Lecture 9

Dr. Umair Rehman

Agenda

- Creational Design Patterns: Builder Pattern
  - Builder Pattern Overview
  - Without Builder Pattern
  - Telescoping Constructors
  - Builder Pattern with Pointers
  - Builder Pattern without Pointers
  - Pros and Cons of Builder Pattern

Builder Pattern Overview

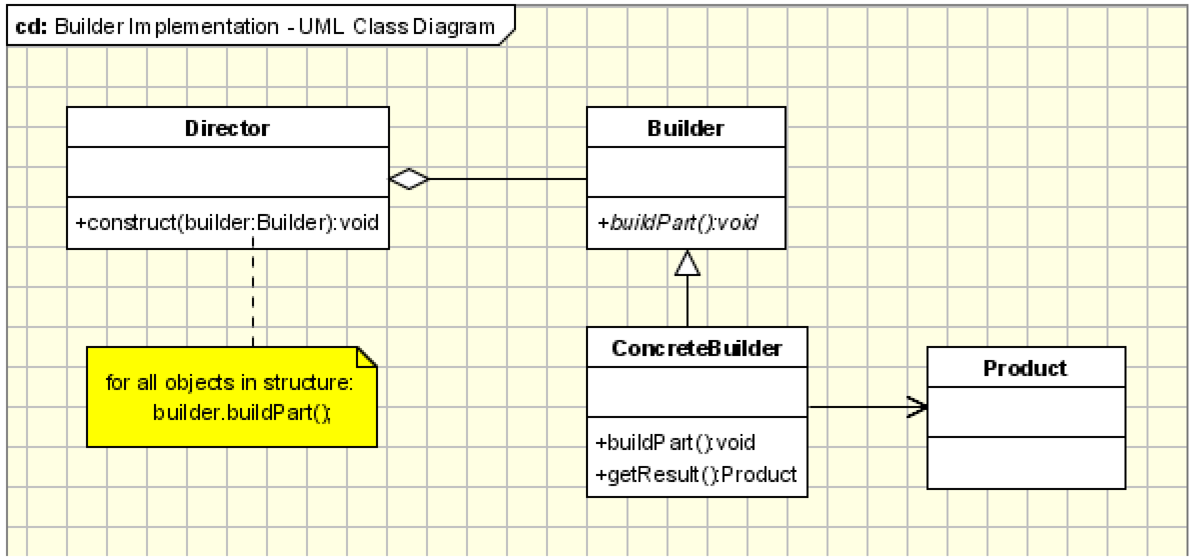
- Creational design pattern to construct complex objects step-by-step.

- Useful when an object has
  - multiple configurations
  - optional parameters

Builder Pattern Overview

- Separating the construction of an object from its representation
- Builder pattern provides more control over the construction process
- Easier to create complex or customizable objects

