Builder Pattern

Wednesday, September 2, 2020 9:40 AM

Builder is a creational design pattern, which allows constructing complex objects step by step

- this pattern is especially useful when you need to create an object w/ lots of possible configuration options

- class w/ a single creation method & several methods to configure the resulting object
  - Builder methods often support chaining

```
someBuilder->setValueA(1)->setValueB(2)->create()
```

Conceptual Example in C++

- to answer these questions
  - ? What classes does it consist of?
  - ? What role do these classes play?
  - ? In what way the elements of the pattern are related?

```
/**
 * The Builder interface specifies methods for creating the different parts of
 * the Product objects.
 */
class Builder{
    public:
    virtual ~Builder(){}
```

```
/**
 * The Builder interface specifies methods for creating the different parts of
 * the Product objects.
 */
class Builder{
    public:
    virtual ~Builder(){}
    virtual void ProducePartA() const =0;
    virtual void ProducePartB() const =0;
    virtual void ProducePartC() const =0;
};
```

```
/**
 * The Concrete Builder classes follow the Builder interface and provide
 * specific implementations of the building steps. Your program may have several
 * variations of Builders, implemented differently.
 */
class ConcreteBuilder1 : public Builder{
    private:

    Product1* product;
    /**
     * A fresh builder instance should contain a blank product object, which is
     * used in further assembly.
     */
    public:

    ConcreteBuilder1(){
        this->Reset();
    }

    ~ConcreteBuilder1(){
        delete product;
    }

    void Reset(){
        this->product= new Product1();
    }
```

simple class w/ std::vector<std::string> parts_; public data type

- fresh product instance

```
        this->product= new Product1();
    }
    /**
     * All production steps work with the same product instance.
     */

    void ProducePartA()const override{
        this->product->parts_.push_back("PartA1");
    }

    void ProducePartB()const override{
        this->product->parts_.push_back("PartB1");
    }

    void ProducePartC()const override{
        this->product->parts_.push_back("PartC1");
    }
```

following delivery of end result to client, a builder instance is expected to be ready to start producing another product

- usual to call reset method
- can wait for an explicit reset call

```
/**
 * Please be careful here with the memory ownership. Once you call
 * GetProduct the user of this function is responsable to release this
 * memory. Here could be a better option to use smart pointers to avoid
 * memory leaks
 */

Product1* GetProduct() {
    Product1* result= this->product;
    this->Reset();
    return result;
```

↳ C++ 14

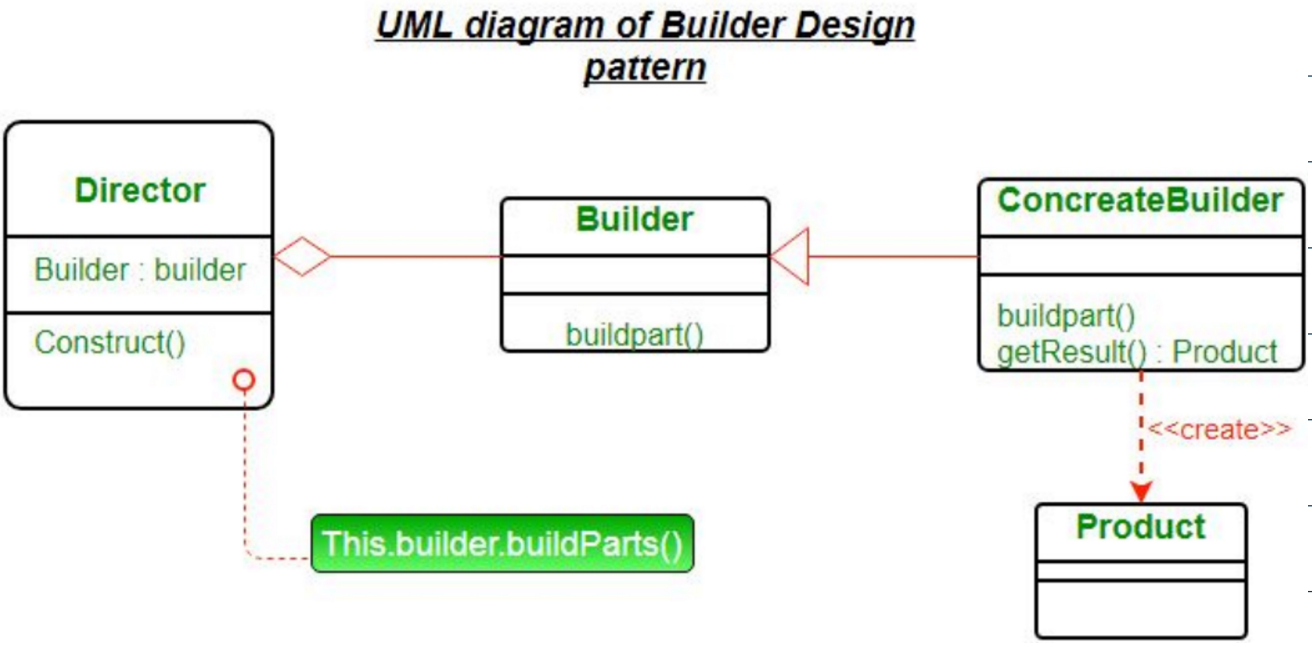
```
    this->Reset();
    return result;
}
```

