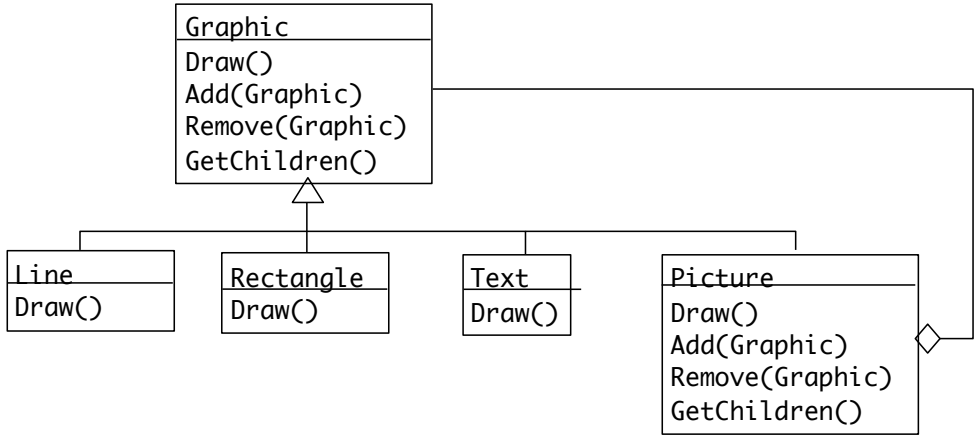
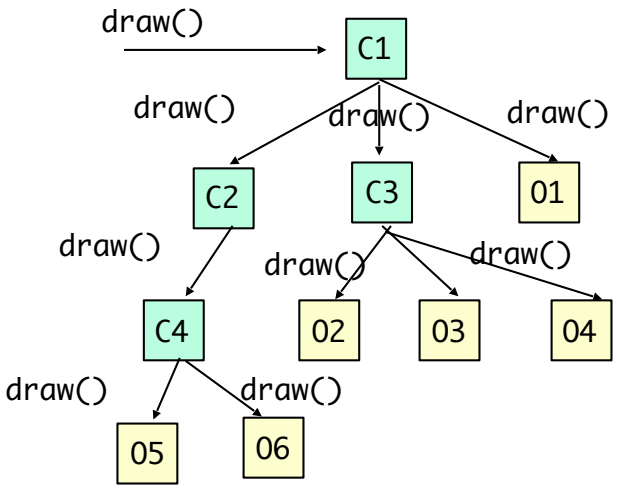
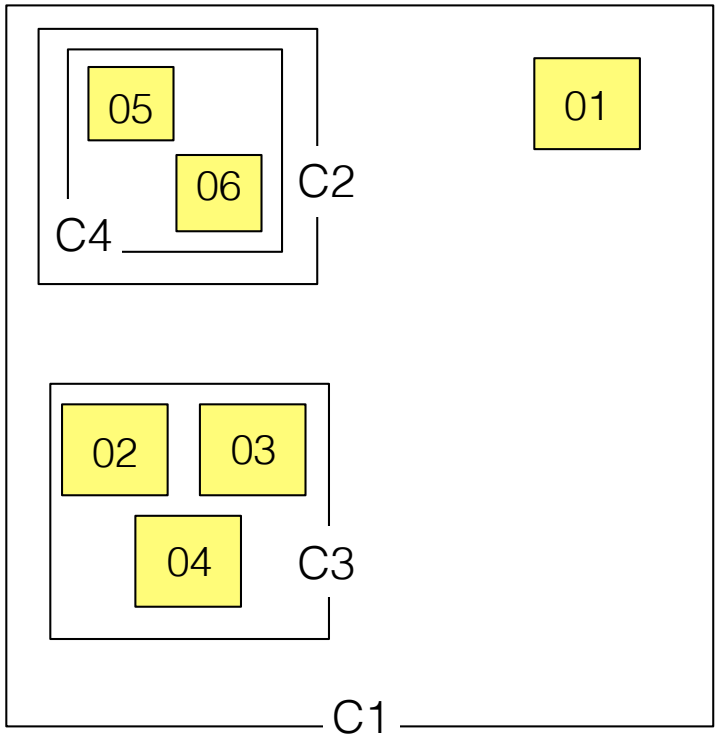
El Patrón Composite

