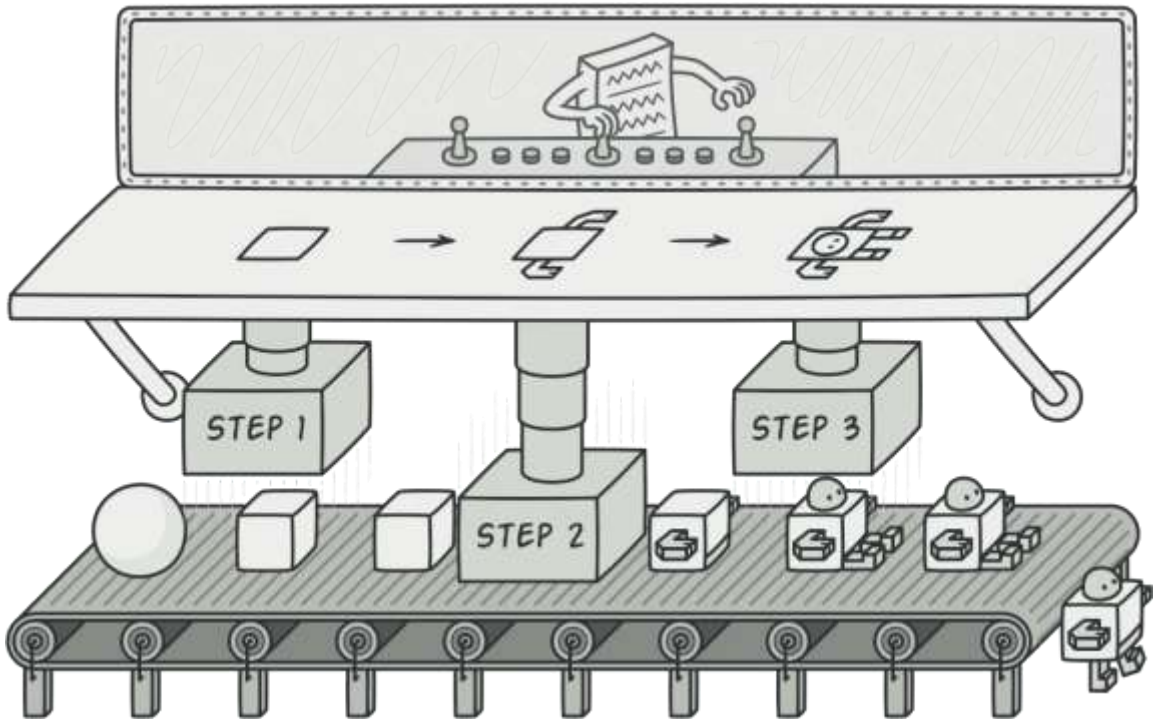
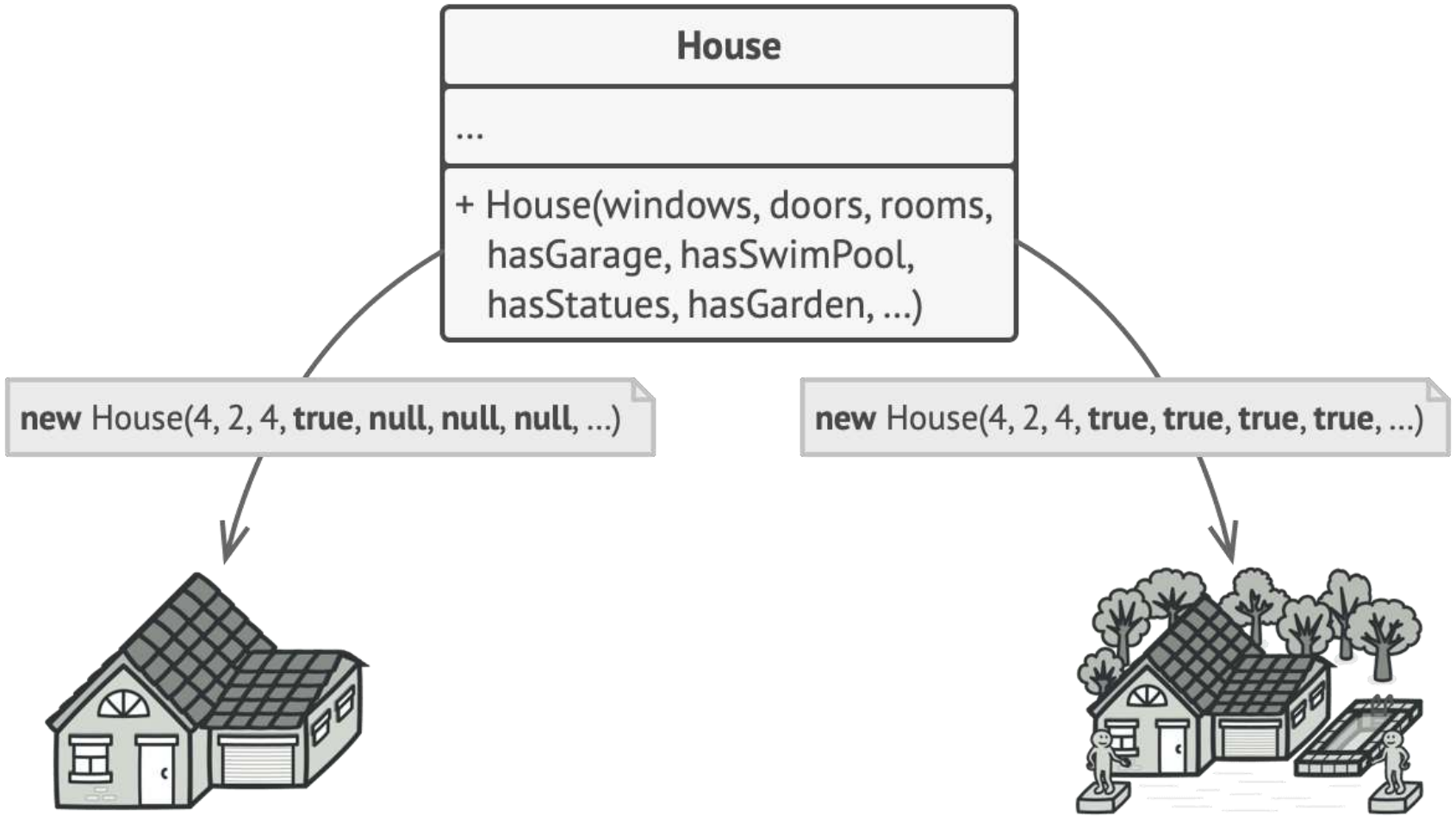
```
/**
 * Step by step construction of a complex object
 */
```

