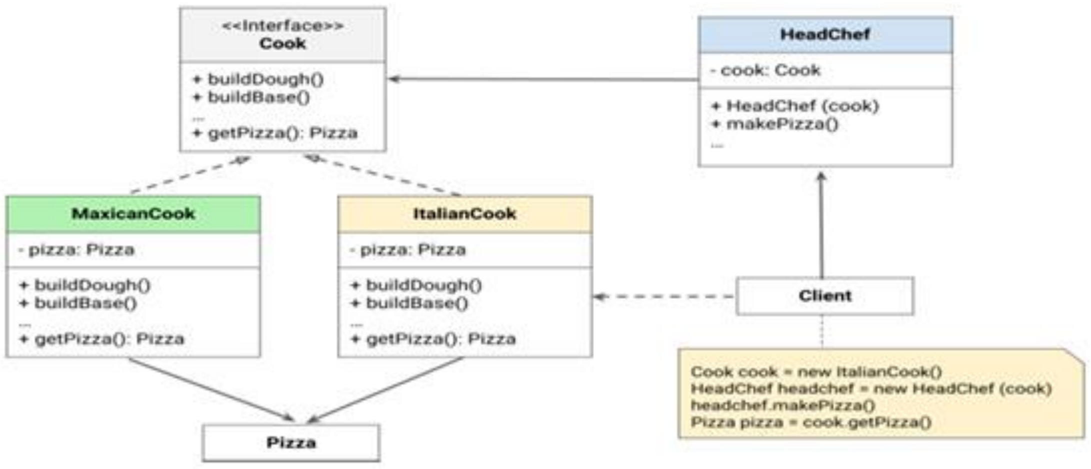
Problem No: 03
Problem Name: Implementation and Design of Builder Design Pattern

Objectives: The objective of this lab report is to implement and demonstrate the Builder Design Pattern in C++ for a pizza ordering system. The goal is to show how the pattern enables step-by-step construction of a customizable pizza while maintaining flexibility, scalability, and code readability. This implementation highlights the advantages of using the Builder Pattern in managing complex object creation in object-oriented design.

Theory:

- Declare common construction steps in the builder interface for building all available products.
- Create concrete builder classes for each product representation and implement their construction steps.
- Add a separate method inside each concrete builder to retrieve the output of the construction, We can not declare this method inside the builder interface because different builders may construct products that do not have a common interface . Therefore , we do not know the return type for such a method. Note: If working with products form a single hierarchy , we can simply add this method to the base interface.
- Create a director class and encapsulate various ways to construct a product using the builder object.
- In the client code , create both the builder and director objects , and pass the builder object to the director to start the construction process.

UML class design:



Implementation in C++:

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

class Pizza {
private:
   string size;
   string crust;
   string cheese;
   string toppings;
   string sauce;

public:
   void setSize(const string& s) { size = s; }
   void setCrust(const string& c) { crust = c; }
   void setCheese(const string& ch) { cheese = ch; }
   void setToppings(const string& t) { toppings = t; }
   void setSauce(const string& s) { sauce = s; }

   void display() {
      cout << "Pizza Details:" << endl;
      cout << "Size: " << size << endl;
      cout << "Crust: " << crust << endl;
      cout << "Cheese: " << cheese << endl;
      cout << "Toppings: " << toppings << endl;
      cout << "Sauce: " << sauce << endl;
   }
};

class PizzaBuilder {
public:
   virtual void buildSize() = 0;
   virtual void buildCrust() = 0;
   virtual void buildCheese() = 0;
   virtual void buildToppings() = 0;
   virtual void buildSauce() = 0;
   virtual Pizza* getPizza() = 0;
};
class CustomPizzaBuilder : public PizzaBuilder {
private:
   Pizza* pizza;
public:
   CustomPizzaBuilder() { pizza = new Pizza(); }
   void buildSize() override { pizza->setSize("Medium"); }
```

```
    void buildCrust() override { pizza->setCrust("Thin Crust"); }
    void buildCheese() override { pizza->setCheese("Mozzarella"); }
    void buildToppings() override { pizza->setToppings("Pepperoni & Olives"); }
    void buildSauce() override { pizza->setSauce("Tomato Basil"); }

    Pizza* getPizza() override { return pizza; }
};

class PizzaChef {
private:
    PizzaBuilder* pizzaBuilder;

public:
    void setPizzaBuilder(PizzaBuilder* builder) { pizzaBuilder = builder; }

    Pizza* makePizza() {
        pizzaBuilder->buildSize();
        pizzaBuilder->buildCrust();
        pizzaBuilder->buildCheese();
        pizzaBuilder->buildToppings();
        pizzaBuilder->buildSauce();
        return pizzaBuilder->getPizza();
    }
};

int main() {
    PizzaChef chef;
    CustomPizzaBuilder customPizza;

    chef.setPizzaBuilder(&customPizza);
    Pizza* pizza = chef.makePizza();

    pizza->display();

    delete pizza;
    return 0;
}
```

Result and Discussion:

The Builder Design Pattern was successfully implemented in C++ for a pizza ordering system. The pattern allowed step-by-step construction of a customized pizza without requiring multiple constructors or complex object initialization. By separating the pizza creation process from its representation, the system became more flexible and maintainable. The key benefits of this approach include improved readability, better organization of code, and enhanced scalability, allowing new pizza variations to be added easily. The implementation demonstrated how the

Builder Pattern simplifies complex object creation while maintaining flexibility and reusability in object-oriented design.