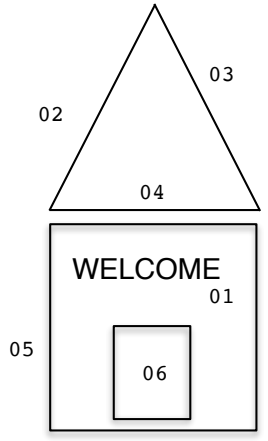
- Manejo de objetos que tienen estructuras jerárquicas de forma que una subestructura (incluso un nodo) se maneje igual que la estructura completa

Ejemplo con Objetos Gráficos

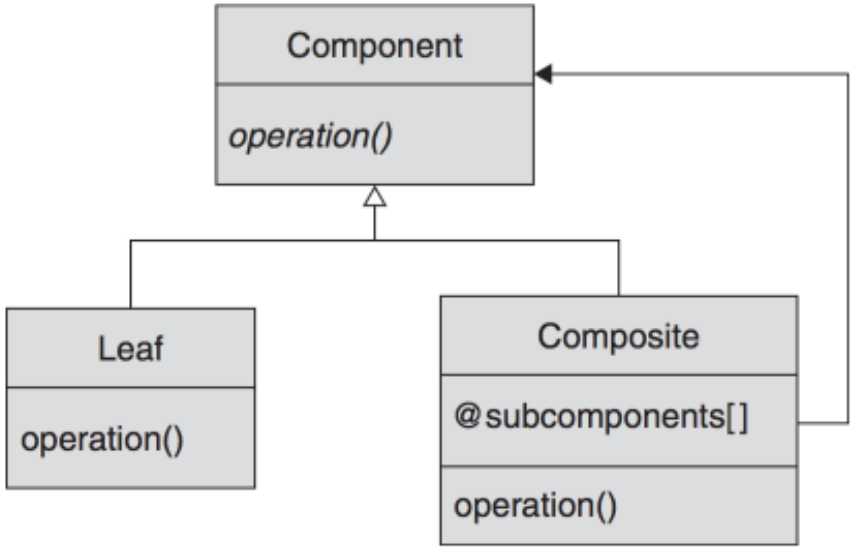








Estructura del Patrón

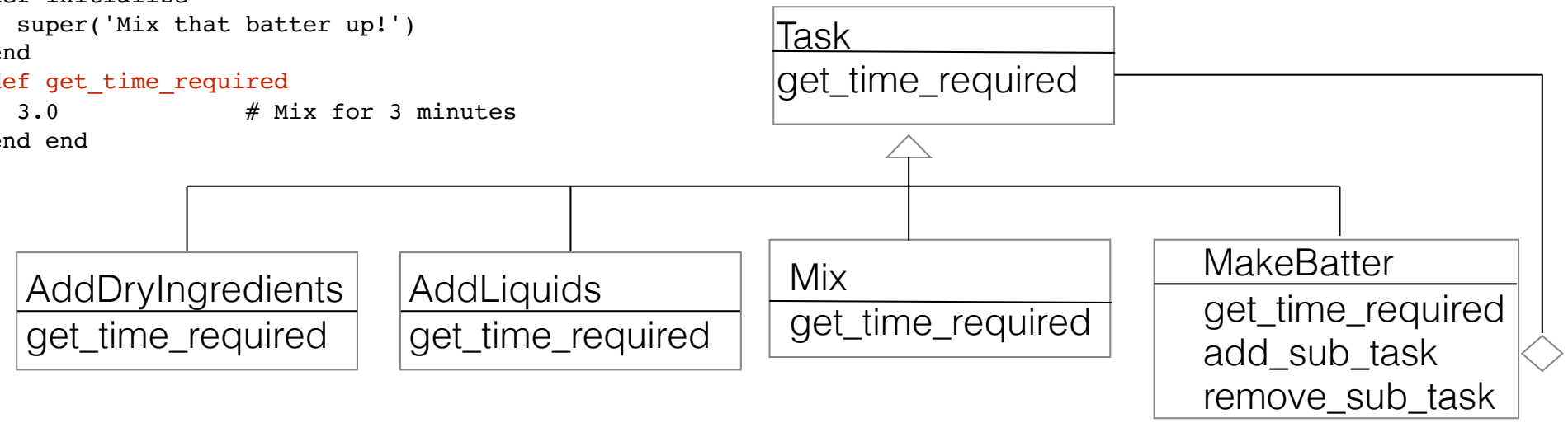


- Component contiene la interfaz base de todos los objetos (qué hay en común entre objetos simples y compuestos)
- Las "hojas" representan los objetos elementales y deben implementar la interfaz de Component
- Composite es también una componente pero contiene subcomponentes