Builder::DesignPattern

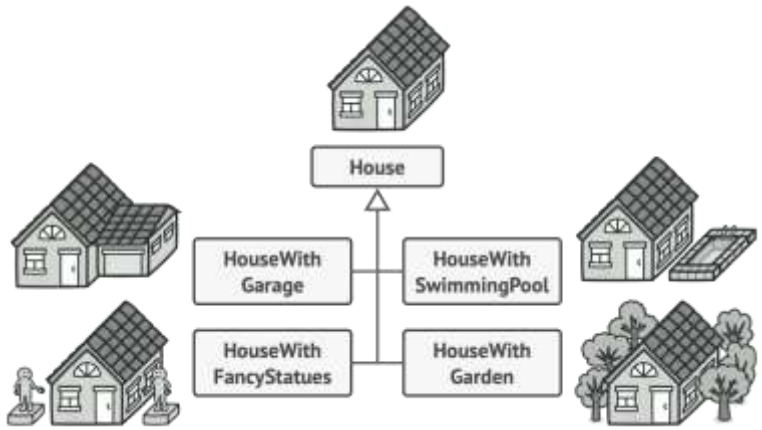


Problem::The common way to create a big objects with optional arguments



Problem::Why Do We Need a Builder?

1. Too Many arguments in constructor
2. Some parameters might be optional
3. Heavy and complex objects




```
class House {
private:
    int area;
    int bedrooms;
    Garden garage;
    Pool pool;
    Garden garden;
    ...
    Balcony balcony;
    Terrace terrace;

public:
   House(int area, int bedrooms, Garage garage, Pool po
        this.area = area;
        ...
        this.garden = garden;
        this.balcony = balcony;
        this.terrace = terrace;
   }
}
```

Solution?

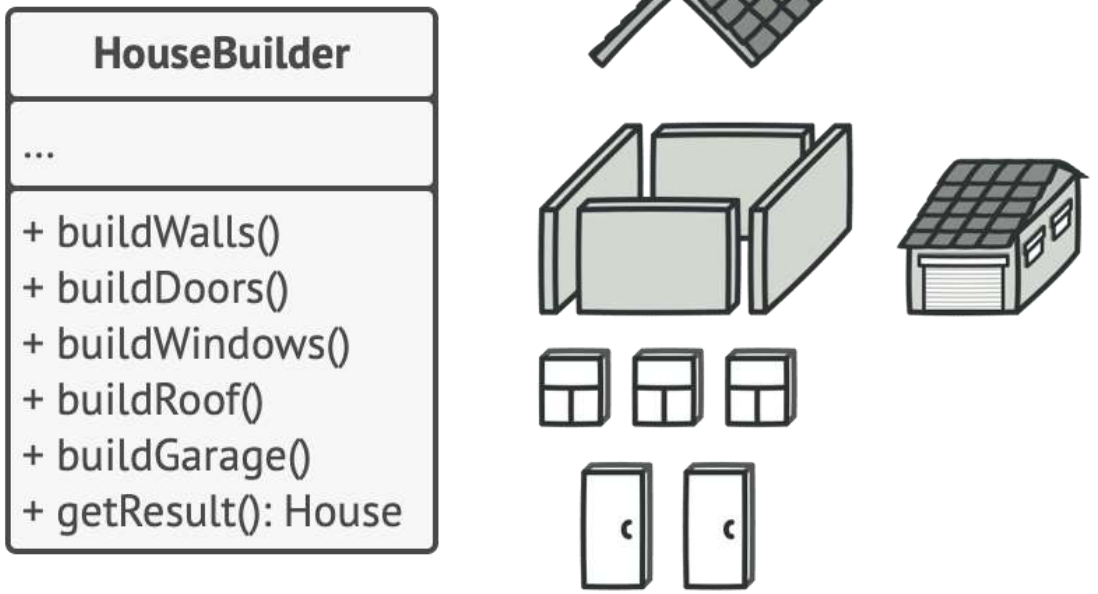
```
...
public:
    void set_area(int area) {
        this.area = area;
    }
    void set_bedrooms(int bedrooms) {
        this.bedrooms = bedrooms;
    }
    ...

int main() {
    House house = House();

    // Is it right way?
    House.set_area(10);
    House.set_bedrooms(200);
    ...
}
```

What if?::methods chaining

```
...
public:
    House set_area(int area) {
        this.area = area;
        return this*
    }
    House set_bedrooms(int bedrooms) {
        this.bedrooms = bedrooms;
        return this*
    }
    ...

int main() {
    // What about immutable objects?
    House house = House()
                        .set_area(10);
                        .set_bedrooms(200);
                        ...
}
```

Solution::Builder

Executing a series of these steps on a builder object, that creates complex object



FluentBuilder::Definition

The Interface that allows us to chain method calls together in a readable and intuitive manner

```
Email::Builder()
    .from("me@mail.com")
    .to("you@mail.com")
    .subject("C++ builders")
    .body("I like this API, don't you?");
```

Real example::command line parser API

```
Parser parser = Parser();
// Could it be implemented?
parser
    .set_option<std::string>("format")
    .set_alternative_name("f")
    .set_description("Specify output format, possibly overriding the format
                      specified in the environment variable TIME.");
```

