

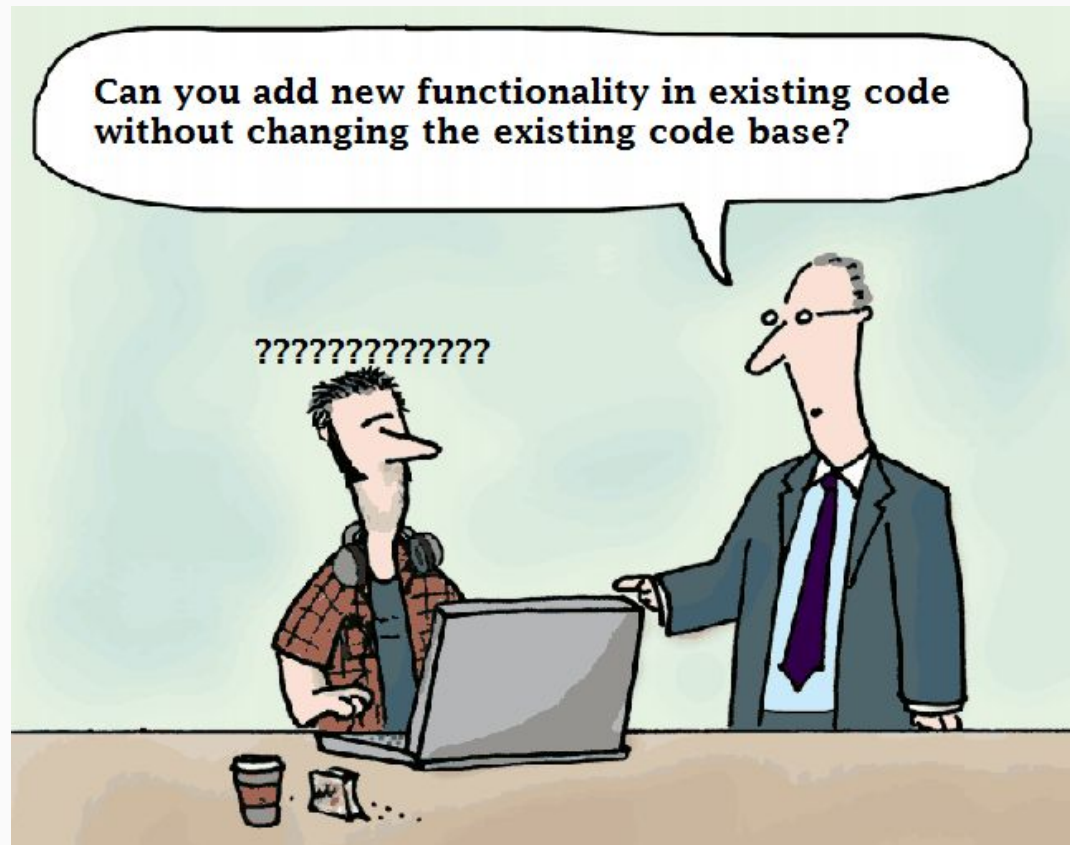
Lesson-11: Design Patterns

Round #1

CS 438
Ali Aburas PhD

Today's Goals

- Design patterns



What is a design pattern?

- A **standard** solution to a common programming problem
 - a design or implementation structure that achieves a particular purpose
 - provide a template to follow.
- **Gang of Four (GOF) Design Patterns**
 - The Gang of Four (GoF) Design Patterns, introduced in the book “Design Patterns: Elements of Reusable Object-Oriented Software,” authored by Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides, provide a catalog of proven solutions to common design problems in software development.



Why should you care?

1. Design patterns are a toolkit of tried and tested solutions to common problems in software design.
2. Good examples of OO principles.
3. Faster design phase.
4. Evidence that system will support change.
5. Offers shared vocabulary between designers.
6. You could come up with these solutions on your own ... But you shouldn't have to!
7. A design pattern is a known solution to a known problem.