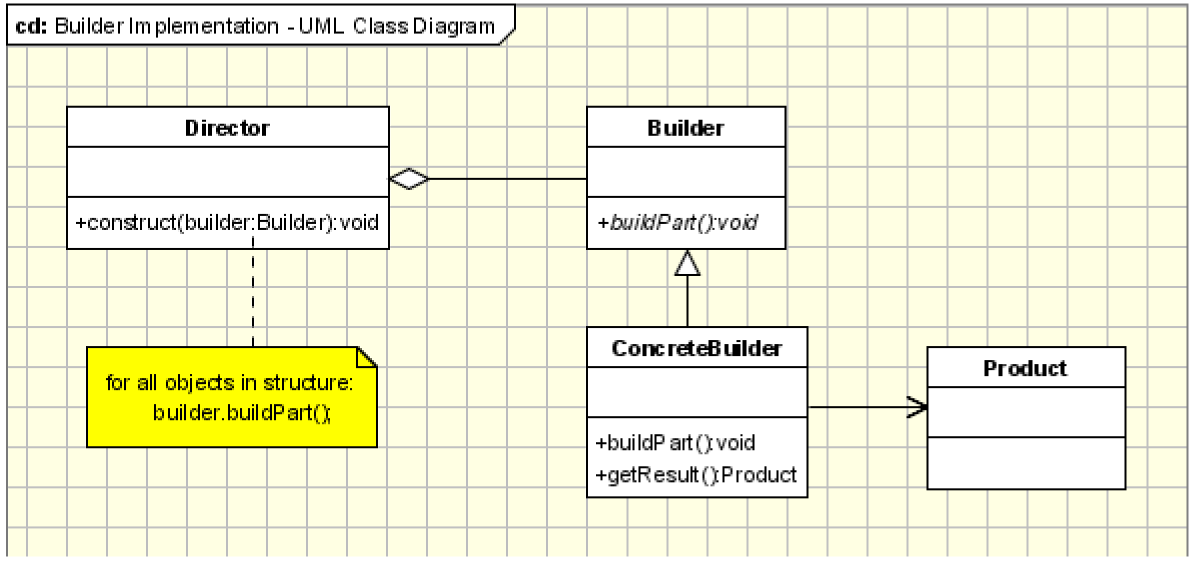
Builder Pattern Implementation



Source: https://www.oodesign.com/builder-pattern

Define the Product

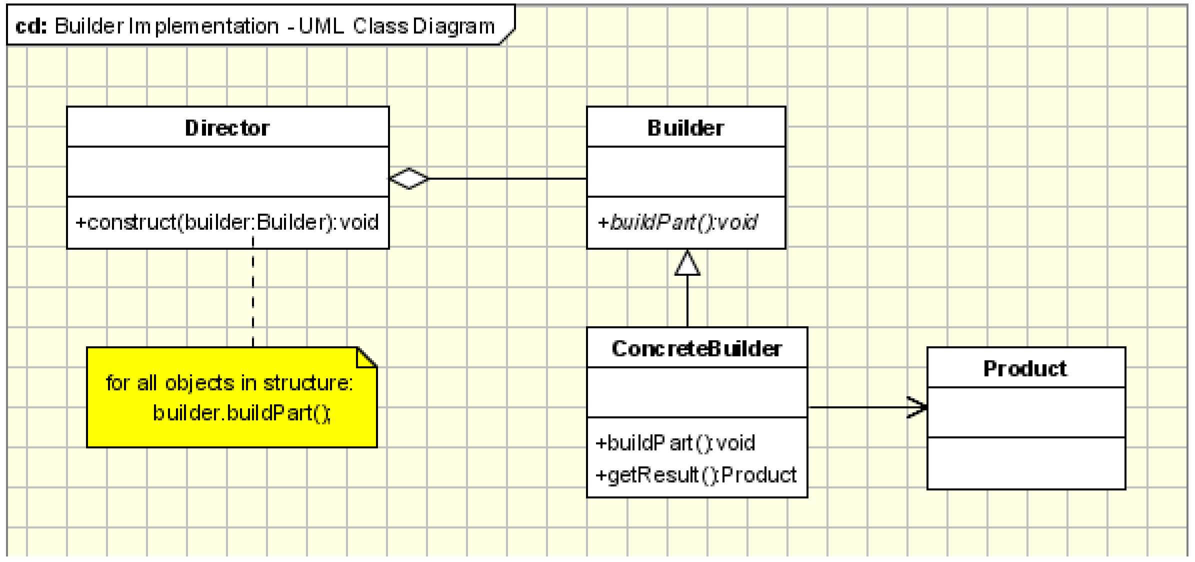
- This is the class that we want to create (e.g., Product).



