mbaSubjectsList.add("Business Studies");
        mbaSubjectsList.add("Operations Management");
        mbaSubjectsList.add("Financial Management");
        this.subjects = mbaSubjectsList;
        return this;
    }
}
```

```java
// Step 5: Director class for common student registration processes
public class StudentRegistrationDirector {

    StudentBuilder studentBuilder;

    StudentRegistrationDirector(StudentBuilder studentBuilder) {
        this.studentBuilder = studentBuilder;
    }

    public Student createStudent() {
        if (studentBuilder instanceof EngineeringStudentBuilder) {
            return createEngineeringStudent();
        } else if (studentBuilder instanceof MBAStudentBuilder) {
            return createMBAStudent();
        }
        return null;
    }

    private Student createEngineeringStudent() {
        return studentBuilder.setRollNumber(1) //
EngineeringStudentBuilder
                .setAge(22)
                .setName("John")
                .setFatherName("Paul")
                .setMotherName("Jane")
                .setBranch("Computer Science and Engineering")
                .setSubjects() // Engineering-specific method
                .build();
    }

    private Student createMBAStudent() {
        return studentBuilder.setRollNumber(2) // MBAStudentBuilder
                .setAge(24)
                .setName("Sarah")
                .setFatherName("Gabriel")
                .setMotherName("Taylor")
                .setBranch("Business Administration")
                .setSubjects() // MBA-specific method
                .setMobileNo("9876543210")
                .setEmailId("sarahgabriel@iitb.com")
                .build();
    }
}
```

```java
// Step 6: Client demonstration
public class Client {

    public static void main(String[] args) {
        System.out.println("===== Builder Pattern =====");
        // Create director objects
        StudentRegistrationDirector enggStudentDirector = new 
StudentRegistrationDirector(new EngineeringStudentBuilder());
        StudentRegistrationDirector mbaStudentDirector = new 
StudentRegistrationDirector(new MBAStudentBuilder());

        // Create students using different builders
        Student engineerStudent = enggStudentDirector.createStudent();
        Student mbaStudent = mbaStudentDirector.createStudent();

        // Print student details
        System.out.println("===> Student details:" + 
engineerStudent.toString());
        System.out.println("===> Student details:" + 
mbaStudent.toString());

    }
}
```

Builder Pattern vs Decorator Pattern

Refer → 23. Builder Design Pattern with Examples, LLD | Low Level Design Interview Question | System Design

Conclusion:

- If you're building complex objects: **Use the Builder Pattern**
- If you're extending behavior dynamically(layering enhancements): **Use the Decorator Pattern**