Types of design patterns

- **Creational patterns**
 - how objects are instantiated
- **Structural patterns**
 - how objects / classes can be combined
- **Behavioral patterns**
 - how objects communicate
- **Concurrency patterns**
 - how computations are parallelized / distributed

Categories of design patterns

1. **Creational patterns** provide object creation mechanisms that increase flexibility and reuse of existing code. Decouple a client from objects it instantiates.
 - **Factory Method** is a creational design pattern that provides an interface for creating objects in a superclass, but allows subclasses to alter the type of objects that will be created.
 - **Abstract Factory** allows the creation of objects without specifying their concrete type.
 - **Builder**. Uses to create complex objects.
 - **Prototype**. Creates a new object from an existing object.
 - **Singleton**. Ensures only one instance of an object is created.
2. **Structural patterns** explain how to assemble objects and classes into larger structures, while keeping these structures flexible and efficient. Clean organization into subsystems.
 - **Adapter**. Allows for two incompatible classes to work together by wrapping an interface around one of the existing classes.
 - **Bridge**. Decouples an abstraction so two classes can vary independently.
 - **Composite**. Takes a group of objects into a single object.
 - **Decorator**. Allows for an object's behavior to be extended dynamically at run time.
 - **Facade**. Provides a simple interface to a more complex underlying object.
 - **Flyweight**. Reduces the cost of complex object models.
 - **Proxy**. Provides a placeholder interface to an underlying object to control access, reduce cost, or reduce complexity.
3. **Behavioral patterns** take care of effective communication and the assignment of responsibilities between objects. Describe how objects interact.
 - **Observer**. Is a publish/subscribe pattern which allows a number of observer objects to see an event.
 - **Visitor**. Separates an algorithm from an object structure by moving the hierarchy of methods into one object.

Creational patterns

Kinds of creational patterns

- **Creational patterns** provide object creation mechanisms that increase flexibility and reuse of existing code. Decouple a client from objects it instantiates.

- Singleton
- Factory
- Builder
- Prototype
- Flyweight

1- Design Pattern: Singleton

- **Singleton** is a creational design pattern that lets you ensure that a class has only one instance, while providing a global access point to this instance.
 - Singletons are classes which can be instantiated once, and can be accessed globally. This single instance can be shared throughout our application, which makes Singletons great for managing global state in an application.
 - Singleton: How to instantiate just one object - one and only one!
- **Why?**
 - This pattern is often used in situations like logging, managing connections to hardware or databases, where having just one instance makes sense.
 - If more than one instantiated: Incorrect program behavior, overuse of resources, inconsistent results
- **Alternatives:**
 - **Use a global variable**
 - Downside: assign an object to a global variable then that object might be created when application begins. If application never ends up using it and object is resource intensive: waste!
 - **Use a static variable**
 - Downside: how do you prevent creation of more than one class object?

The Singleton Pattern

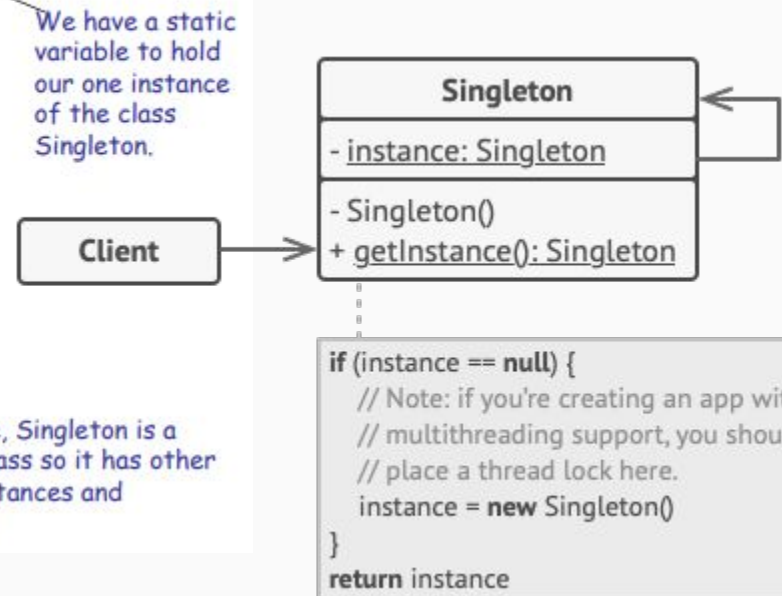
```
public class Singleton {  
    private static Singleton uniqueInstance;  
    // other useful instance variables  
  
    private Singleton () { }  
    public static Singleton getInstance () {  
        if (uniqueInstance == null) {  
            uniqueInstance = new Singleton ();  
        }  
        return uniqueInstance;  
    }  
  
    // other useful methods  
}
```

Constructor is declared private; only singleton can instantiate this class!