Source: https://www.oodesign.com/builder-pattern

Create the Builder Interface or Abstract Class

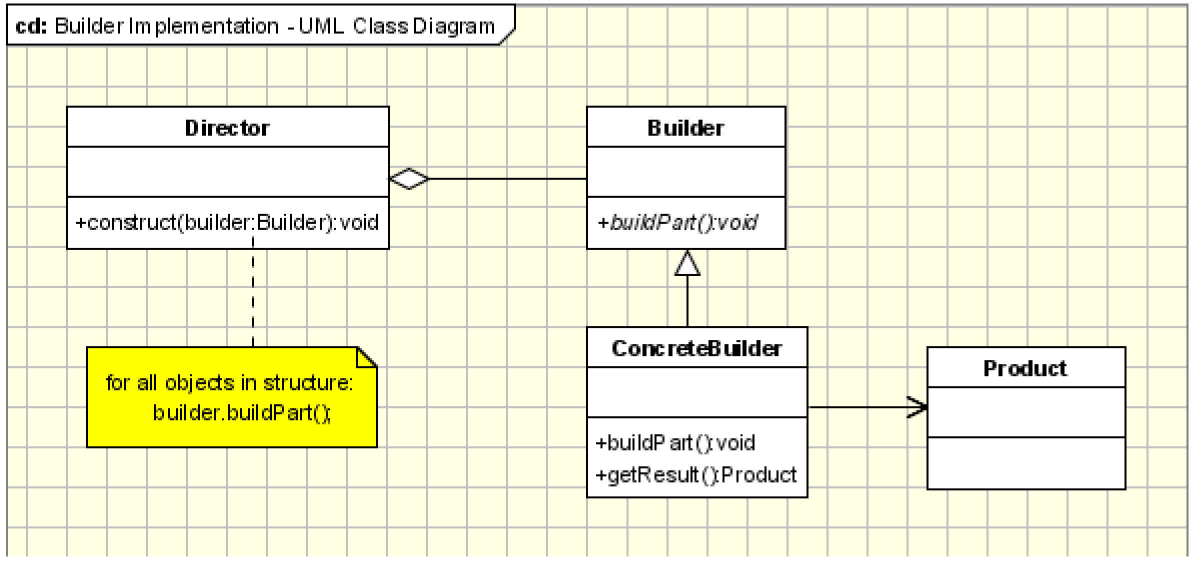
- This outlines the steps needed to create the product.



Source: https://www.oodesign.com/builder-pattern

Implement Concrete Builders

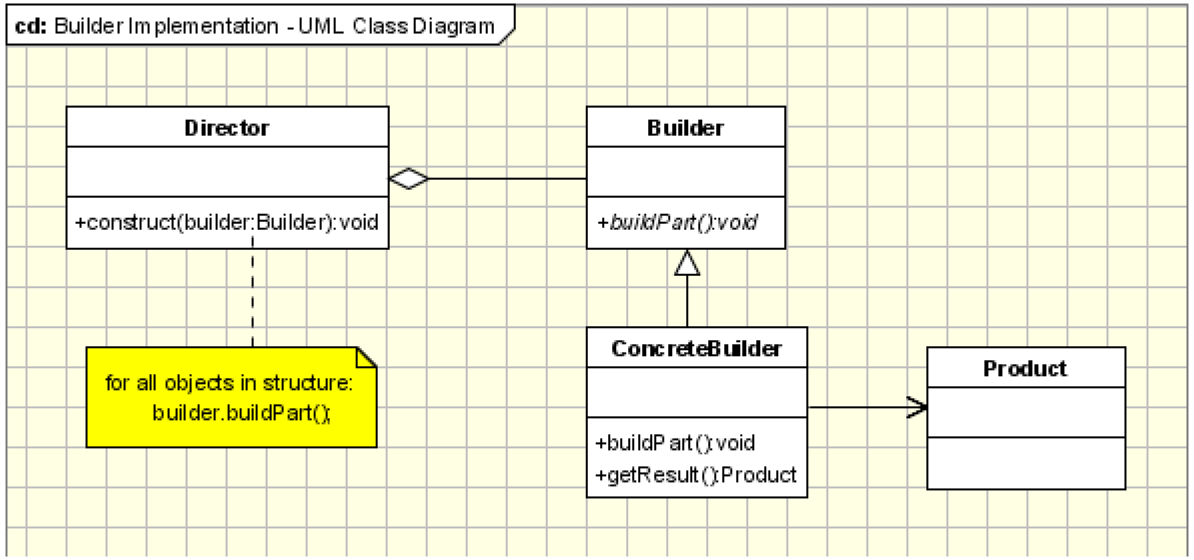
- Each concrete builder class implements the steps to create a specific variation of the product.



