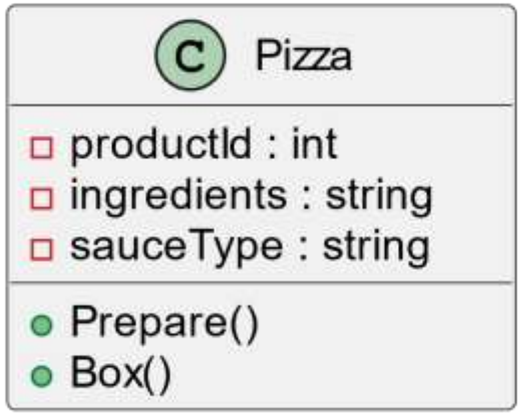
FACTORY METHOD

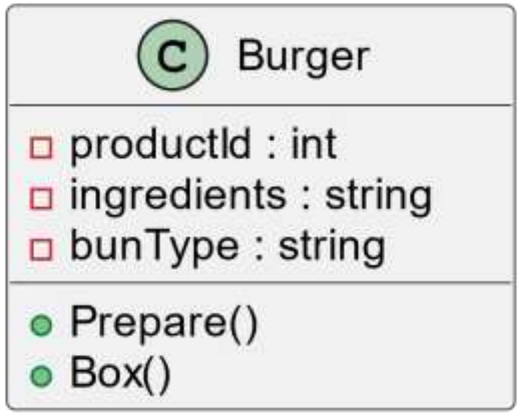
DEFINITON & MOTIVATION

- Creational design pattern
- „Define an interface for creating an object, but let subclasses decide which class to instantiate“
- Factory Method lets a class defer instantiation to subclasses

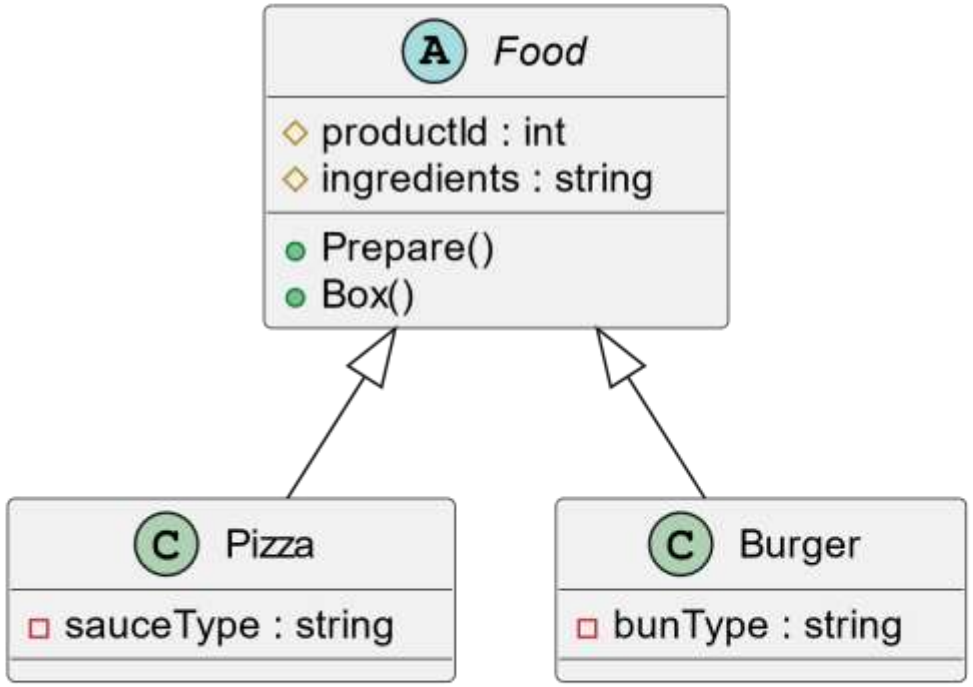
- We use Factory Method pattern when:
  - a class can’t anticipate the class of objects it must create
  - a class wants its subclasses to specify the objects it creates
  - classes delegate responsibility to one of several helper subclasses, and you want to localize the knowledge of which helper subclass is the delegate

```
public class Restaurant
  {
      public ??? orderFood(string request)
      {
          //could be replaced with switch
          if (request == "pizza")
          {
              Pizza pizza = new Pizza();
              pizza.Prepare();
              pizza.Box();
              return pizza;
          }
          else if (request == "burger")
          {
              Burger burger = new Burger();
              burger.Prepare();
              burger.Box();
              return burger;
          }
      }
  }
```