We have a static variable to hold our one instance of the class Singleton.

The getInstance () method gives us a way to instantiate the class and also return an instance of it.

Of course, Singleton is a regular class so it has other useful instances and methods.



Singleton Implementation

```
/**
 * Class Singleton is an implementation of a class that
 * only allows one instantiation.
 */
public class Singleton {
    // The private reference to the one and only instance.
    private static Singleton uniqueInstance = null;
    // An instance attribute.
    private int data = 0;
    /**
     * Returns a reference to the single instance.
     * Creates the instance if it does not yet exist.
     * (This is called lazy instantiation.)
     */
    public static Singleton instance() {
        if(uniqueInstance == null) uniqueInstance = new Singleton();
        return uniqueInstance;
    }
    /**
     * The Singleton Constructor.
     * Note that it is private!
     * No client can instantiate a Singleton object!
     */
    private Singleton() {}
    // Accessors and mutators here!
    public void setData(int d) {
        this.data=d;
    }
    public int getData() {
        return this.data;
    }
}
```

- Here's a test program:

```
public class TestSingleton {
    public static void main(String args[]) {
        // Get a reference to the single instance of Singleton.
        Singleton s = Singleton.instance();
        // Set the data value.
        s.setData(34);
        System.out.println("First reference: " + s);
        System.out.println("Singleton data value is: " +
                           s.getData());

        // Get another reference to the Singleton.
        // Is it the same object?
        s = null;
        s = Singleton.instance();
        System.out.println("\nSecond reference: " + s);
        System.out.println("Singleton data value is: " + s.getData());
    }
}
```

- And the test program output:

```
First reference: Singleton@1cc810
Singleton data value is: 34

Second reference: Singleton@1cc810
Singleton data value is: 34
```

Singleton Design Pattern

- **Advantages** of the Singleton Design Pattern

1. The Singleton pattern guarantees that there's only one instance with a unique identifier.
2. By keeping just one instance, the Singleton pattern can help lower memory usage in applications where resources are limited.

- **Disadvantages** of the Singleton Design Pattern

1. Singletons can make unit testing difficult since they introduce a global state.
2. The Singleton pattern creates a global dependency, which can complicate replacing the Singleton with a different implementation

2- Factory Method Design Pattern

- **What is the Factory Method Design Pattern?**

- The Factory Method pattern falls under the category of creational design patterns and is widely used to create objects without specifying their exact class types.
- Instead of directly instantiating objects using constructors, the Factory Method pattern delegates the responsibility of object creation to subclasses, allowing for greater flexibility and extensibility.

- **When to Use Factory Method Design Pattern?**

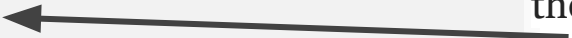
- The exact class of objects to be created is not known at compile time.
- There's a need for flexibility in object creation, allowing subclasses to provide variations of the created objects.
- The application needs to adhere to the Open/Closed Principle, where classes should be open for extension but closed for modification.

Motivation

- Let's say that you want to start a new pizza shop. You serve different kinds of pizzas and obviously you also have menu. Let's see a method that might appear in our **Pizaa store code**.

```
public Pizza orderPizza(String type) {  
    Pizza pizza;  
    if (type.equals("cheese"))  
        pizza = new CheesePizza();  
    else if (type.equals("clam"))  
        pizza = new ClamPizza();  
    else if (type.equals("veggie"))  
        pizza = new VeggiePizza();  
    else  
        return null;  
  
    //once we have correct type of pizza, do operations on it  
    pizza.prepare();  
    pizza.bake();  
    pizza.cut();  
    pizza.box();  
  
    return pizza;  
}
```

If you have done coding a lot then you know that this type of code occurs very frequently in any project.



- Let's say we want to add a new type of pizza or delete clamPizza then making changes in this type of code is even more difficult because we have to go to every place and make same changes.

Motivation

```
Pizza orderPizza(String type){  
    Pizza pizza;  
    if (type.equals("cheese")){  
        pizza = new CheesePizza();  
    }  
else if(type.equals("pepperoni")){  
    pizza = new PepperoniPizza();  
    } else if(type.equals("veggie")){  
        pizza = new VeggiePizza();  
    }  
    // Prep methods  
}
```

- When you see code like this, you know that when it comes time for changes or extensions, you'll have to reopen this code and examine what needs to be added (or deleted).