Haciendo una torta

```
class Task
  attr_reader :name
  def initialize(name)
    @name = name
end
  def get_time_required
    0.0
  end
end
class AddDryIngredients < Task
  def initialize
    super('Add dry ingredients')
  end
  def get_time_required
    1.0              # 1 minute to add flour and sugar
  end
end
class AddLiquids < Task
.  def initialize
    super('Add liquids')
  end
  def get_time_required
    2.0              # 2 minutes to add milk
  end
end
class Mix < Task
  def initialize
    super('Mix that batter up!')
  end
  def get_time_required
    3.0              # Mix for 3 minutes
  end end
```

```
class MakeBatter < Task
  def initialize
    super('Make batter')
    @sub_tasks = []
    add_sub_task( AddDryIngredients.new )
    add_sub_task( AddLiquids.new )
    add_sub_task( Mix.new )
  end
  def add_sub_task(task)
    @sub_tasks << task
  end
  def remove_sub_task(task)
    @sub_tasks.delete(task)
  end
  def get_time_required
    time=0.0
    @sub_tasks.each {|task| time += task.get_time_required}
    time
  end
end
```



Transformando Acciones en Objetos

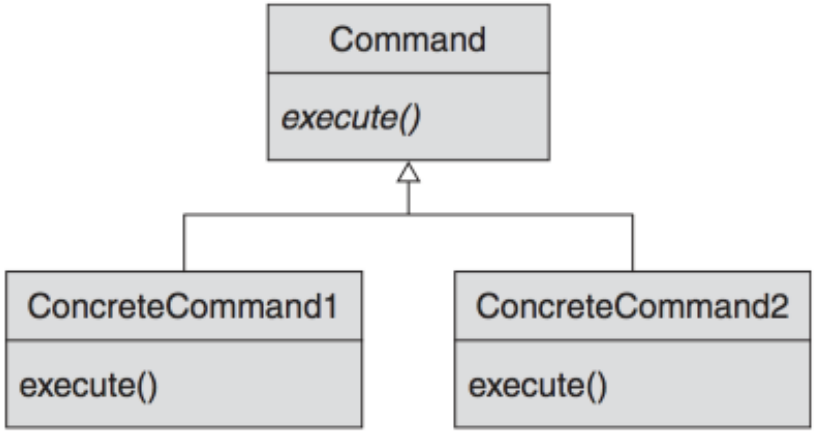
- Imagina la implementación de "undo" de una aplicación en que se editan objetos gráficos
- Hay acciones como agrandar, rotar, trasladar, etc.
- La idea del patrón comando es encapsular esas acciones como objetos
- Undo consiste simplemente en aplicar los objetos en el orden inverso

Muchos botones distintos ...

```
class SlickButton
  #
  # Lots of button drawing and management
  # code omitted...
  #
  def on_button_push
    #
    # Do something when the button is pushed
    #
  end
end

class SaveButton < SlickButton
  def on_button_push
    #
    # Save the current document...
    #
  end
end

class NewDocumentButton < SlickButton
  def on_button_push
    #
    # Create a new document...
    #
  end
end
```

Comando (command)