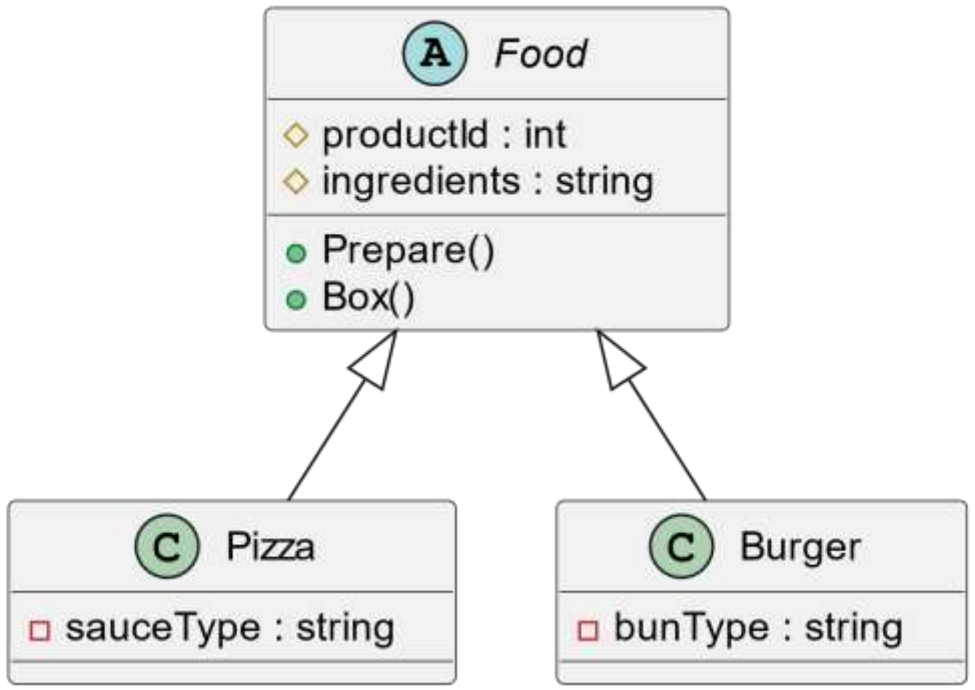




```
public class Restaurant
    {
        public Food orderFood(string request)
        {
            Food food;
            //could be replaced with switch
            if (request == "pizza")
                food = new Pizza();
            else if (request == "burger")
                food = new Burger();
            food.Prepare();
            food.Box();
            return food;
        }
    }
```



```
public class Restaurant
    {
        public Food orderFood(string request)
        {
            Food food;
            //could be replaced with switch
            if (request == "pizza")
                food = new Pizza();
            ~~else if (request == "burger")~~
                ~~food = new Burger();~~
            else if (request == "spaghetti")
                food = new Spaghetti();
            ~~food.Prepare();~~
            ~~food.Box();~~
            return food;
        }
    }
```



```
public class Restaurant
   {
       public Food orderFood(string request)
       {
           Food food;
           //could be replaced with switch
           if (request == "pizza")
               food = new Pizza();
           else if (request == "burger")
               food = new Burger();
           food.Prepare();
           food.Box();
           return food;
       }
   }
```

Single-responsibility ❌

Open-closed ❌

SIMPLE FACTORY

```
public class Restaurant
    {
        public Food orderFood(string request)
        {
          FoodFactory factory = new
FoodFactory();
          Food food =
factory.CreateFood(request);
         food.Prepare();
         food.Box();
          return food;
        }
    }
```

```
public class FoodFactory
    {
        public Food CreateFood(string request)
        {
            Food food;
            if (request == "pizza")
                food = new Pizza();
            else if (request == "burger")
                food = new Burger();
            return food;
        }
    }
```

SIMPLE FACTORY

```
public class Restaurant
    {
        public Food orderFood(string request)
        {
          FoodFactory factory = new
FoodFactory();
          Food food =
factory.CreateFood(request);
         food.Prepare();
         food.Box();
          return food;
        }
```