- This is what varies. As the pizza selection changes over time, you'll have to modify this code over and over.
- This code is NOT closed for modification

Improve-1

- Now let's move object creation out of orderPizza() method into a separate class SimplePizzaFactory whose job is just to create a pizza.

```
public class SimplePizzaFactory{  
    public Pizza createPizza(String type){  
        Pizza pizza=null;  
        if (type.equals("cheese"))  
            pizza = new CheesePizza();  
        else if (type.equals("clam"))  
            pizza = new ClamPizza();  
        else if (type.equals("veggie"))  
            pizza = new VeggiePizza();  
        return pizza  
    }  
}
```

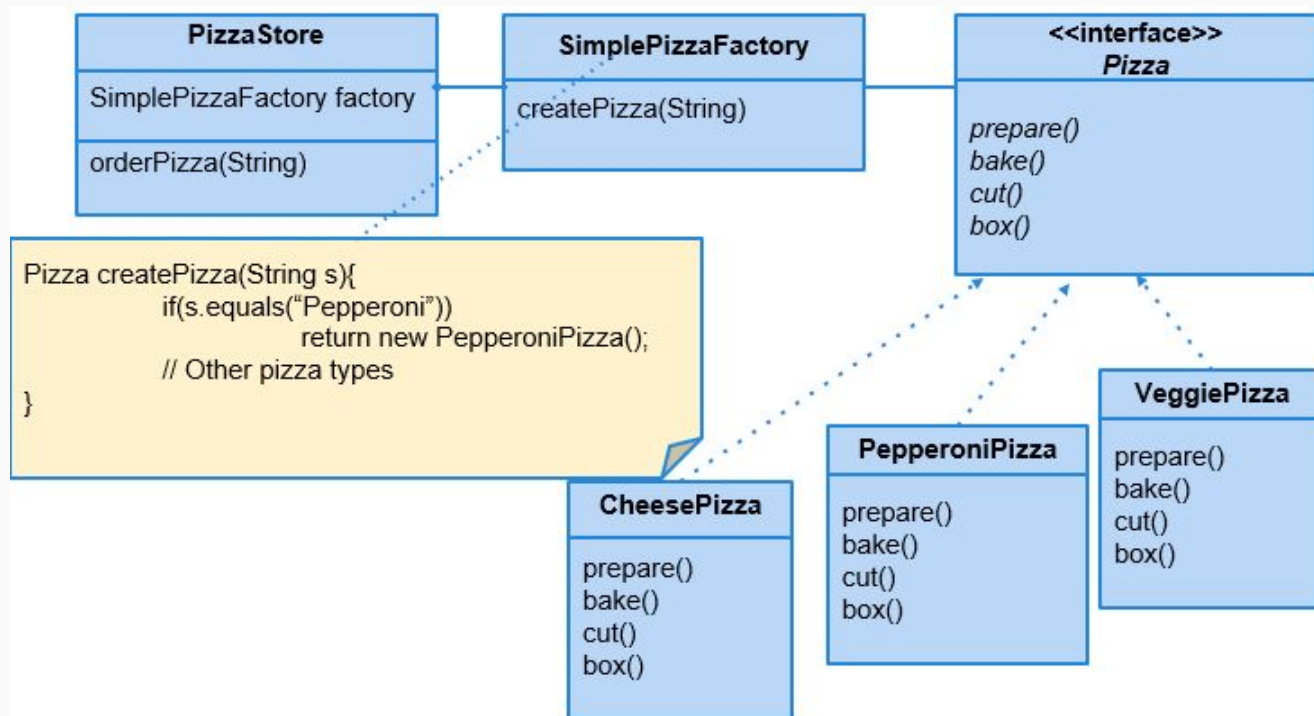

Improve-1

- Now our code from PizzaStore has reduced to this:

```
public class PizzaStore {  
    SimplePizzaFactory factory;  
  
    public PizzaStore(SimplePizzaFactory factory) {  
        this.factory = factory;  
    }  
  
    public Pizza orderPizza(String type) {  
        Pizza pizza = factory.createPizza(type);  
        pizza.prepare();  
        pizza.bake();  
        pizza.cut();  
        pizza.box();  
        return pizza;  
    }  
}
```

The Simple Factory

- **PizzaStore:** This class is the client of the factory. It need SimplePizzaFactory to get instance of type Pizza.
- **SimplePizzaFactory:** This is the factory where we create pizzas. It should be the only part of our application that refers to concrete Pizza classes.
- **Pizza:** This is the product of the factory. here this is a abstract class which has some methods already implemented.
- **Concrete Pizza classes:** these are our concrete products. Each of them should extend Pizza class.



