Design Patterns

Factory Pattern

Who should create?

ebru@hacettepe.edu.tr

ebruakcapinarsezer@gmail.com

http://yunus.hacettepe.edu.tr/~ebru/

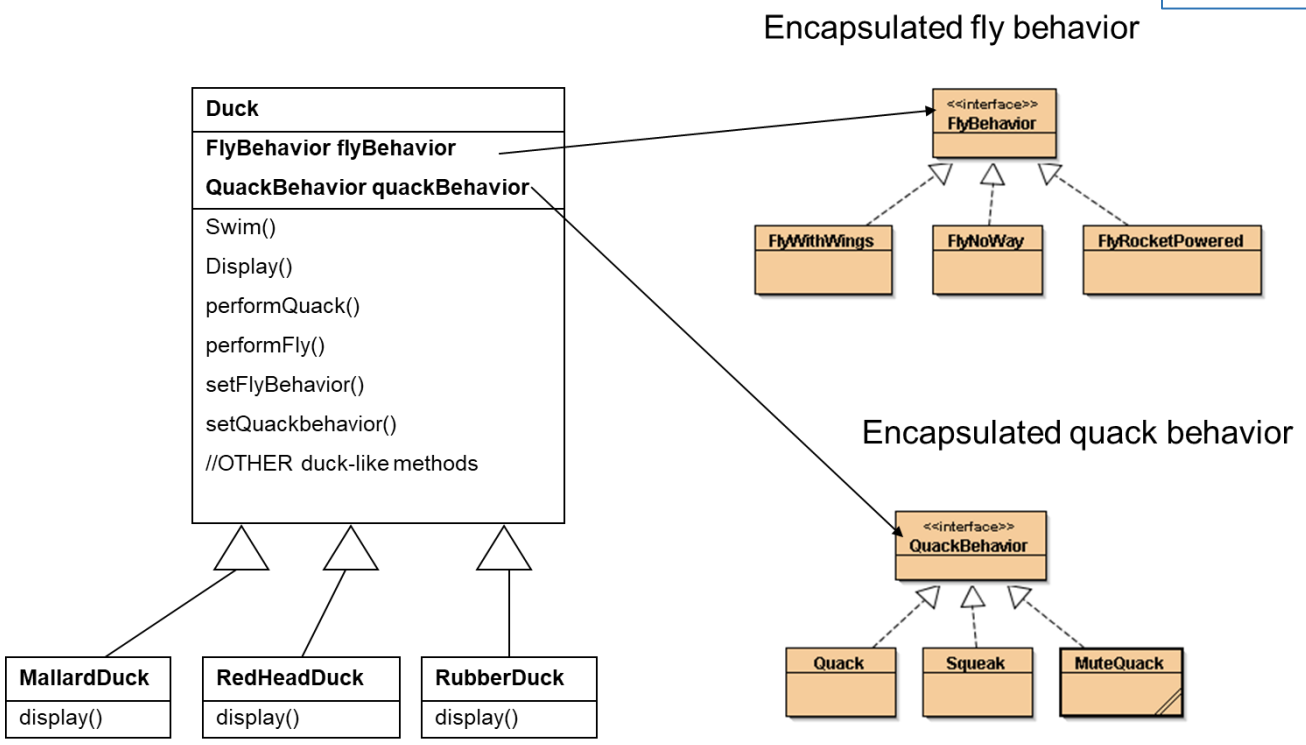
@ebru176

Ekim 2017

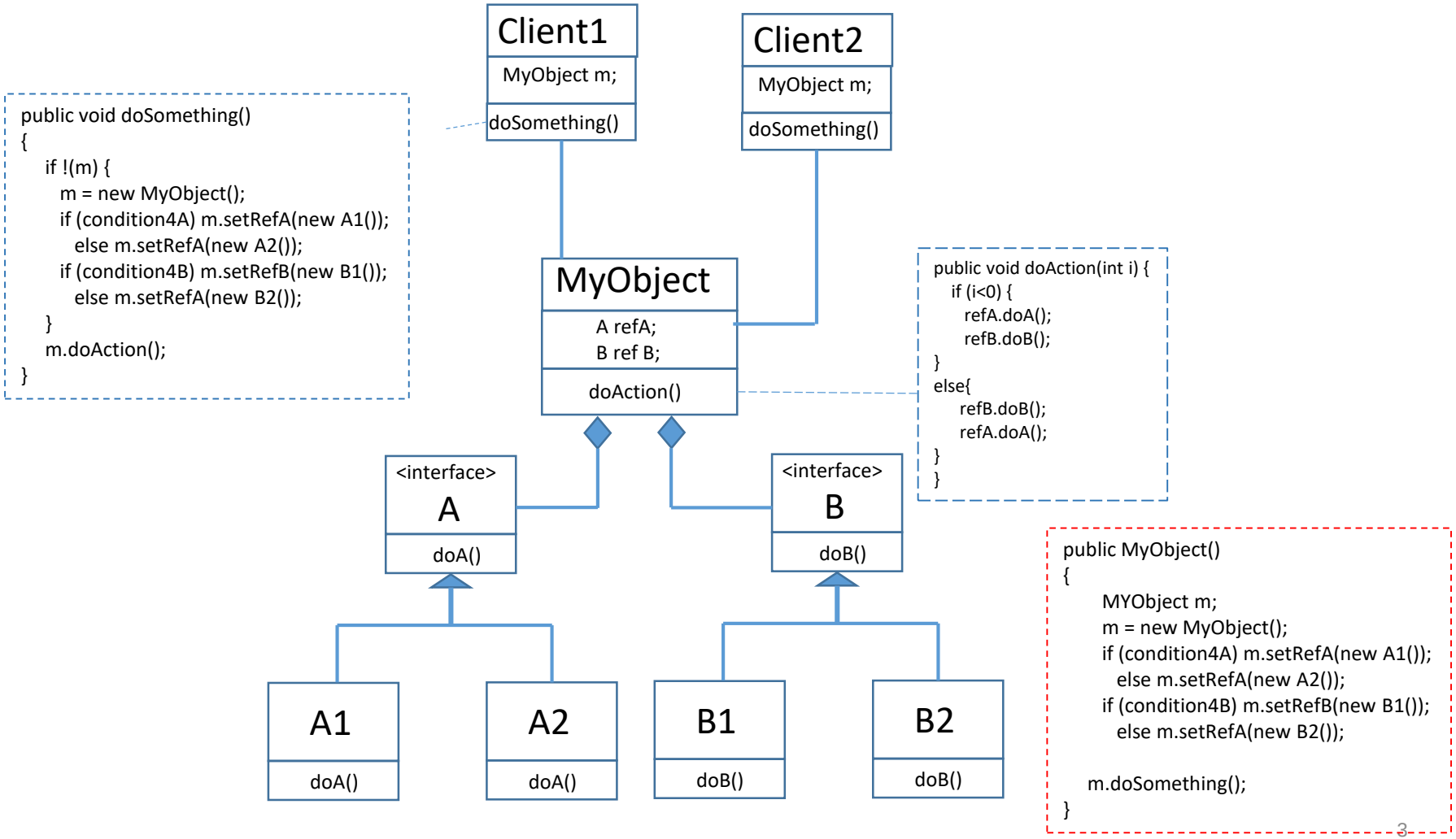
revised from, www.uwosh.edu/faculty_staff/huen/262/f09/slides/10_ Strategy_Pattern**.**ppt***

Last point we remained

Duck should not know:
-Its subclasess
-Its changing behaviour types



Who can create, who should create?



Factory Method Pattern

Define an interface for creating an object, but let subclasses decide which class to instantiate

revision on march 2017, @ebru

Factory Method: Applicability

- Use the Factory Method pattern when
