Software Architecture & Design SEC3071

Lecture No. 34

Muhammad Shahid
Department of Computer Science
National Textile University

shahid.abdullah@hotmail.com

Last Lecture Review

- Builder Design Pattern
  - Motivation
  - Intent
  - Class Diagram
  - Sequence Diagram
  - Implementation
- Application – Get Your Meal
- Applicability

Agenda – What will you Learn Today?

Factory Method Design Pattern

Creational Design Patterns

- Deal with one of the most commonly performed tasks in an OO application, the creation of objects
- Support a uniform, simple, and controlled mechanism to create objects
- Allow the encapsulation of the details about what classes are instantiated and how these instances are created
- Encourage the use of interfaces, which reduces coupling

Creational Design Patterns

		Purpose		
		Creational	**Structural**	**Behavioral**
Scope	**Class**	Factory Method	Adapter	Interpreter
	Object	▪ Abstract Factory ▪ Builder ▪ Prototype ▪ Singleton	▪ Adapter ▪ Bridge ▪ Composite ▪ Decorator ▪ Facade ▪ Flyweight ▪ Proxy	▪ Chain of Responsibility ▪ Command ▪ Iterator ▪ Mediator ▪ Memento ▪ Observer ▪ State ▪ Strategy ▪ Visitor

Factory Method Design Pattern

Solution : Factory Pattern

- "Factory Pattern defines an interface for creating the object but let the subclass decide which class to instantiate. Factory pattern let the class defer instantiation to the subclass."
- It works on the **principle of delegation**

Solution Description

- Encapsulating the functionality required, to select and instantiate an appropriate class separately
- Selects an appropriate class from a class hierarchy based on the application context and other influencing factor
- Instantiates the selected class and returns it as an instance of the parent class type
- This approach will decouple the client from object creation

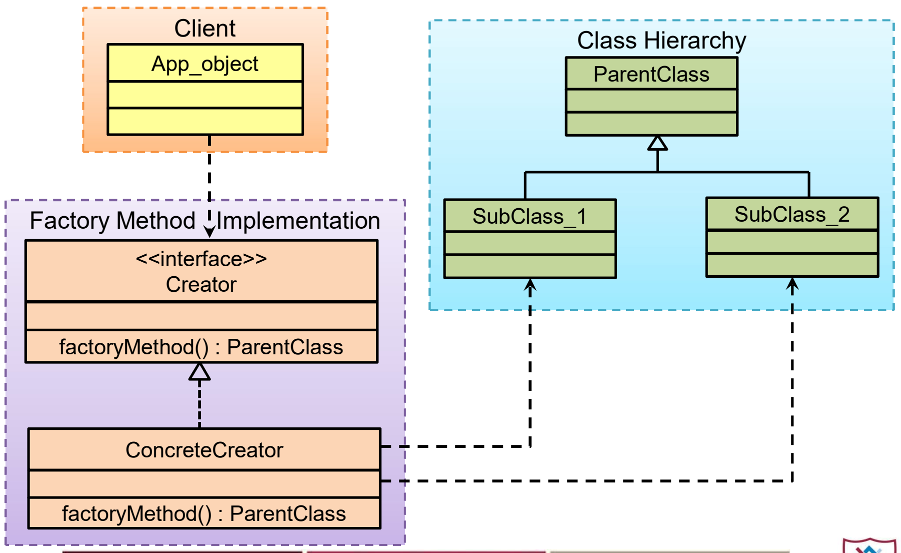
Factory Method

- Application objects can make use of the factory method to get access to the appropriate class instance
- This eliminates the need for an application object to deal with the varying class selection criteria
- Besides the class selection criteria, the factory method also implements any special mechanisms required to instantiate the selected class

