Factory Pattern

CS342 Fall ‘16

Scenario

You opened a pizza shop and thanks to your amazing software that automates all of the pizza operations, business is booming. You have a class PizzaShop with a method called orderPizza that will prepare, bake, cut, box, and deliver the pizza object.

Your shop originally only made cheese pizza, but now you want to expand to pepperoni and greek pizzas.

Problem

Design a method orderPizza() such that

- it can be passed a string with the “type” of pizza (greek, cheese, pepperoni)
- Depending on the type passed as a string, the method instantiates the right kind of pizza object
- The pizza then needs to be prepared(), baked(), cut(), boxed()
- After that, the orderPizza() method returns the pizza

A Solution (but not a good one)

```
class PizzaShop
        ...
        def orderPizza ( pizzaType)
                if (pizzaType == “cheese”)
                        pizza = CheesePizza().new
                elsif (pizzaType == “greek”)
                        pizza = GreekPizza().new
                elsif (pizzaType == “pepperoni”)
                        pizza = PepperoniPizza().new
                pizza.prepare().bake().cut().box()
        end
end
```

Separate the changes from static

- Let’s identify what changes from what stays the same on the orderPizza() method
  - Object creation and processing stays the same
  - the type of object created changes
- The PizzaShop’s job is to create and process pizza objects, not determine what kind of objects should be created
  - Push the specialized object creation onto subclasses
    - def orderPizza ()

      @pizza = newPizza()

      pizza.prepare().bake().cut().box()

PizzaShops

Object Creation Classes

```
class PizzaShop
        def orderPizza ()
                @pizza = newPizza()
                pizza.prepare().bake().cut().box()
        end

        def newPizza()
                raise NotImplementedError.new
        end
end

class CheesePizzaShop < PizzaShop
        def newPizza
                CheesePizza.new
        end
end
```

Driver Code

```
cheesePizza = CheesePizzaShop.new.orderPizza()
```

Factory Method

- Define the newPizza in each of the the subclasses
  - CheesePizzaShop, PepperoniPizzaShop, GreekPizzaShop,
  - Each returns their own kind of pizza

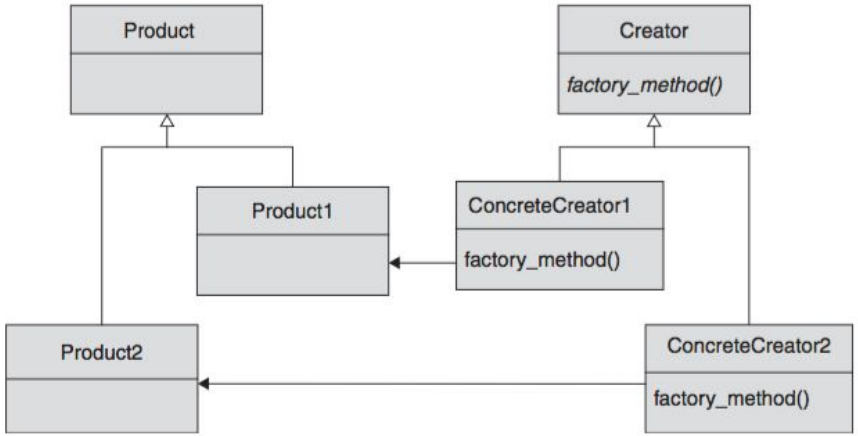

Figure 13-1 Class Diagram for the Factory Method pattern

Factory Method Class Structure

Object Creation Classes

```
class BaseFactory
        def createProduct ()
                product = newProduct()
                ...
        end
        def newProduct()
                raise NotImplementedError.new
        end
end
class ConcreteFactoryA < BaseFactory
        def newProduct
                ConcreteProductA.new
        end
end
class ConcreteFactoryB < BaseFactory
        def newProduct
                ConcreteProductB.new
        end
end
```

Product Classes

```
class BaseProduct
        …
end

class ConcreteProductA < BaseProduct
        …
end

class ConcreteProductB < BaseProduct
        ...
end
```

Factory Method seems familiar

- A series of steps with the specifics pushed onto a subclass? Where have we seen that before?
- Factory method is the template method applied to object creation
  - When you have a series of steps that will always need to be executed in the same order on a newly created object, Factory method is your pattern
- What is the fundamental problem we are solving?
  - We cannot dynamically decide what kind of class we will create
    - or can we...