  - to make client class as unable to anticipate the class of objects it must create/have
  - a class wants its subclasses to specify the objects it creates

The Factory Method Pattern

- This is a ‘Creational’ pattern, i.e. it is concerned with object instantiation

- Used where an abstract class (A) may, itself, need to create objects of other classes
  - where the precise class is not necessarily known.

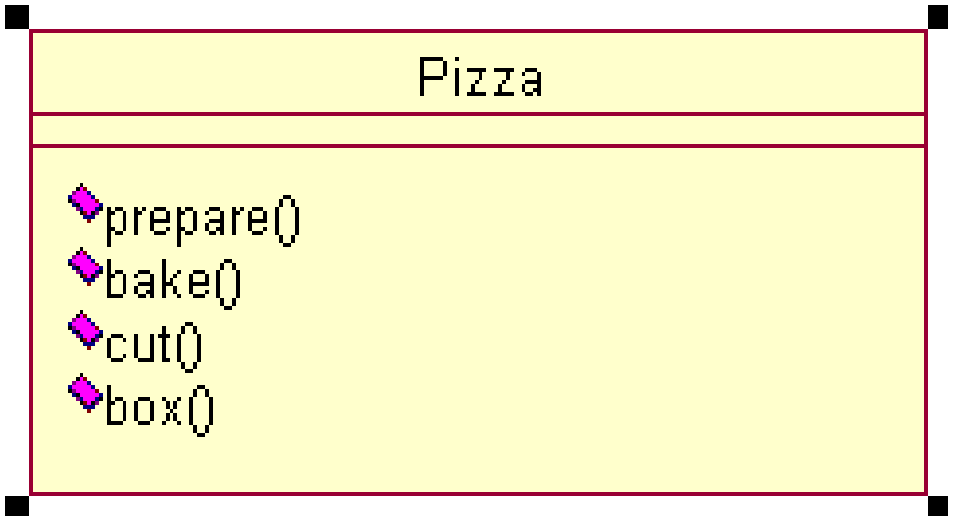
- The precise class to be instantiated may only be known within a sub-class of A.
  - However, all sub-classes of A will share the common signature of the super-class. Therefore, the abstract class (A) may interact with the created object through this interface.