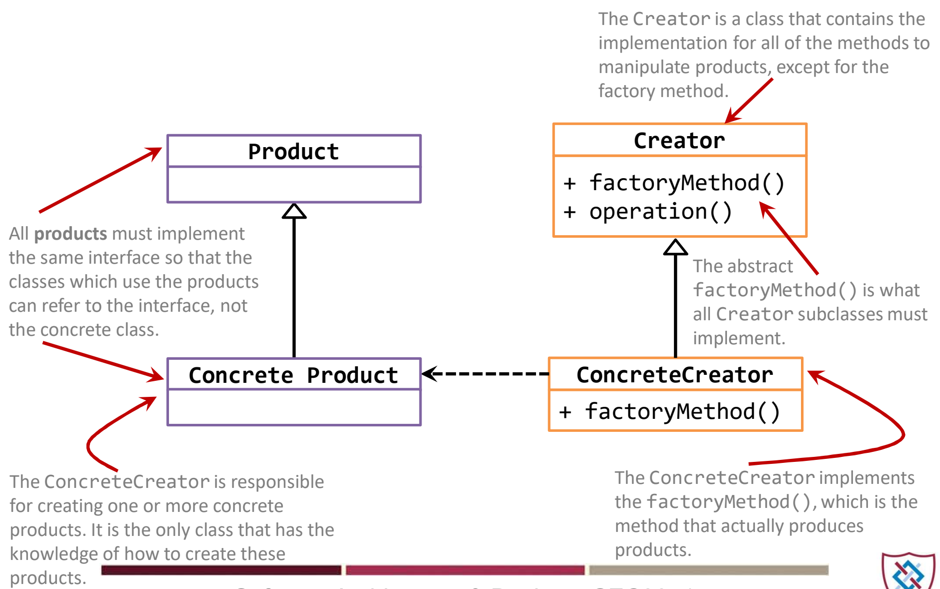
Factory Method

- This is applicable if different classes in the hierarchy need to be instantiated in different ways
- The factory method hides these details from application objects

Factory Pattern – Class Diagram



Factory Pattern – Class Diagram



The `Creator` is a class that contains the implementation for all of the methods to manipulate products, except for the factory method.

All **products** must implement the same interface so that the classes which use the products can refer to the interface, not the concrete class.

The abstract `factoryMethod()` is what all `Creator` subclasses must implement.

The `ConcreteCreator` is responsible for creating one or more concrete products. It is the only class that has the knowledge of how to create these products.

The `ConcreteCreator` implements the `factoryMethod()`, which is the method that actually produces products.

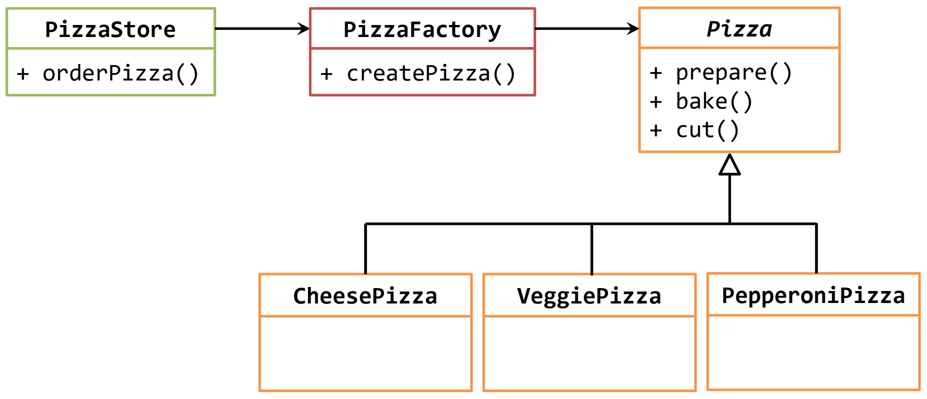
FM Pattern - Problem Statement

- There is a pizza shop which is offering different delicious pizza’s to it’s customer. There are some standard processes which are involved in a pizza creation like first the pizza is prepared by putting together all the ingredients, then it is prepared for baking, after baking it is cut and put into the boxes of as per order placed by the customer.

FM Pattern - Problem Statement

- There are different type of pizzas like cheese, chicken, vegetable etc. and this list keeps on increasing with addition of in demand pizzas and removal of not in demand pizzas

Factory Method Pattern – Solution



Factory Method Pattern – Solution

```
Pizza orderPizza(string type) {
        Pizza pizza;
        if(type.Equals("cheese")) {
                pizza = new CheesePizza();
        }
        else if(type.Equals("greek")) {
                pizza = new GreekPizza();
        }
        else if(type.Equals("peproni")) {
                pizza = new PeproniPizza();
        }
        pizza.bake();
        pizza.cut();
        pizza.box();
        return pizza;
}
```

Factory Method Pattern – Solution

```
Pizza orderPizza(string type) {
        Pizza pizza;
        if(type.Equals("cheese")) {
                pizza = new CheesePizza();
        }
        ~~else if(type.Equals("greek")) {~~
                ~~pizza = new GreekPizza();~~
        }
        else if(type.Equals("peproni")) {
                pizza = new PeproniPizza();
        }
        pizza.bake();
        pizza.cut();
        pizza.box();
        return pizza;
}
```

What Rule we are violating?

```
else if(type.Equals("veggie")) {
                pizza = new VeggiePizza();
}
```

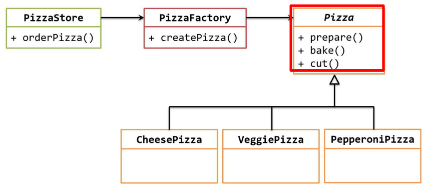
- Greek pizza is every where now. We want Veggie pizza to get weightage over our competitors.

Design Principle

Encapsulate What Varies

"Identify the aspects of your application that vary and separate them from what stays the same."

