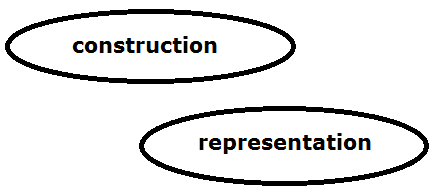
**Builder**

Builder Creational Pattern is used to separate the construction of a complex object from its representation so that the same construction process can create different objects representations.



Builder Creational Pattern

# Problem

We want to construct a complex object, however we do not want to have a complex constructor member or one that would need many arguments.

# Solution

Define an intermediate object whose member functions define the desired object part by part before the object is available to the client. Builder Pattern lets us defer the construction of the object until all the options for creation have been specified.

Builder design pattern describes how to solve such problems:

* Encapsulate creating and assembling the parts of a complex object in a separate Builder object.
* A class delegates object creation to a Builder object instead of creating the objects directly.

# Advantages

1. Allows you to vary a product’s internal representation
2. Encapsulates code for construction and representation
3. Provides control over steps of construction process

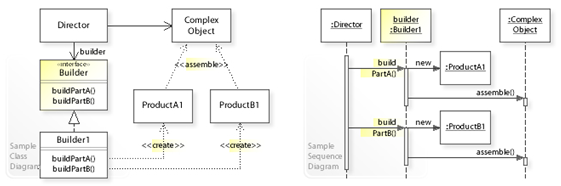
# Disadvantages

1. Requires creating a separate ConcreteBuilder for each different type of product
2. Requires the builder classes to be mutable
3. Data members of class aren't guaranteed to be initialized
4. Dependency injection may be less supported

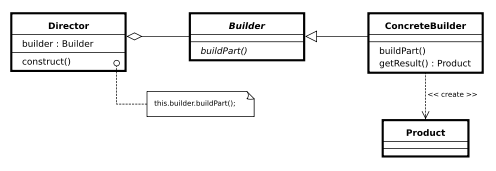


Builder Creational Patterns

# Structure



UML class and sequence diagram for the Builder design pattern



Class diagram

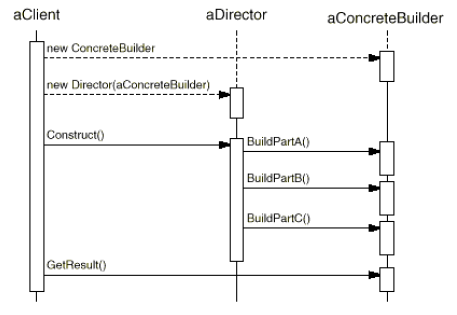
**Builder:** Abstract interface for creating objects (product)

**ConcreteBuilder:** Provides implementation for Builder. It is an object able to construct other objects. Constructs and assembles parts to build the objects

**Director:** constructs an object using the Builder interface

**Product:** represents the complex object under construction. ConcreteBuilder builds the product's internal representation and defines the process by which it's assembled

Includes classes that define the constituent parts, including interfaces for assembling the parts into the final result

****

Interaction Diagram

# Example

#include <string>

#include <iostream>

#include <memory>

using namespace std;

// "Product"

class Pizza {

public:

void setDough(const string& dough) {

m\_dough = dough;

}

void setSauce(const string& sauce) {

m\_sauce = sauce;

}

void setTopping(const string& topping) {

m\_topping = topping;

}

void open() const {

cout << "Pizza with " << m\_dough << " dough, " << m\_sauce << " sauce and " << m\_topping << " topping. Mmm." << endl;

}

private:

string m\_dough;

string m\_sauce;

string m\_topping;

};

// "Abstract Builder"

class PizzaBuilder {

public:

virtual ~PizzaBuilder() {};

Pizza\* getPizza() {

return m\_pizza.release();

}

void createNewPizzaProduct() {

m\_pizza = make\_unique<Pizza>();

}

virtual void buildDough() = 0;

virtual void buildSauce() = 0;

virtual void buildTopping() = 0;

protected:

unique\_ptr<Pizza> m\_pizza;

};

//----------------------------------------------------------------

class HawaiianPizzaBuilder : public PizzaBuilder {

public:

virtual ~HawaiianPizzaBuilder() {};

virtual void buildDough() {

m\_pizza->setDough("cross");

}

virtual void buildSauce() {

m\_pizza->setSauce("mild");

}

virtual void buildTopping() {

m\_pizza->setTopping("ham+pineapple");

}

};

class SpicyPizzaBuilder : public PizzaBuilder {

public:

virtual ~SpicyPizzaBuilder() {};

virtual void buildDough() {

m\_pizza->setDough("pan baked");

}

virtual void buildSauce() {

m\_pizza->setSauce("hot");

}

virtual void buildTopping() {

m\_pizza->setTopping("pepperoni+salami");

}

};

//----------------------------------------------------------------

class Cook {

public:

void openPizza() {

m\_pizzaBuilder->getPizza()->open();

}

void makePizza(PizzaBuilder\* pb) {

m\_pizzaBuilder = pb;

m\_pizzaBuilder->createNewPizzaProduct();

m\_pizzaBuilder->buildDough();

m\_pizzaBuilder->buildSauce();

m\_pizzaBuilder->buildTopping();

}

private:

PizzaBuilder\* m\_pizzaBuilder;

};

int main() {

Cook cook;

HawaiianPizzaBuilder hawaiianPizzaBuilder;

SpicyPizzaBuilder spicyPizzaBuilder;

cook.makePizza(&hawaiianPizzaBuilder);

cook.openPizza();

cook.makePizza(&spicyPizzaBuilder);

cook.openPizza();

}

# END