Factory Method pattern

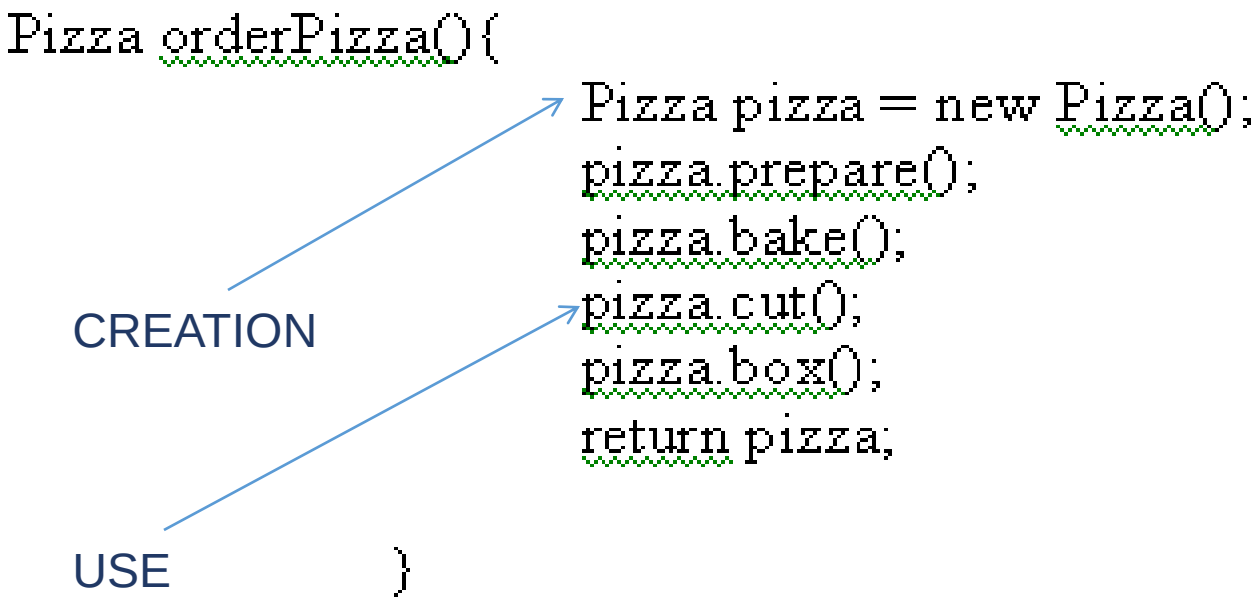
- Define an interface for creating an object, but let subclasses decide which class to instantiate. It lets a class defer instantiation to subclasses

- PROBLEM:
  - A framework with abstract application classes and application-specific subclasses to instantiate to realize different implementations
- SOLUTION:
  - The Factory Method pattern encapsulates the knowledge of which subclass to create and moves this knowledge out of the framework