Real example::Product

```
template <typename T>
class Option {
private:
    std::vector<std::string> _names;
    T _value;
    std::string _description;
    bool _required = false;
    bool _positional = false;
    std::vector<std::string> _args;
    std::vector<std::string> _dependencies;
}
public:
    …
```

Real example::fluent builder

```
template <typename T>
class OptionBuilder {
private:
   std::shared_ptr<Option<T>> option;
public:
   OptionBuilder() { … }
   OptionBuilder set_name() { … };
   OptionBuilder set_description(const std::string& description) { … };
   OptionBuilder set_required() { … };
   OptionBuilder set_alternative_name(const std::string& alternativeName) { … };
   OptionBuilder set_positional() { … };
   OptionBuilder set_default(const T& value) {… };
   OptionBuilder set_dependency(const std::string& dependency) {…};
};
```

Real example::Readable user API

```
Class Parser {
    …
    /** Adds option to parser and create new builder with it */
   template <typename T>
   OptionBuilder<T> set_option(const std::string& name) {…};
}


Parser parser = Parser();
parser
   .set_option<std::string>("format")
   .set_alternative_name("f")
   .set_description("Specify output format, possibly overriding the format
                     specified in the environment variable TIME.");
```

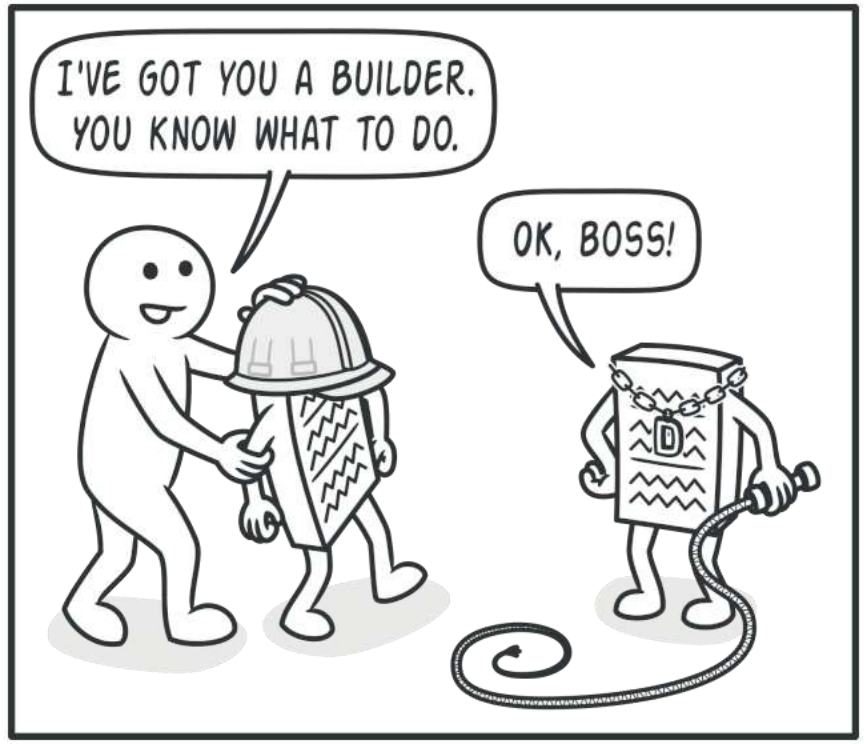
Pass a builder::feature

```
void add_addresses(EmailBuilder& builder)
{
   builder
         .from("me@mail.com")
         .to("you@mail.com");
}

void compose_mail(EmailBuilder& builder)
{
   builder
         .subject("I know the subject")
         .body("And the body.");
}
```

```
int main()
{
   EmailBuilder builder;
   add_addresses(builder);
   compose_mail(builder);

   Email mail = builder;
   cout << mail << endl;
}
```