Source: https://www.oodesign.com/builder-pattern

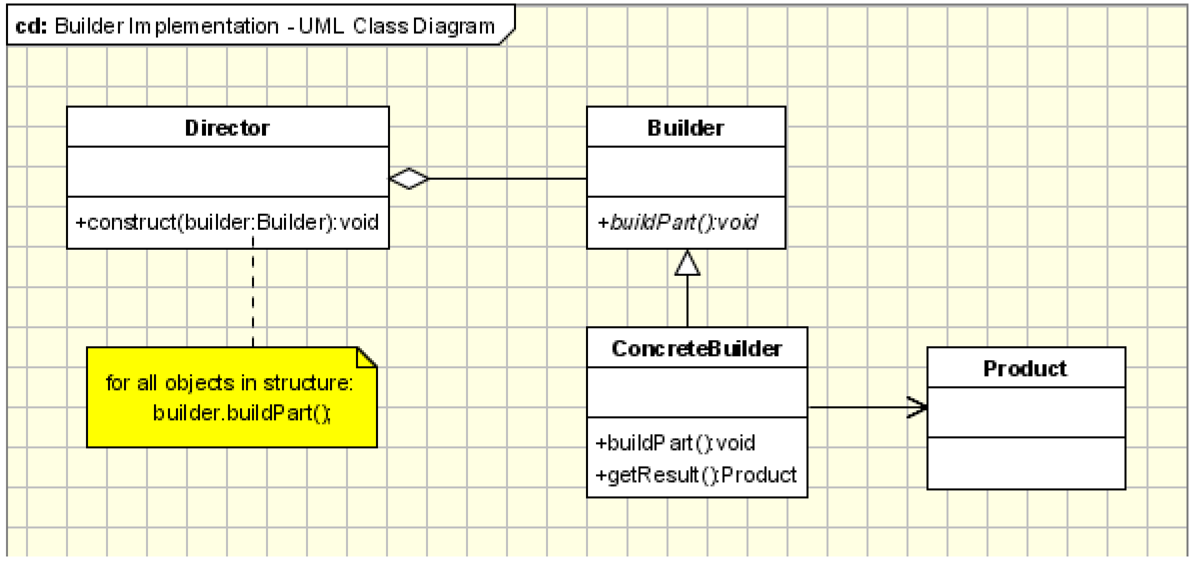
Use a Director (optional)

- The director class is responsible for managing the construction process and ensuring that all parts are built in the correct order.



Source: https://www.oodesign.com/builder-pattern

Builder to Create the Object



Source: https://www.oodesign.com/builder-pattern

Example Problem

- Designing a flexible system to construct different types of cars
    - With varying configurations (like engine type, seat count, and color)

- We want to build various types of cars (e.g., sports cars, family cars) that may have different features and specifications.

Without the Builder Pattern

```cpp
#include <iostream>
#include <string>

using namespace std;

// Car class representing the product
class Car {
private:
    string engine;
    int seats;
    string color;

```

Without the Builder Pattern

```cpp
public:
    // Constructor that initializes with default or given values
    Car(const string& engineType = "", int seatCount = 0, const string& carColor = "")
        : engine(engineType), seats(seatCount), color(carColor) {}

    // Setter methods to set each property manually
    void setEngine(const string& engineType) {
        engine = engineType;
    }

    void setSeats(int seatCount) {
        seats = seatCount;
    }

    void setColor(const string& carColor) {
        color = carColor;
    }

    // Display the details of the car
    void show() const {
        cout << "Car with " << engine << " engine, " << seats << " seats, color " << color << endl;
    }
};