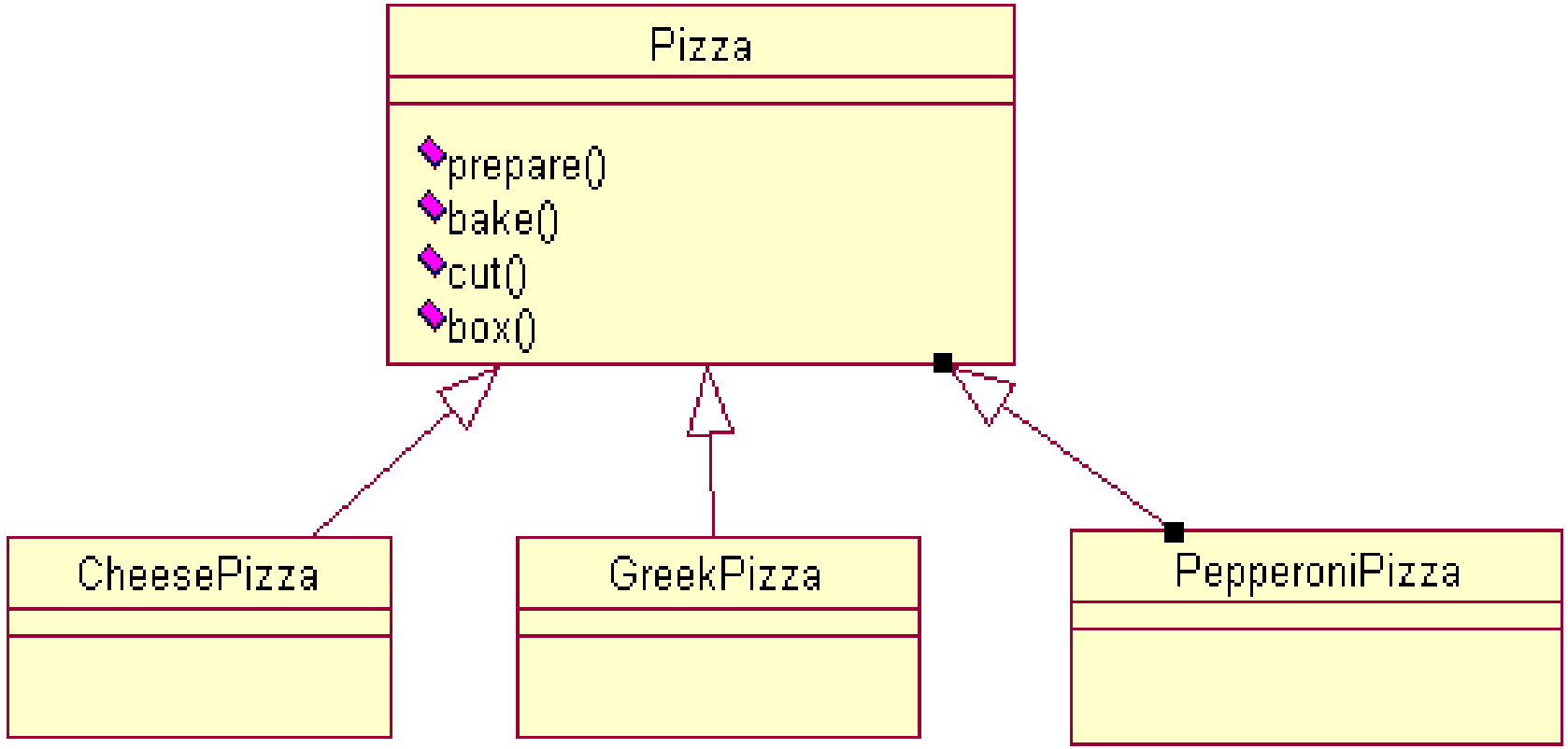
Example: Simple Pizza Class



Creation and use of Pizza Class



Pizza starts subclassing

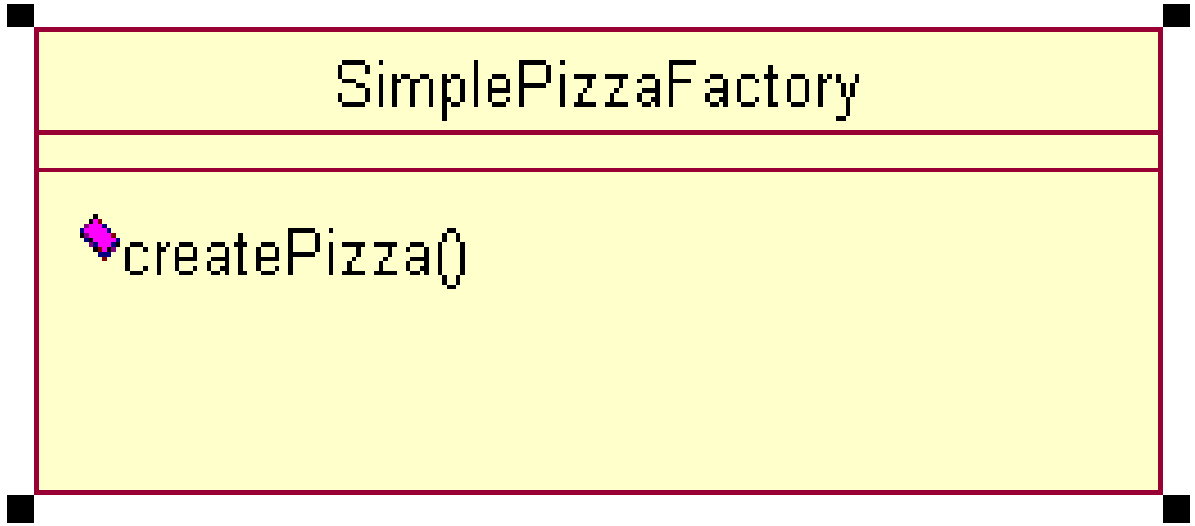


```
Pizza orderPizza(String type)
{
        Pizza pizza;
        if(type.equals("cheese")){
                        pizza = new CheesePizza();
        }else if(type.equals("greek")){
                        pizza = new GreekPizza();
        }else if (type.equals("Pepperoni")){
                        pizza = new PepperoniPizza();
        }
        pizza.prepare();
        pizza.bake();
        pizza.cut();
        pizza.box();
        return pizza;

}
```