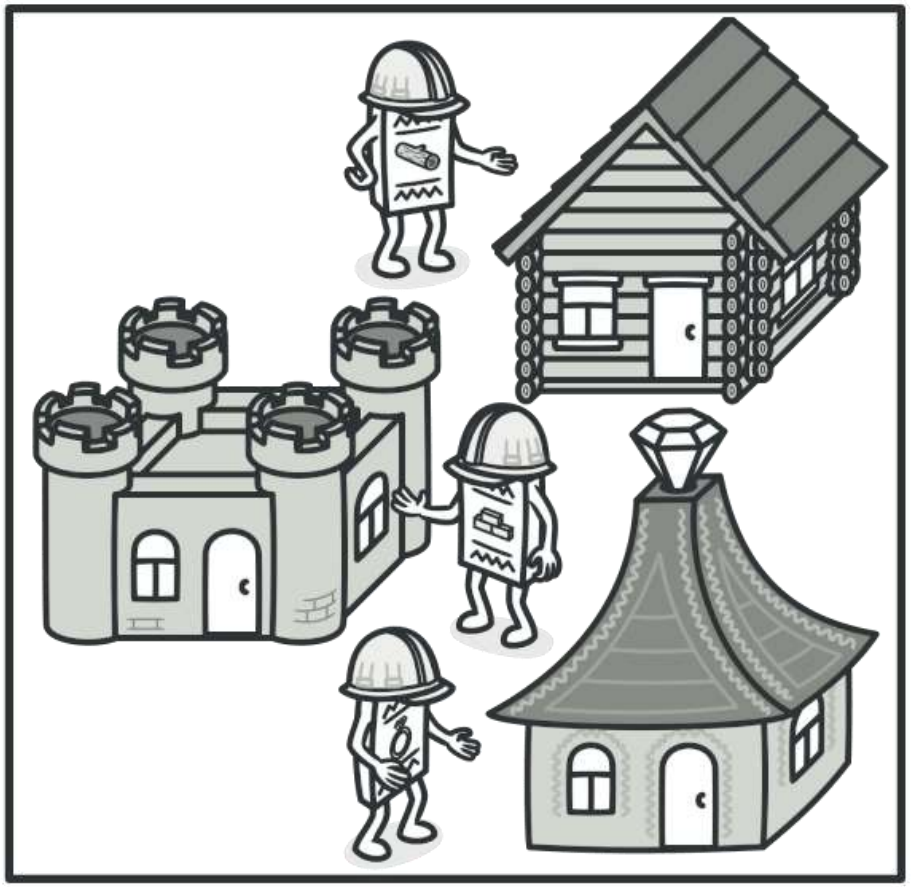
Director

- The director class defines the order in which to execute the building steps
- Having a director class in your program isn’t strictly necessary



Concrete builders

Several different builder classes that implement the similar set of building steps, but in a different manner.