Improve-2

- Your pizza shop has now become famous and everyone loves it. You want to expand your business to New York and Chicago. Let's see how you can do this:
 - Make our createPizza factory method in the PizzaStore as a abstract method
 - PizzaStore will also be a abstract class

```
public abstract class PizzaStore {  
  
    abstract Pizza createPizza(String item);  
  
    public Pizza orderPizza(String type) {  
        Pizza pizza = createPizza(type);  
        System.out.println("--- Making a " + pizza.getName() + " ---");  
        pizza.prepare();  
        pizza.bake();  
        pizza.cut();  
        pizza.box();  
        return pizza;  
    }  
}
```

Improve-2

- Each region will have its own PizzaStore type which will extend this PizzaStore and provide implementation for createPizza method

```
public class ChicagoPizzaStore extends PizzaStore {  
    Pizza createPizza(String item) {  
        if (item.equals("cheese")) {  
            return new ChicagoStyleCheesePizza();  
        } else if (item.equals("veggie")) {  
            return new ChicagoStyleVeggiePizza();  
        } else if (item.equals("clam")) {  
            return new ChicagoStyleClamPizza();  
        } else if (item.equals("pepperoni")) {  
            return new ChicagoStylePepperoniPizza();  
        } else return null;  
    }  
}
```

Improve-2

- Each region will have its own PizzaStore type which will extend this PizzaStore and provide implementation for createPizza method

```
public class NYPizzaStore extends PizzaStore {  
  
    Pizza createPizza(String item) {  
        if (item.equals("cheese")) {  
            return new NYStyleCheesePizza();  
        } else if (item.equals("veggie")) {  
            return new NYStyleVeggiePizza();  
        } else if (item.equals("clam")) {  
            return new NYStyleClamPizza();  
        } else if (item.equals("pepperoni")) {  
            return new NYStylePepperoniPizza();  
        } else return null;  
    }  
}
```