```

Without the Builder Pattern

```
// Helper functions to create specific types of cars without a builder

// Function to create a sports car
Car createSportsCar() {
    Car car;
    car.setEngine("V8");
    car.setSeats(2);
    car.setColor("Red");
    return car;
}

// Function to create a family car
Car createFamilyCar() {
    Car car;
    car.setEngine("V6");
    car.setSeats(5);
    car.setColor("Blue");
    return car;
}

```

Without the Builder Pattern

```cpp

int main() {
    // Directly create a sports car without a builder
    Car sportsCar = createSportsCar();
    sportsCar.show(); // Outputs: Car with V8 engine, 2 seats, color Red

    // Directly create a family car without a builder
    Car familyCar = createFamilyCar();
    familyCar.show(); // Outputs: Car with V6 engine, 5 seats, color Blue

    return 0;
}

```

Without the Builder Pattern

- The `Car` class has properties like engine, seats, and color.
- A constructor to set these properties to default or specified values and setter methods (setEngine, setSeats, setColor) to set each attribute manually.
- Each helper function creates a specific type of Car with predefined configurations.

Without the Builder Pattern

- `createSportsCar()` sets the car's engine to "V8", seats to 2, and color to "Red".

- `createFamilyCar()` sets the engine to "V6", seats to 5, and color to "Blue".

- `main()` directly calls these helper functions to create specific types of cars.

Without the Builder Pattern

- Limited Flexibility: Adding configurations requires more helper functions or direct modifications.
- Maintenance Complexity: Code can become bloated as configurations increase.
- No Incremental Setup: Lacks step-by-step construction, making complex setups harder to manage.

Telescoping Constructors

- Pattern of creating multiple overloaded constructors, each adding parameters.

- Provides flexible initialization for complex objects.

- Each constructor calls a more detailed constructor with extra parameters.

Telescoping Constructors

```cpp
#include <iostream>
#include <string>

using namespace std;

// Car class representing the product with telescoping constructors
class Car {
private:
    string engine;
    int seats;
    string color;
```

Telescoping Constructors

```cpp
public:
    // Basic constructor with no parameters (default values)
    Car() : engine(""), seats(0), color("") {}

    // Constructor with engine only
    Car(const string& engineType) : engine(engineType), seats(0), color("") {}

    // Constructor with engine and seats
    Car(const string& engineType, int seatCount) : engine(engineType), seats(seatCount), color("") {}

    // Constructor with engine, seats, and color
    Car(const string& engineType, int seatCount, const string& carColor)
        : engine(engineType), seats(seatCount), color(carColor) {}

    // Display the details of the car
    void show() const {
        cout << "Car with " << engine << " engine, " << seats << " seats, color " << color << endl;
    }
};
```

Telescoping Constructors

```
int main() {
    // Using different constructors to create cars with varying configurations

    // Basic car with no specific configurations
    Car basicCar;
    basicCar.show(); // Outputs: Car with  engine, 0 seats, color

    // Car with only engine specified
    Car engineOnlyCar("V6");
    engineOnlyCar.show(); // Outputs: Car with V6 engine, 0 seats, color

    // Car with engine and seats specified
    Car engineSeatsCar("V8", 2);
    engineSeatsCar.show(); // Outputs: Car with V8 engine, 2 seats, color

    // Fully configured car with engine, seats, and color
    Car fullyConfiguredCar("V8", 2, "Red");
    fullyConfiguredCar.show(); // Outputs: Car with V8 engine, 2 seats, color Red

    return 0;
}
```

Telescoping Constructors

- Car Class Constructors: Multiple constructors for varied setups.
  - Default Constructor: Sets default values.
  - Engine Constructor: Only engine specified.
  - Engine and Seats: Takes engine and seat count.
  - Full Config Constructor: engine, seats, and color.

- Main Function:
  - Basic Car: All values default.
  - Engine Only: Only engine specified.
  - Engine and Seats: Engine and seats specified.
  - Fully Configured: Engine, seats, and color specified.