Single-responsibility ✓

Open-closed ✗

```
public class FoodFactory
   {
       public Food CreateFood(string request)
       {
           Food food;
           if (request == "pizza")
               food = new Pizza();
           else if (request == "burger")
               food = new Burger();
           return food;
       }
   }
```

FACTORY METHOD

```
public abstract class Restaurant
    {
        public Food orderFood()
        {
            Food food = createFood();
            food.Prepare();
            food.Box();
            return food;
        }
        //Factory Method
        public abstract Food
createFood();
    }
```

```
public class PizzaRestaurant : Restaurant
    {
        public override Food createFood()
        {
            return new Pizza();
        }
    }

public class BurgerRestaurant : Restaurant
    {
        public override Food createFood()
        {
            return new Burger();
        }
    }
```

```
     //client code
     Restaurant restaurant = new PizzaRestaurant();
     Food food = restaurant.orderFood();
```

FACTORY METHOD

```
public abstract class Restaurant
    {
        public Food orderFood()
        {
            Food food = createFood();
            food.Prepare();
            food.Box();
            return food;
        }
        //Factory Method
        public abstract Food
createFood();
    }
```

```
public class PizzaRestaurant : Restaurant
    {
        public override Food createFood()
        {
            return new Pizza();
        }
    }

public class BurgerRestaurant : Restaurant
    {
        public override Food createFood()
        {
            return new Burger();
        }
    }
```

```
//client code
Restaurant restaurant = new PizzaRestaurant();
Food food = restaurant.orderFood();
```

Single-responsibility ✓

Open-closed ✓

C++ BITS – VIRTUAL CONSTRUCTOR

```
class Restaurant {
public:
    //Factory Method
    virtual std::unique_ptr<Food> createFood() =
0;
    std::unique_ptr<Food> orderFood() {
        std::unique_ptr<Food> food = createFood();
        food->Prepare();
        food->Box();
        return food;
    }
    virtual ~Restaurant() {}
};
```

```
class PizzaRestaurant : public Restaurant {
public:
    std::unique_ptr<Food> createFood() override
{
        return std::make_unique<Pizza>();
    }
};

class BurgerRestaurant : public Restaurant {
public:
    std::unique_ptr<Food> createFood() override
{
        return std::make_unique<Burger>();
    }
};
```

```
//client code
std::unique_ptr<Restaurant> restaurant = std::make_unique<PizzaRestaurant>();
std::unique_ptr<Food> food = restaurant->orderFood();
```

C++ Bits – Lazy Initialization

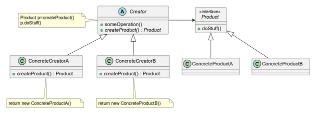
```
class Restaurant {
public:
    //Factory Method
    virtual std::shared_ptr<Food> createFood() =
0;
    std::shared_ptr<Food> orderFood() {
        if (!food_) {
            food_ = createFood();
        }
        food_->prepare();
        food_->box();
        return food_;
    }
private:
    std::shared_ptr<Food> food_;
};
```

```
class PizzaRestaurant : public Restaurant {
public:
    std::shared_ptr<Food> createFood() override
{
        return std::make_shared<Pizza>();
    }
};

class BurgerRestaurant : public Restaurant {
public:
    std::shared_ptr<Food> createFood() override
{
        return std::make_shared<Burger>();
    }
};
```

```
//client code
std::shared_ptr<Restaurant> restaurant = std::make_shared<PizzaRestaurant>();
std::shared_ptr<Food> food = restaurant->orderFood();
```