Factory Method Pattern – Solution



```
public abstract class Pizza
{
    protected string name;
    protected string dough;
    protected string sauce;
    protected ArrayList toppings = new ArrayList();

    public String getName()
    {
    return name;
    }
```

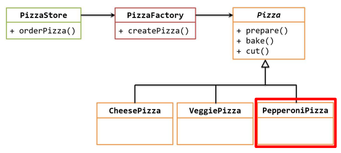
Factory Method Pattern – Solution

```
    public void prepare()
    {
        Console.WriteLine("Preparing:" + name);
        Console.WriteLine(".................");
        Console.Write Line("Tossing:" + dough);
        Console.WriteLine("Adding:" + sauce);
        Console.Write("Adding toppings:");

        foreach (string topping in toppings)
        {
            Console.Write(topping);
        }
    }
```

Factory Method Pattern – Solution

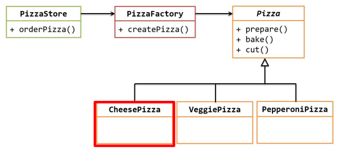
```
        public void bake()
        {
                Console.WriteLine("Baking:{0}", name);
        }

        public void cut()
        {
                Console.WriteLine("Cutting:{0}", name);
        }

        public void box()
        {
                Console.WriteLine("Boxing:{0}", name);
        }
} // End of Pizza class
```

Factory Method Pattern – Solution



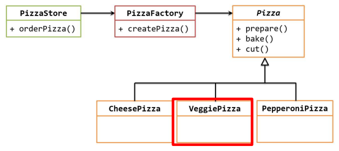
```
public class PepperoniPizza : Pizza
{
        public PepperoniPizza()
        {
                name = "Pepperoni Pizza";
                dough = "Crust";
                sauce = "Marinara sauce";
                toppings.Add("Sliced Pepperoni");
                toppings.Add("Sliced Onion");
                toppings.Add("Grated parmesan cheese");
        }
}
```

Factory Method Pattern – Solution



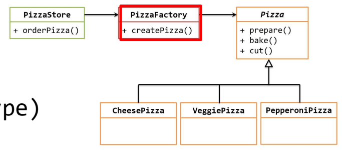
```
public class CheesePizza : Pizza
{
        public CheesePizza()
        {
                name = "Cheese Pizza";
                dough = "Regular Crust";
                sauce = "Marinara Pizza Sauce";
                toppings.Add("Fresh Mozzarella");
                toppings.Add("Parmesan");
        }
}
```

Factory Method Pattern – Solution



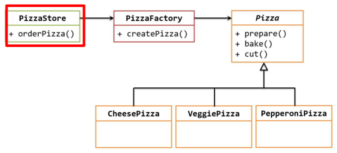
```
public class VeggiePizza : Pizza
{
        public VeggiePizza()
        {
                name = "Veggie Pizza";
                dough = "Crust";
                sauce = "Marinara sauce";
                toppings.Add("Shredded mozzarella");
                toppings.Add("Sliced mushrooms");
                toppings.Add("Sliced red pepper");
                toppings.Add("Sliced black olives");
        }
}
```

Factory Method Pattern – Solution



```
public class PizzaFactory
{
        public Pizza createPizza(string type)
        {
                Pizza pizza = null;
                if (type.Equals("cheese"))
                {
                        pizza = new CheesePizza();
                }
                else if (type.Equals("pepperoni"))
                {
                        pizza = new PepperoniPizza();
                }
                else if (type.Equals("veggie"))
                {
                        pizza = new VeggiePizza();
                }
        return pizza;
        }
}
```

Factory Method Pattern – Solution



```
public class PizzaStore
{
        PizzaFactory factory;

        public PizzaStore(PizzaFactory factory)
        {
                this.factory = factory;
        }
```

Factory Method Pattern – Solution

```
        public Pizza orderPizza(string type)
        {
                Pizza pizza;
                pizza = factory.createPizza(type);
                pizza.prepare();
                pizza.bake();
                pizza.cut();
                pizza.box();

                return pizza;
        }
}
```

Factory Method Pattern – Solution

```
static void Main(string[] args)
{
        PizzaFactory factory = new PizzaFactory();
        PizzaStore store = new PizzaStore(factory);

        Pizza cheese = store.orderPizza("cheese");
        Pizza veggie = store.orderPizza("veggie");
}
```

Factory Method Pattern – Solution

```
Preparing       Cheese Pizza
----------------------------
Tossing            Regular Crust
Adding             Marinara Pizza Sauce
Adding toppings:
                   --- Fresh Mozzarella
                   --- Parmesan
Baking             Cheese Pizza
Cutting            Cheese Pizza
Boxing             Cheese Pizza
```

Factory Method Pattern – Solution

```
Preparing       Veggie Pizza
----------------------------
Tossing            Crust
Adding             Marinara sauce
Adding toppings:
                   --- Shredded mozzarella
                   --- Sliced mushrooms
                   --- Sliced red pepper
                   --- Sliced black olives
Baking             Veggie Pizza
Cutting            Veggie Pizza
Boxing             Veggie Pizza
```

Recap

- Creational Pattern
- Factory Method Pattern
- Factory Pattern – Class Diagram
- Practical Example - Pizza Store

Questions