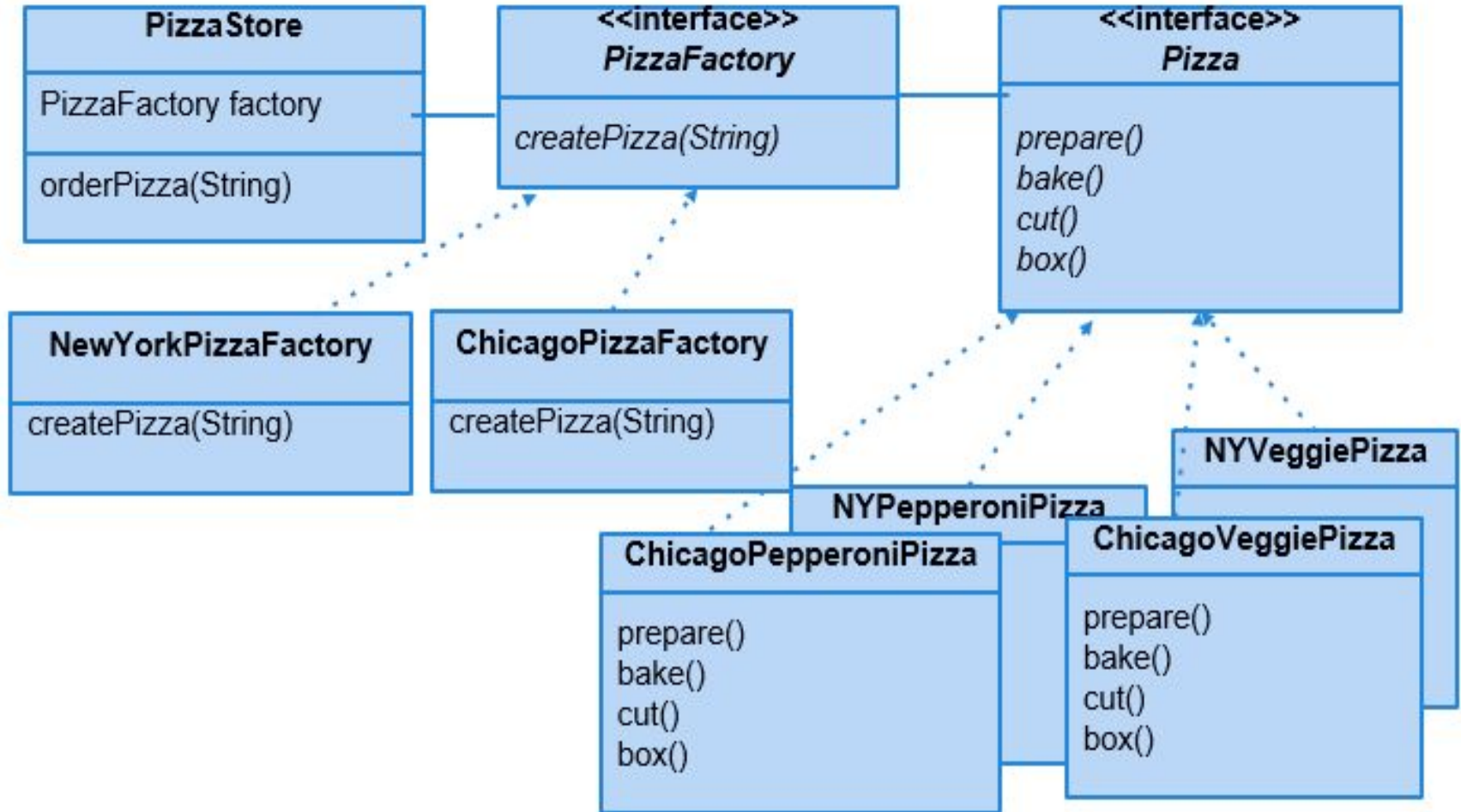
Improve-2

- abstract createPizza() method in abstract PizzaStore class act as a factory method.

```
//Here's how you can order pizza using this approach  
PizzaStore store = new NYPizzaStore();  
Pizza pizza = store.orderPizza("cheese");
```