The Ruby Way

- How can we alter our code to eliminate all those subclasses?
  - What if we could just tell the PizzaShop what kind of pizza class we wanted?
- Reflection is the object oriented concept of a class knowing what class it is
  - in other words, every object has an internal field keeping track of its own and parent class names
- Ruby implements this with the “is_a?” statement
  - it also has the “instance_of?” which tests the exact class, not superclasses

Ruby Classes are Objects

- Remember that all classes are Constants
  - Anything that starts with a capital letter is a constant
- Constants are known, static values from compile time
  - Constants are created in memory as soon as your program starts, and remain until it ends
- Ruby classes are objects with their own methods
  - new(), class()
- You can pass classes as parameters because they are objects
  - How can we use this to simplify the factory pattern?

Passing Class Type Objects as Parameters

- We can alter our orderPizza method to take a type parameter
  - def order(pizzaType)

    @pizza = pizzaType.new

    @pizza.prepare().bake().cut().box()

    end
  - PizzaShop.new.order(CheesePizza)
    - Eliminates the need for subclasses of PizzaShop

You've gone corporate

Your business takes off, and you expand to other food types. You change the name of your business (and class) to 'GoodEats.' You can now make different kinds of Burritos.

However, now the steps in creating the food is different. A burrito is filled, rolled, and wrapped in foil. How can we ensure the object gets created properly?

One Who Knows

- Instead of passing the food classes to GoodEats, we can pass a single factory object that knows how to create a product.
  - We have one version of the object for Pizzas and another for Burritos
    - class CheesePizzaFactory

```
def newFood
    @food = CheesePizza.new
end
def prepare()
    @food.bake().cut().box()
end
```

```
class BurritoFactory
    def newFood
        @food = Burrito.new
    end
    def prepare()
        @food.fill().roll().wrap()
    end
end
```

Abstract Factory Pattern

- We can now pass a factory to our order() method and guarantee the correct object gets created
  - def order(foodFactory)

    foodFactory.newFood()

    foodFactory.prepare()

    end
  - GoodEats.new.order(BurritoFactory.new)
- We are an extra layer to createa family of objects

Family of Objects the Ruby Way

- Just like the Factory Method, we can simplify the Abstract Factory to create only specific type objects
  - Accomplish this by, again, passing the Class Constant as a parameter
  - As long as the type of object derives from the same base class, we can use reflection to ensure the right type of object is selected
    - class PizzaFactory

```
def newFood(FoodType)
    obj = FoodType.new
    if obj.is_a? “Food” return obj #could be CheesePizza or Burrito
    else raise TypeError.new
```

Factory Method vs Abstract Factory

- Factory Method
  - If part of a class's job is creating objects of a certain base class, but not what kind of subclass to create
  - A single method in the subclass Factory does all of the object creation
- Abstract Factory
  - The class is responsible for creating several different objects within a specific set of objects
  - You need several methods for creating different kinds of objects, but they are limited to a certain type

Factory Method and Abstract Factory

- Factory Method and Abstract Factory takes Template Method and Strategy and apply it to object creation
- Problems with the factory
  - YAGNI principle (You ain't going to need it)
    - Perhaps I am dealing with only pizza and beer at the moment, but maybe in the future I might need to cope with burritos and wines. Should I build a factory now to get ready? Probably not.
    - Don't build a 50 bedroom mansion for 1 person.
  - Ruby doesn't lend itself to Factories due to duck typing and Convention over Configuration Principles
    - Factories partly solve type problems
    - The first of several patterns we will see that begin to break down in Ruby

Classwork:

Toys

```
class ToyFactory
        def makeToy
                toy_mold = getToyMold()
                toy_mold.mix.inject.package.ship()
        end
end

class ToyCarFactory < ToyFactory
        def getToyMold
                CarMold.new
        end
end

toycar = ToyCarFactory.maketoy
```