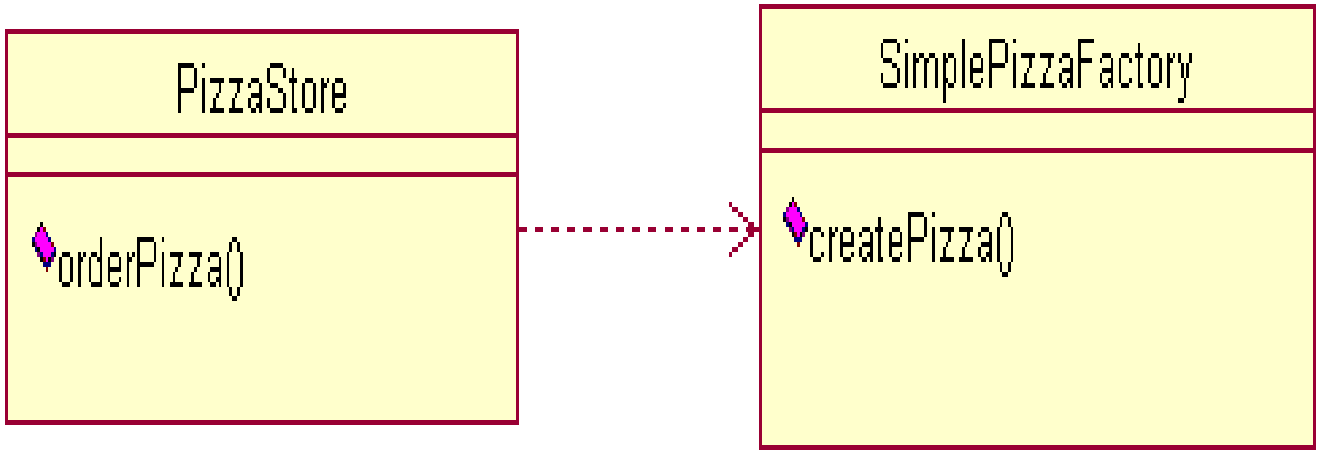
CREATION KNOWLEDGE

```
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("Pepperoni"))
        {
                pizza = new PepperoniPizza();
        }else if (type.equals("clam"))
        {
                pizza = new ClamPizza();
        }else if (type.equals("veggie"))
                pizza = new VeggiePizza();
        }
        pizza.prepare();
        pizza.bake();
        pizza.cut();
        pizza.box();
        return pizza;

}
```