Another take on Builder Design Pattern

definition: separate the construction of a complex object from its representation so that the same construction process can create different representations

- construct a complex object step-by-step

UML Diagram of Builder Design Pattern



Product - the product class defines the type of the complex object that is to be generated by the builder pattern

Builder - abstract base class defines all of the steps that must be taken in order to correctly create a product

- each step is generally abstract as actual

- each step is generally abstract as actual functionality of builder carried out in the concrete subclasses

Concrete Builder - may be a # of concrete builder classes

Director - controls the algorithm that generates the final product object

EXAMPLE

Home = final end product (object) that is to be returned as output of the construction process

```java
interface HousePlan
{
    public void setBasement(String basement);

    public void setStructure(String structure);

    public void setRoof(String roof);

    public void setInterior(String interior);
}

class House implements HousePlan
{

    private String basement;
    private String structure;
    private String roof;
    private String interior;

    public void setBasement(String basement)
    {
        this.basement = basement;
    }

    public void setStructure(String structure)
    {
        this.structure = structure;
    }

    public void setRoof(String roof)
    {
        this.roof = roof;
    }

    public void setInterior(String interior)
    {
        this.interior = interior;
    }

}
```



```java
interface HouseBuilder
{

    public void buildBasement();

    public void buildStructure();

    public void bulidRoof();

    public void buildInterior();

    public House getHouse();
}

class IglooHouseBuilder implements HouseBuilder
{
    private House house;

    public IglooHouseBuilder()
    {
        this.house = new House();
    }

    public void buildBasement()
    {
        house.setBasement("Ice Bars");
    }

    public void buildStructure()
    {
        house.setStructure("Ice Blocks");
    }

    public void buildInterior()
    {
        house.setInterior("Ice Carvings");
    }

    public void bulidRoof()
    {
        house.setRoof("Ice Dome");
    }

    public House getHouse()
    {
        return this.house;
    }
}
```

Advantages of Builder Design Pattern

- parameters to the constructor are reduced & are provided in highly readable method calls
- minimizing # of parameters in constructor
- object always instantiated in complete state
- w/out much complex logic in object building process

Disadvantages

- number of lines of code increase to double
- requires creating a separate ConcreteBuilder for each different type of Product

Design Concept

How to implement builder pattern in a C++ class that exports to JS?

- separate job builder header & source files...
  - JS driving code:

```
var job = new empire.jobObject();
job.setEmitterLoc();
job.setReceiverLoc();
....
```

all params set ↓

```
empire.calePropLoss(job);
```