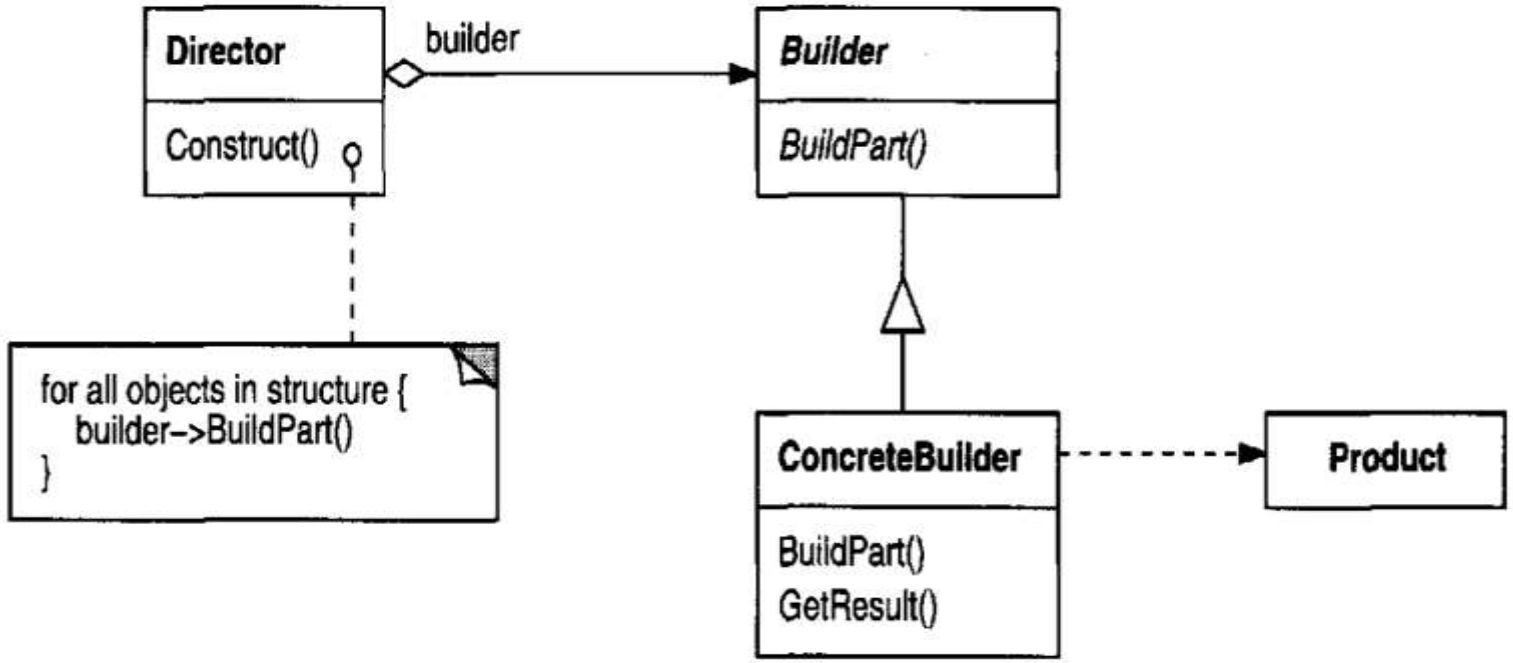
Other participants

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

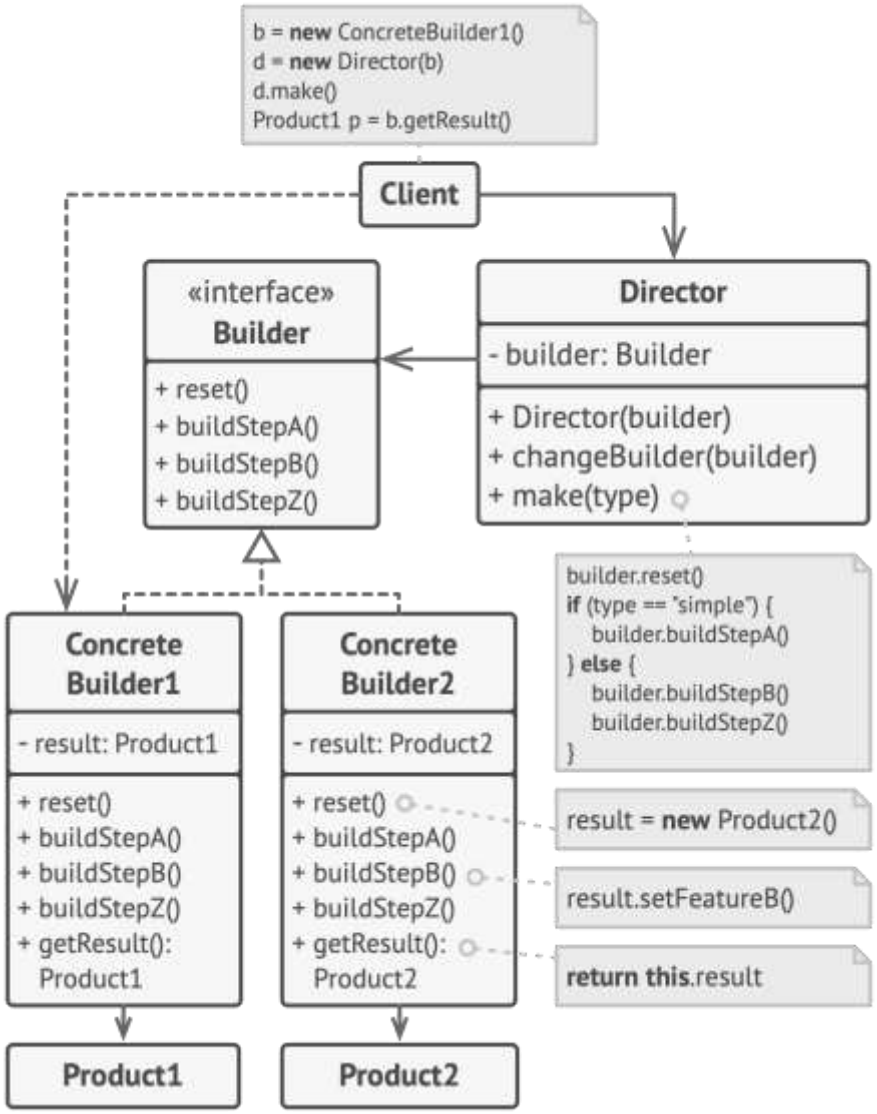
Concrete Builder - constructs and assembles parts of the product by *implementing* the Builder interface.