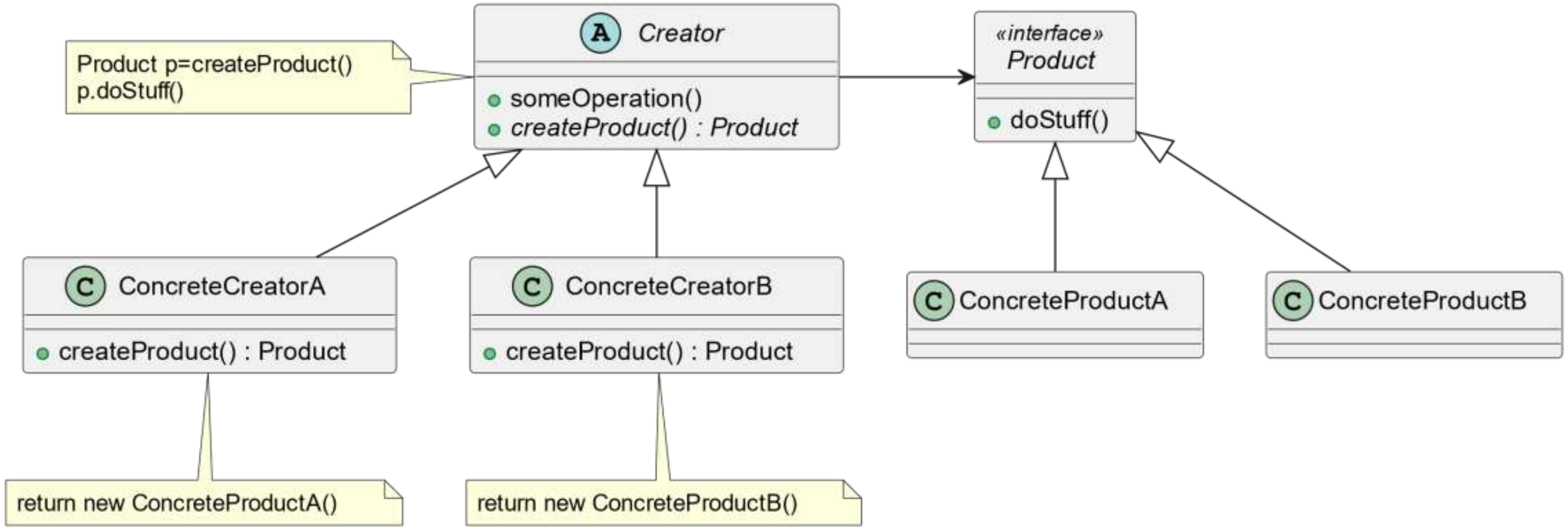
STRUCTURE


Product p=createProduct()
p.doStuff()
A Creator
someOperation()
createProduct() : Product
«interface»
Product
doStuff()
C ConcreteCreatorA
createProduct() : Product
C ConcreteCreatorB
createProduct() : Product
C ConcreteProductA
C ConcreteProductB
return new ConcreteProductA()
return new ConcreteProductB()

- Product
  - defines the interface of objects the factory method creates
- ConcreteProduct
  - implements the Product interface
- Creator
  - declares the factory method, which returns an object of type Product
  - may also define a default implementation of the factory method that returns a default ConcreteProduct object
  - may call the factory method to create a Product object
- ConcreteCreator
  - overrides the factory method to return an instance of a ConcreteProduct