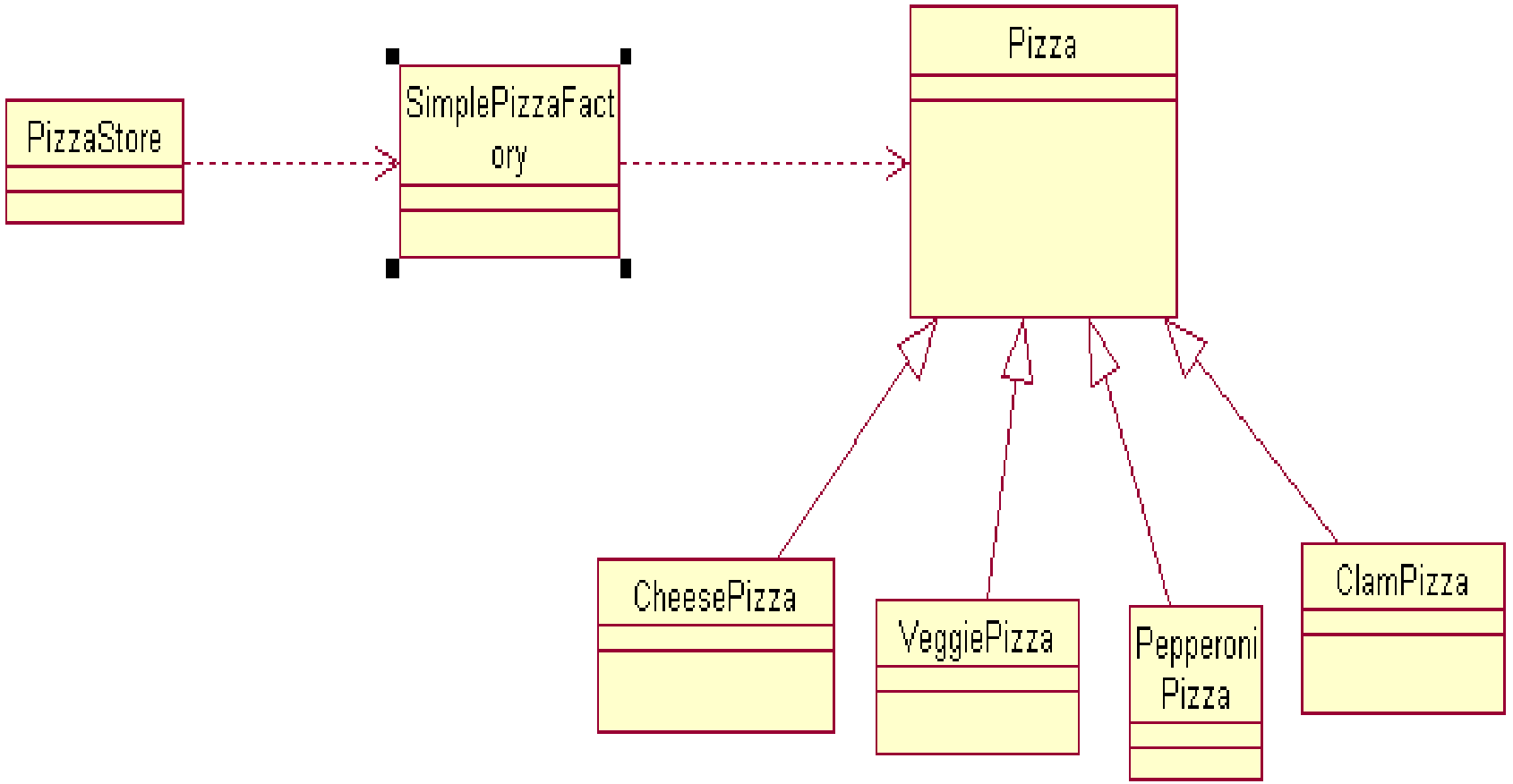
CHANGE IN CREATION


SimplePizzaFactory
createPizza()

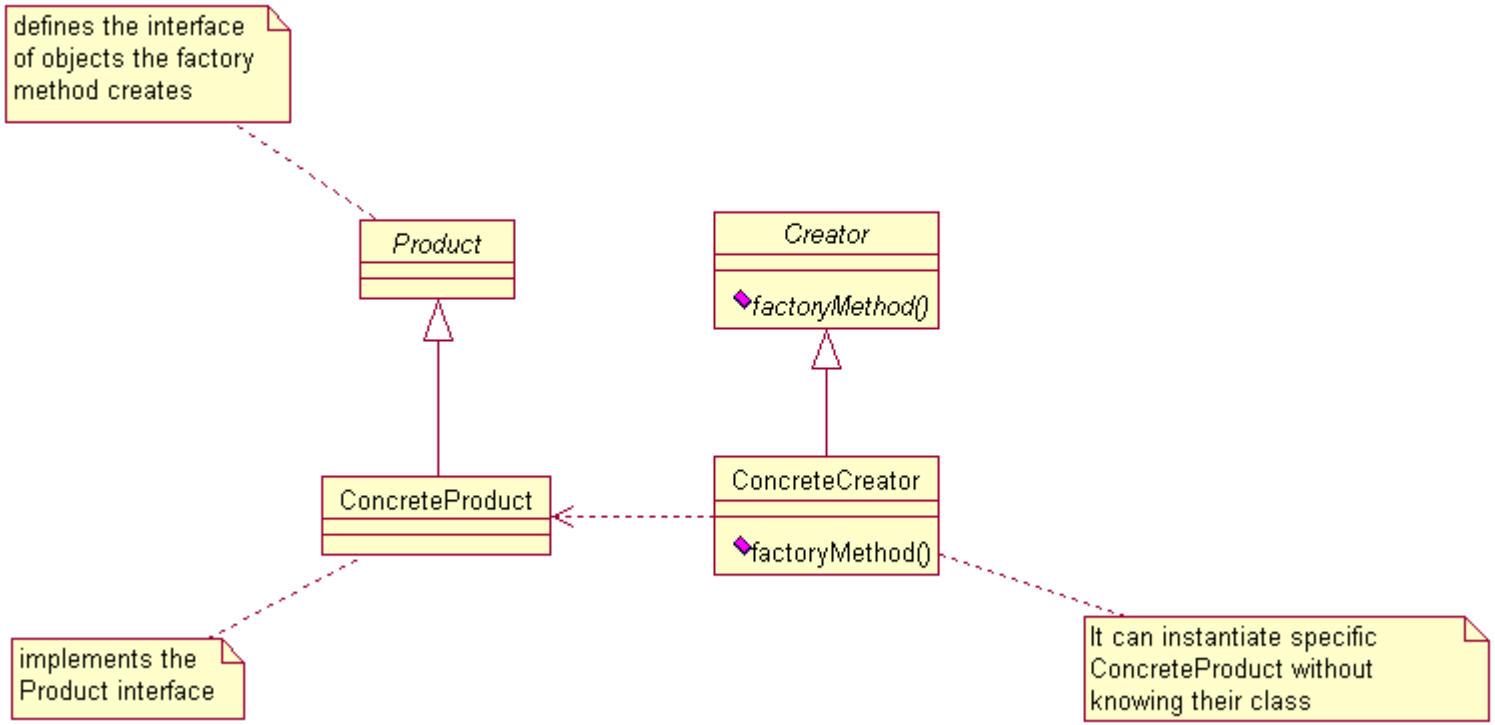
```
public class SimplePizzaFactory {
        public Pizza createPizza(String type)
        {
                Pizza pizza = null;
                if(type.equals("cheese")){
                        pizza = new CheesePizza();
                }else if (type.equals("Pepperoni")){
                       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


PizzaStore
orderPizza()
SimplePizzaFactory
createPizza()