Director - constructs an object using the *Builder interface*

Product - represents the complex object under construction..



Structure


b = new ConcreteBuilder1()
d = new Director(b)
d.make()
Product1 p = b.getResult()
Client
«interface»
Builder
+ reset()
+ buildStepA()
+ buildStepB()
+ buildStepZ()
Director
- builder: Builder
+ Director(builder)
+ changeBuilder(builder)
+ make(type)
builder.reset()
if (type == "simple") {
builder.buildStepA()
} else {
builder.buildStepB()
builder.buildStepZ()
}
Concrete
Builder1
- result: Product1
+ reset()
+ buildStepA()
+ buildStepB()
+ buildStepZ()
+ getResult():
Product1
Concrete
Builder2
- result: Product2
+ reset()
+ buildStepA()
+ buildStepB()
+ buildStepZ()
+ getResult():
Product2
result = new Product2()
result.setFeatureB()
return this.result
Product1
Product2

Implementation::Builder interface

```
class Car {}

// Abstract Builder interface defining the building steps.
class Builder {
public:
    virtual void reset() = 0;
    virtual void setSeats(int number) = 0;
    virtual void setEngine(const std::string& engineType) = 0;
    virtual void setTripComputer(bool hasTripComputer) = 0;
    virtual void setGPS(bool hasGPS) = 0;
    virtual ~Builder() {}
};
```

Implementation::Concrete builder

```
// Concrete Builder for Car
class CarBuilder : public Builder {
private:
    Car car;
public:
    CarBuilder(){
        reset();
    }
    void reset() override {}
    void setSeats(int number) override {}
    void setEngine(const std::string& engineType) override {}
    void setTripComputer(bool hasTripComputer) override {}
    void setGPS(bool hasGPS) override {}
    Car getProduct() {
        Car product = car;
        reset();  // Prepare builder for next build.
        return product;
    }
};
```

Implementation::Director

```
// Director class to encapsulate the construction process.
class Director {
public:
    void constructSportsCar(Builder& builder) {
        builder.reset();
        builder.setSeats(2);
        builder.setEngine("SportEngine");
        builder.setTripComputer(true);
        builder.setGPS(true);
    }

    // Add more methods to construct different types of cars.
};
```

Implementation::Usage

```
// Client code
int main() {

    Director director;
    CarBuilder carBuilder;
    director.constructSportsCar(carBuilder);
    Car car = carBuilder.getProduct();

    // car and manual objects are now built and ready for use.
    return 0;
}
```

Applicability

- Use the Builder pattern to get rid of a “telescoping constructor”

```
class Pizza {
    Pizza(int size) { ... }
    Pizza(int size, boolean cheese) { ... }
    Pizza(int size, boolean cheese, boolean pepperoni) { ... }
    // ...
}
```

- Use the Builder pattern when you want your code to be able to create different representations of some product (for example, stone and wooden houses).