Telescoping Constructors

- Advantages
  - Simple for Few Params: Works well with limited options.
  - Direct Initialization: No setters or helpers needed.

- Disadvantages
  - Hard to Maintain: More parameters lead to many constructors.
  - Error-Prone: Easy to misorder parameters.
  - Not Self-Documenting: Unclear which values are set.

Builder Pattern

```cpp
#include <iostream>
#include <string>
using namespace std;

// Product class that represents the complex object
class Car {
public:
    string engine;
    int seats;
    string color;

    void show() {
        cout << "Car with " << engine << " engine, " << seats << " seats, color " << color << endl;
    }
};
```

Product (Car class)

- This class has various attributes like engine, seats, and color.
- A show method displays the configured car's properties.

Builder Pattern

```cpp
// Builder interface or abstract class
class CarBuilder {
public:
    virtual CarBuilder* setEngine(const string& engineType) = 0;
    virtual CarBuilder* setSeats(int numberOfSeats) = 0;
    virtual CarBuilder* setColor(const string& carColor) = 0;
    virtual Car* build() = 0;
    virtual ~CarBuilder() = default;
};

```

Builder Interface

- Defines virtual methods for setting each part of the Car.
- The build method is used to retrieve the fully constructed Car object.

Builder Pattern

```cpp
// Concrete Builder for Car
class ConcreteCarBuilder : public CarBuilder {
private:
    Car* car;

public:
    ConcreteCarBuilder() {
        car = new Car();
    }
```

Builder Pattern

```
    CarBuilder* setEngine(const string& engineType) override {
        car->engine = engineType;
        return this;
    }
    CarBuilder* setSeats(int numberOfSeats) override {
        car->seats = numberOfSeats;
        return this;
    }

    CarBuilder* setColor(const string& carColor) override {
        car->color = carColor;
        return this;
    }

    Car* build() override {
        return car;
    }
};
```

Concrete Builder

- The ConcreteCarBuilder has a dependency on Car since it relies on Car to produce a final product.

- This dependency means CarBuilder needs Car to function but doesn’t strictly own or control its lifecycle.

Builder Pattern

```cpp
// Optional Director class
class CarDirector {
private:
    CarBuilder* builder;

public:
    CarDirector(CarBuilder* builder) : builder(builder) {}

    Car* constructSportsCar() {
        return builder->setEngine("V8")->setSeats(2)->setColor("Red")->build();
    }

    Car* constructFamilyCar() {
        return builder->setEngine("V6")->setSeats(5)->setColor("Blue")->build();
    }
};

```

Director Class

- It defines methods for building specific types of cars (like sports cars and family cars).

- Uses the builder to set specific configurations.

Director Class

- The Director and Builder have an aggregation relationship, where the Director holds a reference to the Builder but does not own it.

- This means the Builder is created and managed externally, allowing the Director to use it temporarily for construction without controlling its lifecycle.

Builder Pattern

```cpp
int main() {
    // Using the builder directly
    CarBuilder* builder = new ConcreteCarBuilder();
    Car* customCar = builder->setEngine("Electric")->setSeats(4)->setColor("Green")->build();
    customCar->show();

    // Using the director to create predefined types
    CarDirector director(builder);
    Car* sportsCar = director.constructSportsCar();
    sportsCar->show();

    Car* familyCar = director.constructFamilyCar();
    familyCar->show();

    delete builder;
    delete customCar;
    delete sportsCar;
    delete familyCar;

    return 0;
}
```

Builder Pattern

- First, we use the builder to create a custom car with specific properties.
- Then, we use the director to build predefined types of cars: sports and family.

Builder Pattern without Pointers

```cpp
#include <iostream>
#include <string>

using namespace std;

// Product class representing the Car
class Car {
public:
    string engine;
    int seats;
    string color;

    // Display the car's details
    void show() const {
        cout << "Car with " << engine << " engine, " << seats << " seats, color " << color << endl;
    }
};

```

Method Chaining

- When you're not using pointers,
  - Method chaining can still work if you return a reference to the current object (using *this).