```
public class PizzaStore
{
        SimplePizzaFactory factory;

        public PizzaStore(SimplePizzaFactory factory) {
                this.factory = factory;
        }

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

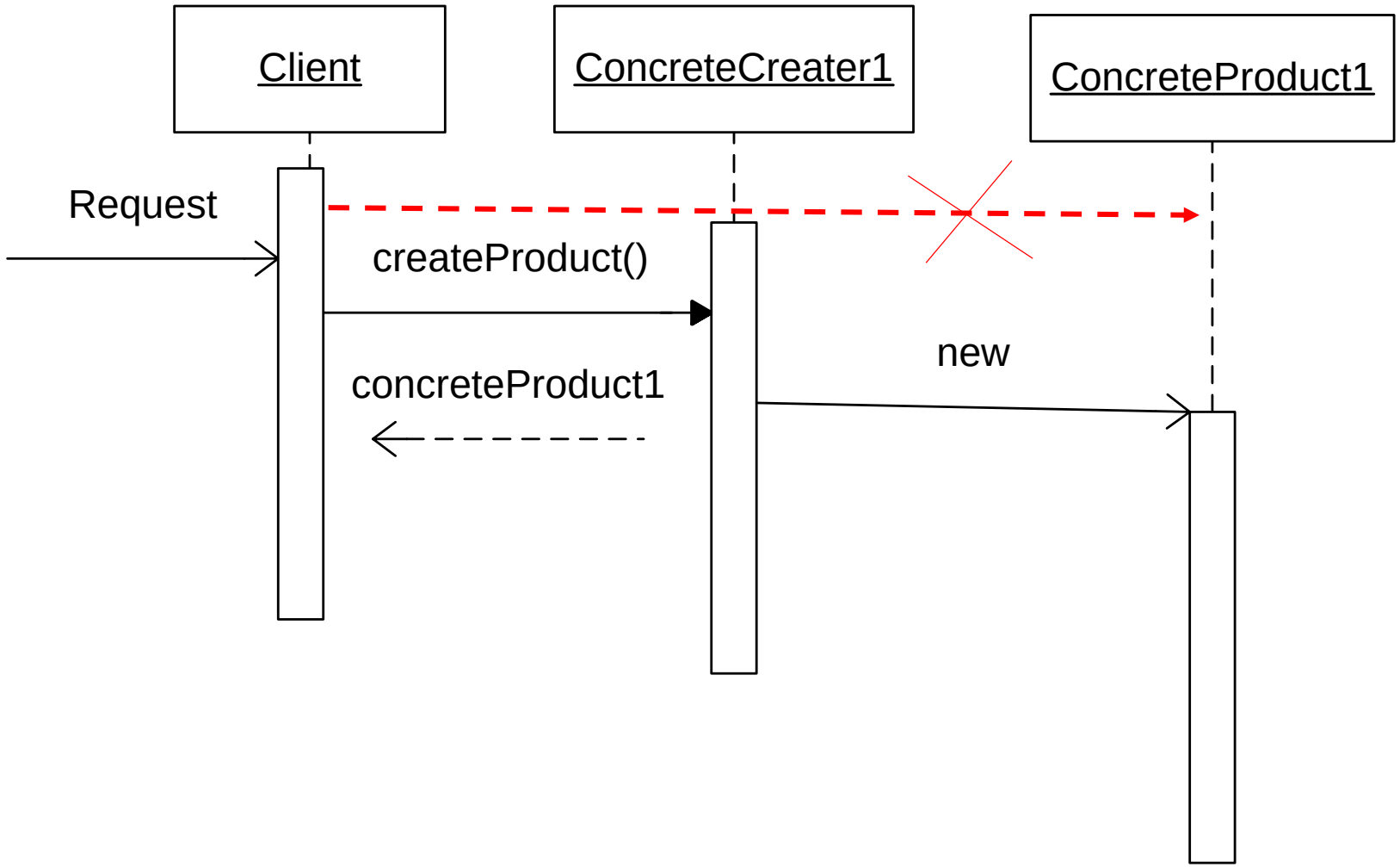

PizzaStore
SimplePizzaFactory
Pizza
CheesePizza
VeggiePizza
PepperoniPizza
ClamPizza