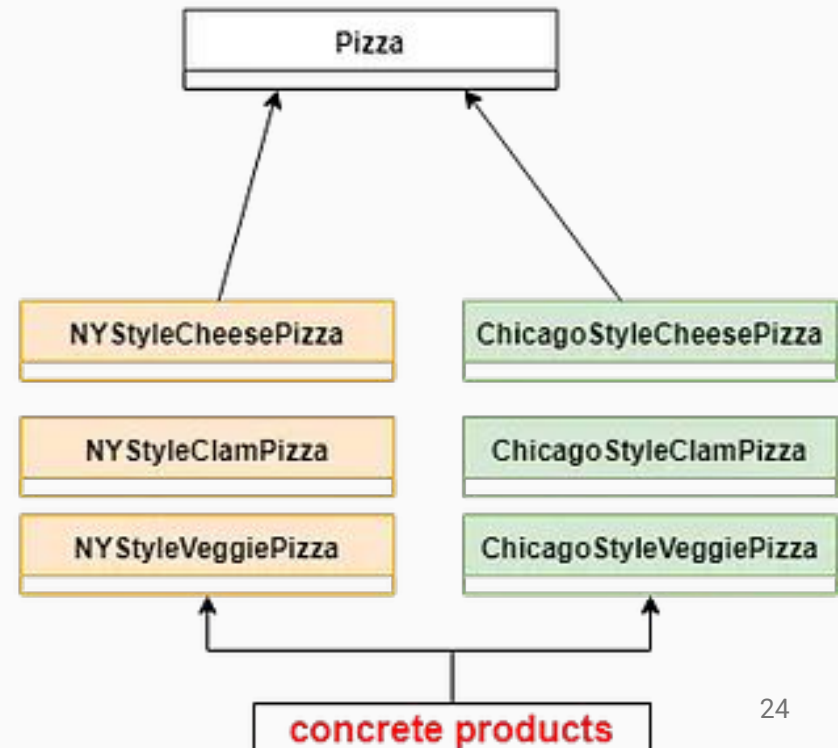
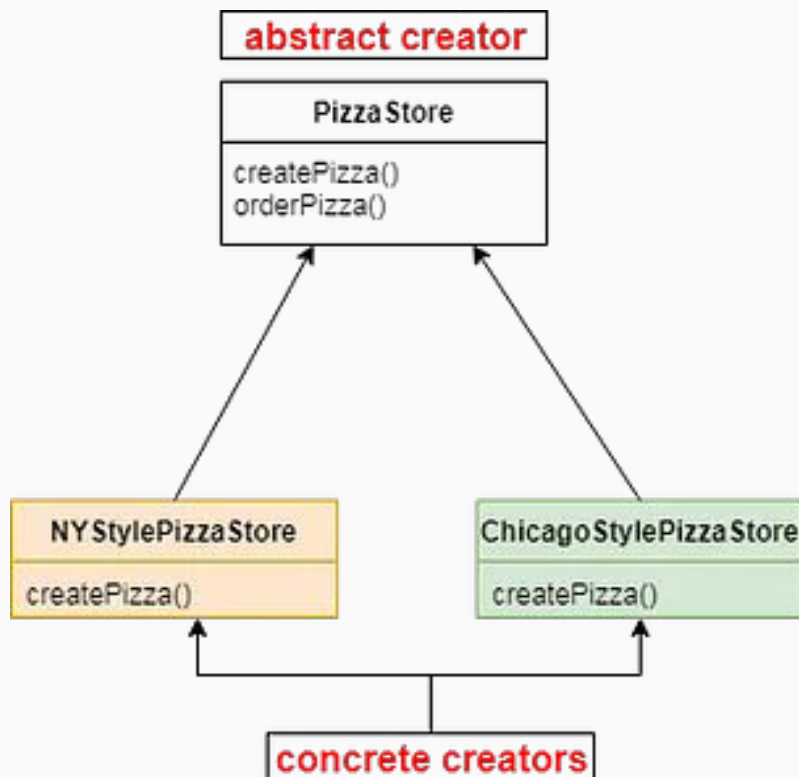
- In the above code when we call store.orderPizza("cheese") , then orderPizza() method of PizzaStore will be called and further this method will call createPizza() method, but createPizza is abstract in this class and has no implementation,
- so what will it do? It all depends on which sub-class of PizzaStore is instantiated. As we have instantiated NYPizzaStore, so createPizza() of NYPizzaStore will be called, and it will return appropriate Pizza type.