- Each setter method returns the object itself
  - Allowing the next method to be called directly on the returned reference.

Builder Pattern without Pointers

```cpp
// CarBuilder class for step-by-step construction of Car objects
class CarBuilder {
private:
    Car car; // Directly hold an instance of Car

public:
    // Setter method for engine type, returning a reference to CarBuilder for chaining
    CarBuilder& setEngine(const string& engineType) {
        car.engine = engineType;
        return *this; // Return current CarBuilder for method chaining
    }

    // Setter method for seat count, returning a reference to CarBuilder for chaining
    CarBuilder& setSeats(int seatCount) {
        car.seats = seatCount;
        return *this;
    }
```

Builder Pattern without Pointers

```

    // Setter method for color, returning a reference to CarBuilder for chaining
    CarBuilder& setColor(const string& carColor) {
        car.color = carColor;
        return *this;
    }

    // Final build method to return the fully constructed Car object
    Car build() const {
        return car; // Return a copy of the constructed Car
    }
};

```

Method Chaining

- Returning a reference to the class (using *this) allows us to execute multiple methods on the same instance in a single, chained statement.

Builder Pattern without Pointers

```cpp
// Optional Director class to define specific car configurations
class CarDirector {
private:
    CarBuilder builder; // No pointers, just hold the CarBuilder object directly

public:
    // Constructor that initializes CarDirector with a CarBuilder
    CarDirector(const CarBuilder& builder) : builder(builder) {}

    // Method to construct a sports car
    Car constructSportsCar() {
        return builder.setEngine("V8").setSeats(2).setColor("Red").build();
    }

    // Method to construct a family car
    Car constructFamilyCar() {
        return builder.setEngine("V6").setSeats(5).setColor("Blue").build();
    }
};
```

Director Class

- `CarDirector` holds a reference to the CarBuilder
  - Lets it use `CarBuilder’s` methods to configure the Car object.

- `CarDirector` doesn’t create or manage the Car object directly.

- Calls methods on `CarBuilder (like setEngine, setSeats, and setColor),` which in turn configure the Car object.

Builder Pattern without Pointers

```cpp
int main() {
    // Create a CarBuilder instance (no pointers needed)
    CarBuilder builder;

    // Use CarBuilder directly to build a custom car
    Car customCar = builder.setEngine("Electric").setSeats(4).setColor("Green").build();
    customCar.show(); // Outputs: Car with Electric engine, 4 seats, color Green

    // Use CarDirector to build predefined car types
    CarDirector director(builder);
    Car sportsCar = director.constructSportsCar();
    sportsCar.show(); // Outputs: Car with V8 engine, 2 seats, color Red

    Car familyCar = director.constructFamilyCar();
    familyCar.show(); // Outputs: Car with V6 engine, 5 seats, color Blue

    return 0;
}

```

Builder Pattern without Pointers

- Car (Product): Defines attributes (engine, seats, color) and a show method for configuration display.
- CarBuilder: Holds a Car instance; each set method updates Car and returns *this for chaining; build() returns a configured Car.
- CarDirector: Uses CarBuilder to create preset car configurations (constructSportsCar, constructFamilyCar).
- Main: Demonstrates custom car creation with CarBuilder and predefined configurations with CarDirector.

Builder Pattern without Pointers

- Positives:
  - Flexibility: Ideal for complex, step-by-step construction.
  - Readability: Chained methods clarify configuration.
  - Encapsulation: Hides complex creation logic.
  - Immutability: Creates fully-configured objects.

- Negatives:
  - Complexity: More classes, especially for simple objects.
  - Memory Overhead: Builder stores instance directly.
  - Overkill for Simple Objects: Adds unnecessary structure.

Conclusion

- Creational Design Patterns: Builder Pattern
  - Builder Pattern Overview
  - Without Builder Pattern
  - Telescoping Constructors
  - Builder Pattern with Pointers
  - Builder Pattern without Pointers
  - Pros and Cons of Builder Pattern