Factory Method: class diagram

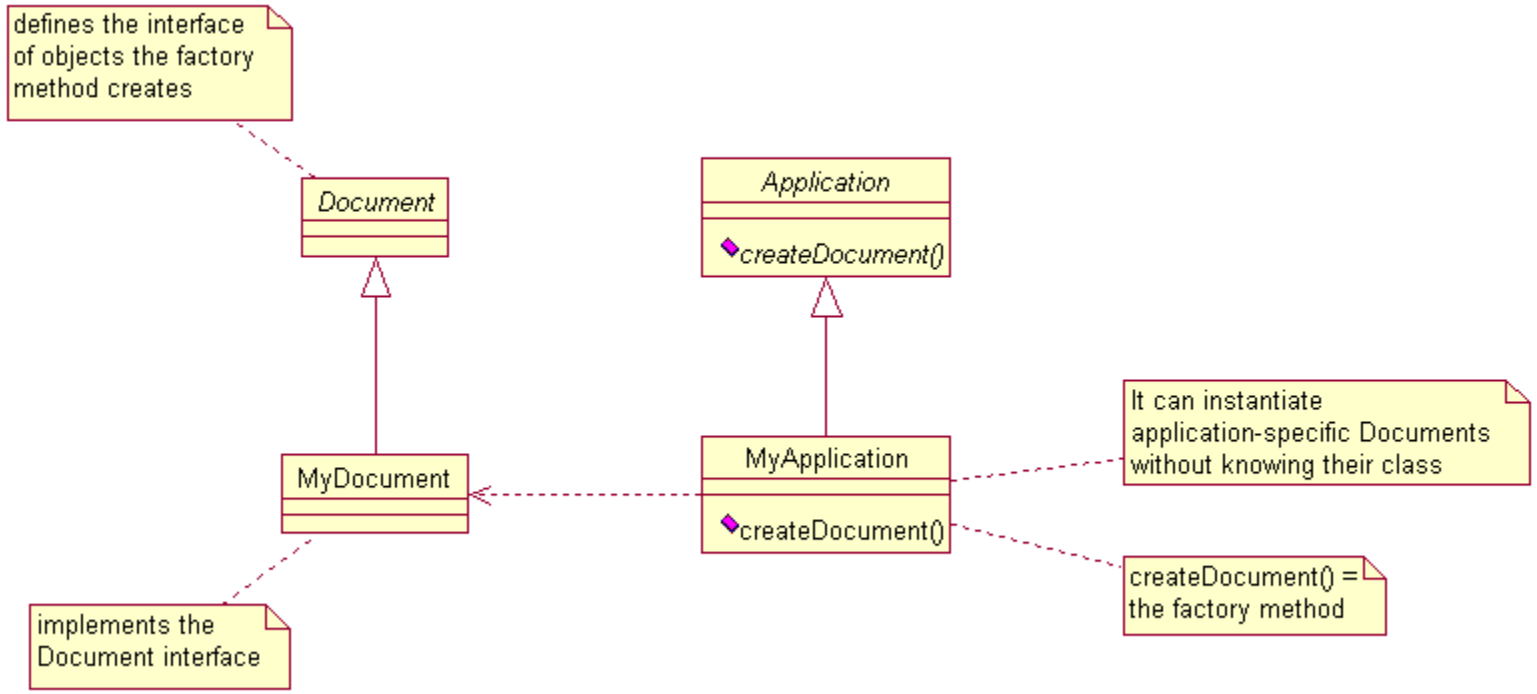


- the subclasses redefine abstract methods of the abstract class to return the appropriate subclass

Factory Method Sequence Diagram



Factory Method: class diagram sample



- we call createDocument() the factory method because it is responsible for "manufacturing" an object

Factory Method pattern

- applicabilities:
  - the class that must instantiate classes only knows about abstract classes, which it cannot instantiate. It only knows *when* or *how* to create an object but not *what kind* oh object to create, because this is application-specific
  - a class want its subclasses to specify the objects to be created
  - classes delegate responsibility to one or several helper subclasses and you want to localize the knowledge of which helper subclass is the delegate

Factory Method pattern

- CONSEQUENCES:
  - Factory methods eliminate the need to bind application-specific classes into your code
  - The code only deals with the Product interface and then it can work with any user-defined ConcreteProduct class
  - Clients might have to subclass the Creator class to create a particular ConcreteProduct object