- Use the Builder to construct complex objects.

Advantages

- You can construct objects step-by-step
- You can reuse the same construction code when building various representations of products.
- Single Responsibility Principle. You can isolate complex construction code from the business logic of the product.

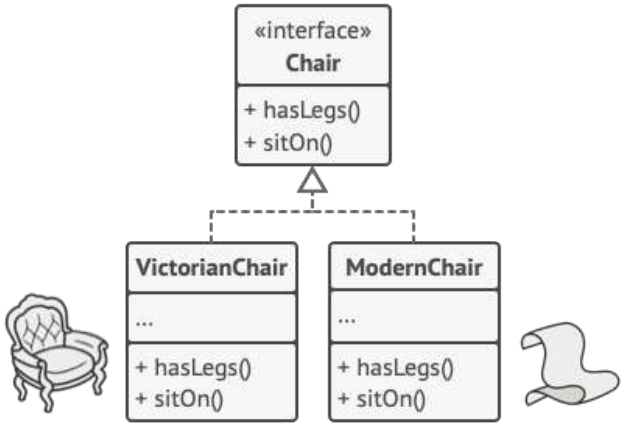
Disadvantages

- The overall complexity of the code increases since the pattern requires creating multiple new classes.

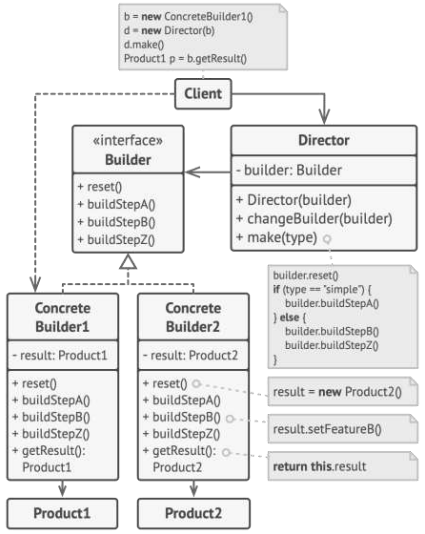
Relations with Other Patterns

Factory methods: Abstract Factory, Prototype, or Builder

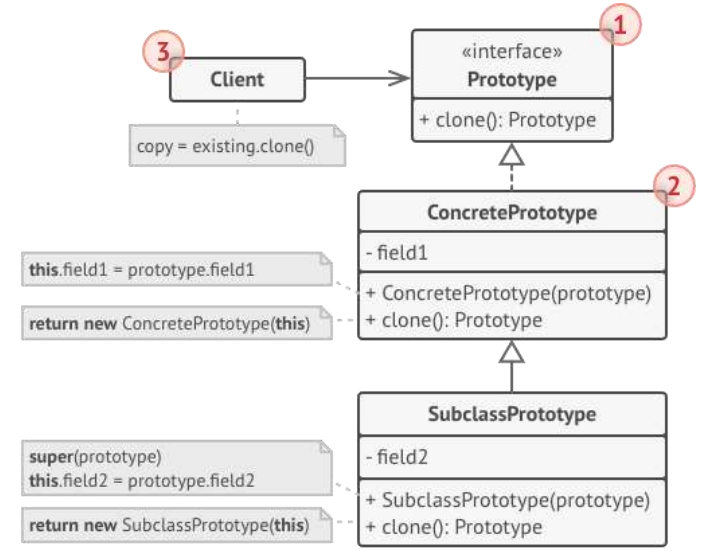
- **Builder** focuses on constructing **complex objects** step by step.
- **Abstract Factory** specializes in creating **families** of related objects.
- **Prototype** lets you **copy** existing objects without making your code dependent on their classes.



Abstract Factory



Builder



Prototype

```
Thank::you for your attention()
```