Franchising the Factory

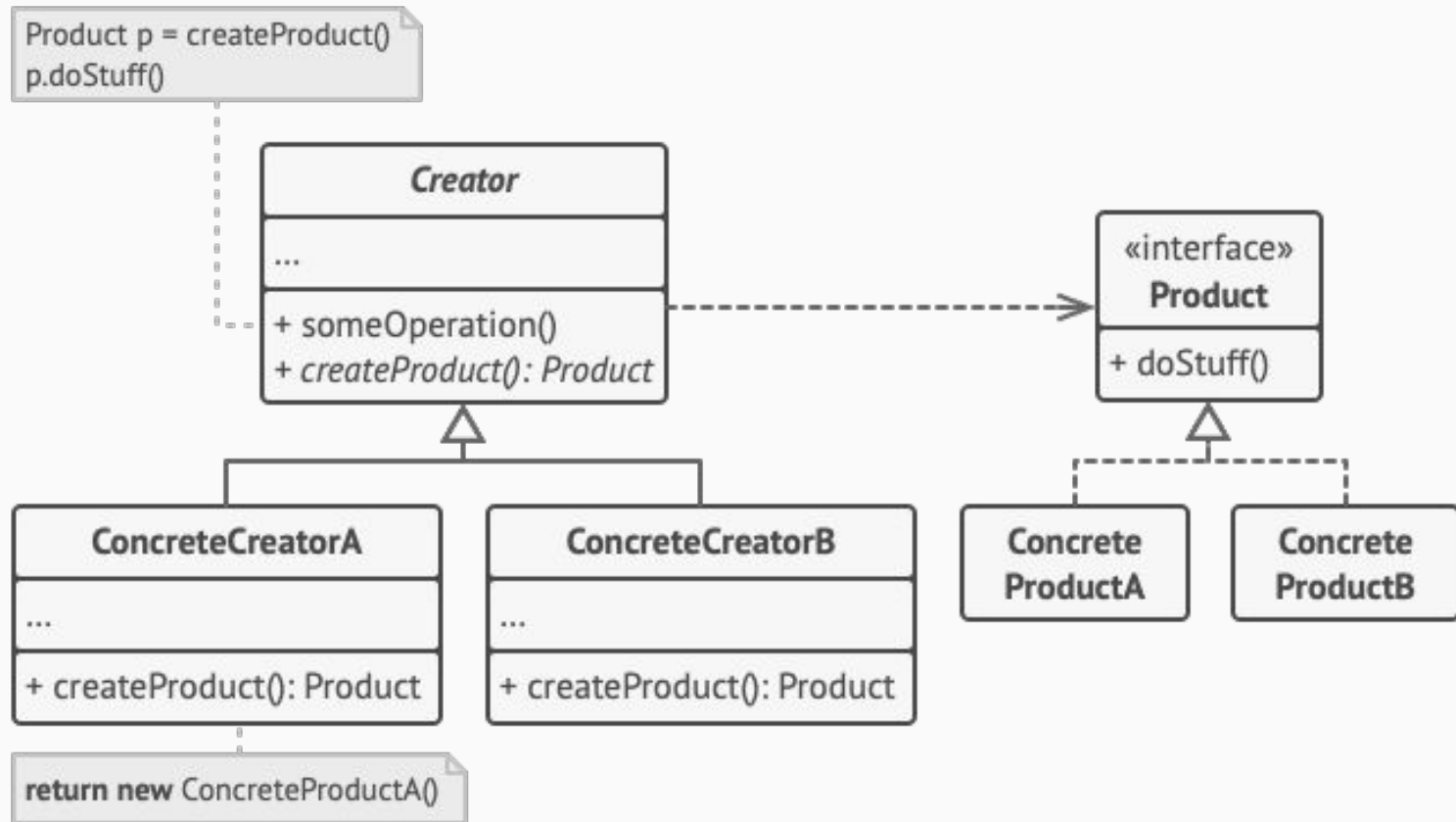


Factory Pattern - the class diagram

- abstract creator class defines an abstract factory method that subclasses implement to produce products.
- classes that produce products are called concrete creators
- Factories produce products and here our product is pizza



Factory Pattern - In Practice



2- Design Pattern - Factory Method Pattern

- Factory pattern is a creational design pattern that lets you produce families of related objects without specifying their concrete classes.
- This pattern defines an interface for creating an object, but lets subclasses decide which class to instantiate. Factory method lets a class defer instantiation to subclasses.

● Advantages	● Disadvantages
You avoid tight coupling between the creator and the concrete products.	The code may become more complicated since you need to introduce a lot of new subclasses to implement the pattern.
Promotes code reusability by defining common creation logic in superclass methods.	Increases complexity in the codebase, especially when dealing with multiple factories and product types.
Open/Closed Principle. You can introduce new types of products into the program without breaking existing client code.	May introduce runtime errors if the factory method is not implemented properly

Use Cases of Factory Design Pattern

1. Database Connection Factory
 - a. **Use Case:** In applications that need to connect to different types of databases (e.g., MySQL, PostgreSQL, Oracle), a Database Connection Factory can be used to create the appropriate database connection based on the configuration or environment.
 - b. **Why Factory:** Instead of manually instantiating a database connection object based on specific database types, the factory allows the creation of the connection object to be abstracted. The client code remains agnostic of the specific database implementation.
2. Notification Systems (Email, SMS, Push Notifications)
 - a. **Use Case:** In systems that send different types of notifications (e.g., email, SMS, push notifications), a Notification Factory can be used to create the appropriate notification service based on user preferences or application settings.
 - b. **Why Factory:** The factory hides the details of how notifications are sent, making the system easier to extend with new notification types.
3. Loggers (File Logger, Database Logger, Console Logger)
 - a. **Use Case:** In applications that need to log messages in different formats (e.g., to a file, database, or console), a Logger Factory can be used to create the appropriate logger.
 - b. **Why Factory:** This ensures that the logging system is flexible and can be easily extended to support new loggers without modifying the application code.