DIAGRAM

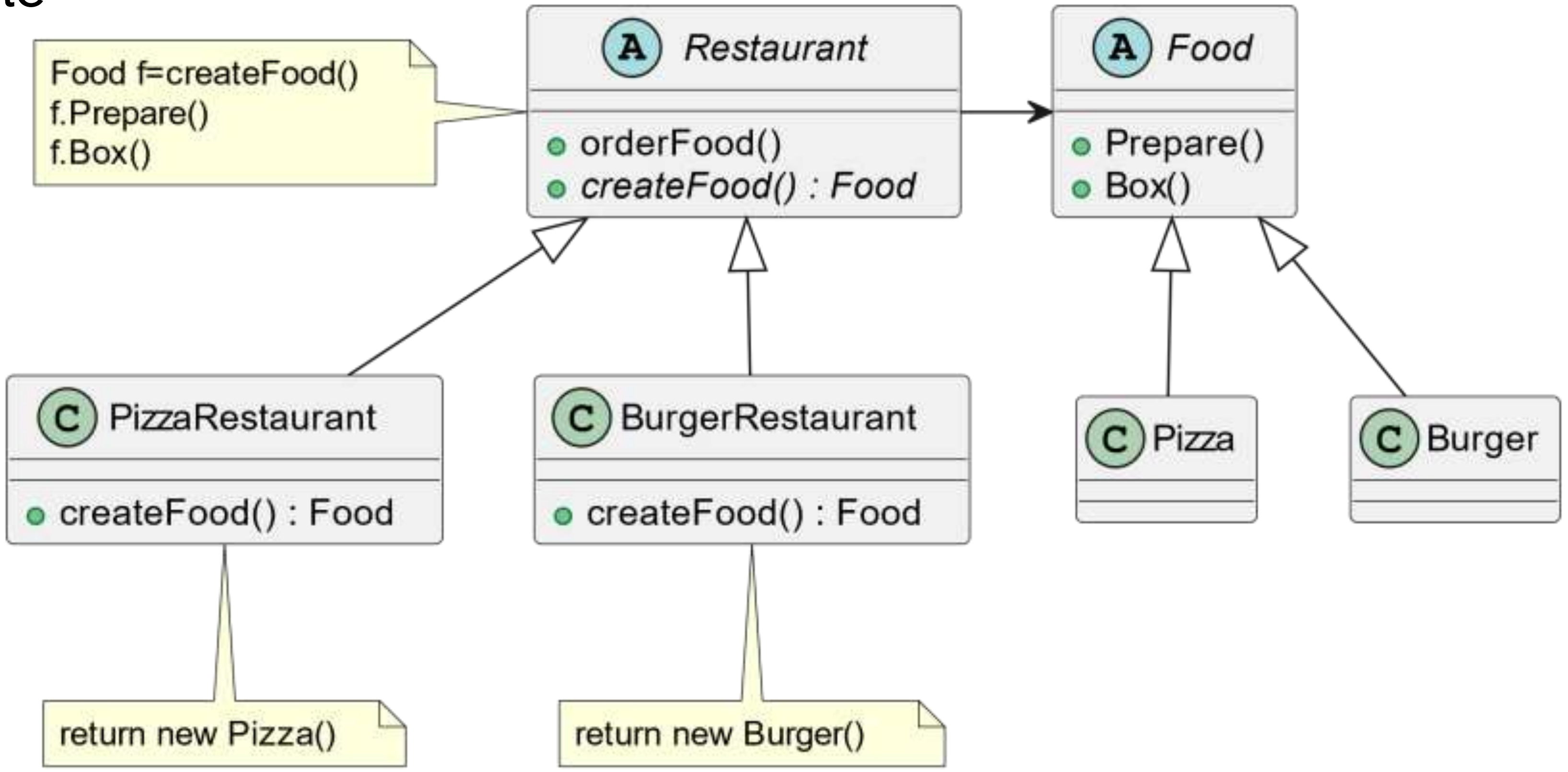
- General



Product p=createProduct()
p.doStuff()
A Creator
someOperation()
createProduct() : Product
«interface»
Product
doStuff()
C ConcreteCreatorA
createProduct() : Product
C ConcreteCreatorB
createProduct() : Product
C ConcreteProductA
C ConcreteProductB
return new ConcreteProductA()
return new ConcreteProductB()

DIAGRAM

- Concrete



PARAMETRIZED FACTORY METHOD

```
public class PizzaRestaurant : Restaurant
    {
        public override Food createFood()
        {
            return new Pizza();
        }
    }
```

PARAMETRIZED FACTORY METHOD

```
public class PizzaRestaurant : Restaurant
        {
            public override Food createFood(string pizzaType)
            {
                if (pizzaType == "cheese")
                {
                    return new CheesePizza();
                }
                else if (pizzaType == "pepperoni")
                {
                    return new PepperoniPizza();
                }
            }
        }
```

PARAMETRIZED FACTORY METHOD V2

```
public class SpaceshipFactory : GameFactory
    {
        public override GameObject createGameObject(string name,
                                                     int healthPoints,int power,int
size)
        {
            return new Spaceship(name,healthPoints,power,size);
        }
    }
```

STATIC FACTORY METHOD //SIMPLE FACTORY//

```
public class Restaurant
    {
        public Food orderFood(string request)
        {
          FoodFactory factory = new FoodFactory();
          Food food = factory.CreateFood(request);

          food.Prepare();

          food.Box();

          return food;
        }
    }
```

```
public class FoodFactory
    {
        public Food CreateFood(string request)
        {
            Food food;
            if (request == "pizza")
                food = new Pizza();
            else if (request == "burger")
                food = new Burger();
            return food;
        }
    }
```

Static Factory Method //Simple Factory//

```
public class Restaurant
    {
        public Food orderFood(string request)
        {
          Food food =
FoodFactory.CreateFood(request);

         food.Prepare();

         food.Box();


          return food;
        }
    }
```

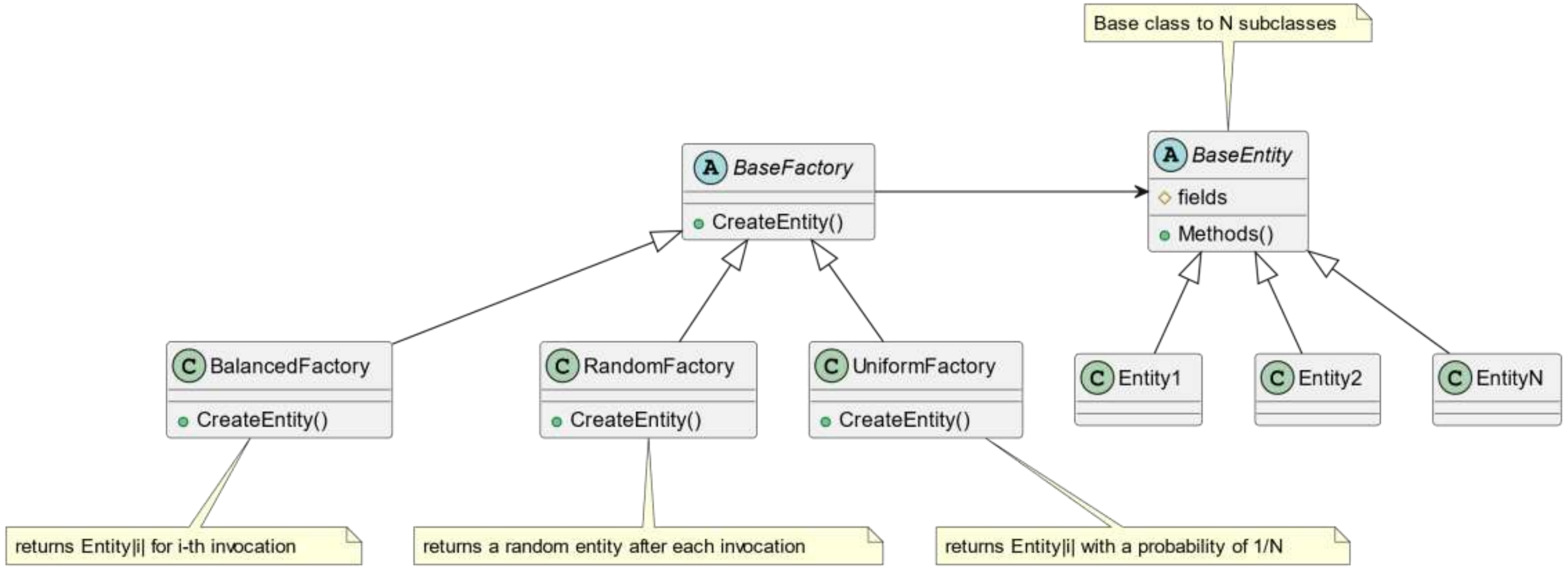
```
public class FoodFactory
    {
        public static Food CreateFood(string request)
        {
            Food food;
            if (request == "pizza")
                food = new Pizza();
            else if (request == "burger")
                food = new Burger();
            return food;
        }
    }
```

STATIC FACTORY METHOD //COLLECTIONS//

```
//Java
Map<String, Integer> map = new HashMap<>();
map.put("Alice", 25);
map.put("Bob", 30);
map.put("Charlie", 35);


Map<String, Integer> map = Map.of(
        "Alice", 25,
        "Bob", 30,
        "Charlie", 35
);
```

ONE MORE EXAMPLE



RELATIONS TO OTHER PATTERNS

- Abstract Factory
  - Often used with Factory Method
  - Factory Method can be considered a simplified version of AF
  - Families of objects vs. One type of object
- Template Method
  - Factory Method is a specialization of Template method
  - Factory Method can often often called fromTemplate Method
- Prototype
  - Doesn‘t require subclassing Creator
  - Often requires an Initialize operation on the Product class

SUMMARY

Pros

- Encapsulation
- Flexibility
- Loose coupling >> Interface vs concrete implementation
- Readability
- Testing >> Mock objects
- Open-closed principle
- Single-responsibility principle

Cons

- Complexity – Too many classes
- Needs inheritance >> generics, templates
- Potential overhead >> Scalability
- Limited complexity >> Abstract factory

SOURCES

- GoF
- https://www.youtube.com/watch?v=JEk7B_GUErc
- https://www.youtube.com/watch?v=EcFVTgRHJLM
- https://www.youtube.com/watch?v=EdFq_JIThqM