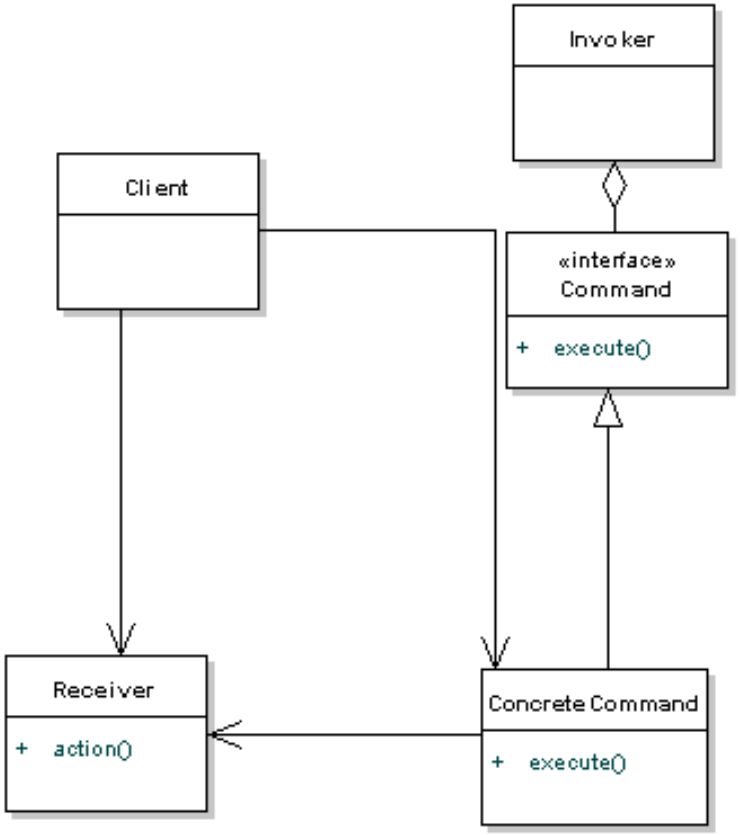
```
class SlickButton
  attr_accessor :command
  def initialize(command)
    @command = command
  end

  def on_button_push
    @command.execute if @command
    end
  end

class SaveCommand
  def execute
    #
    # Save the current document...
    #
  end
end

save_button = SlickButton.new( SaveCommand.new )
```

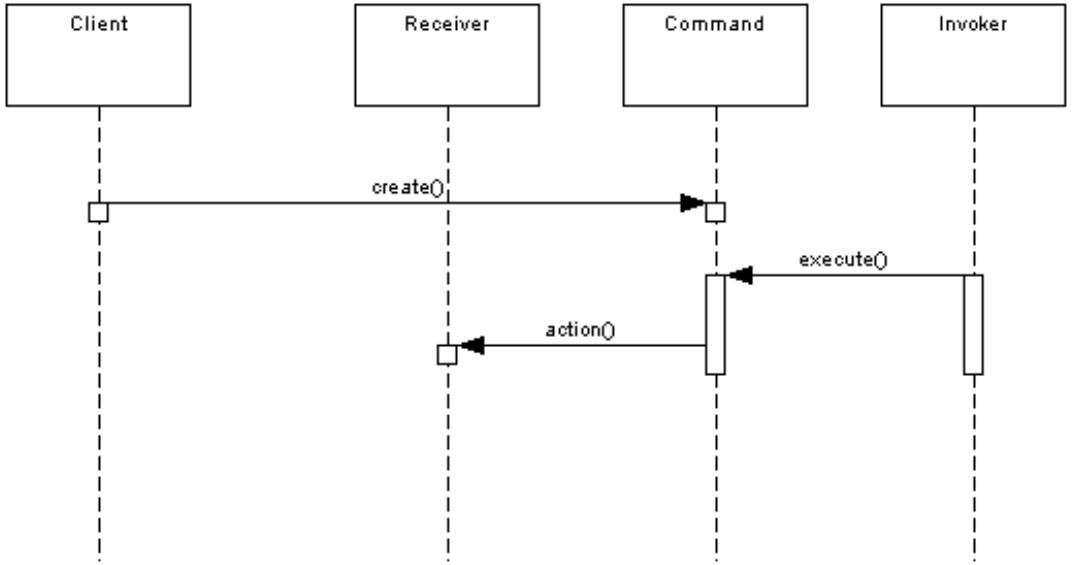


Esencia del Patrón *Command*

- se representa una acción como un objeto
- cliente que requiere ejecutar el comando se desacopla de detalles y dependencias de la lógica del comando
- permite ejecución no inmediata (cola)
- pueden guardarse acciones en caso de un restart
- permite implementar undo

El Comando en GoF





En Ruby pueden usarse blocks

```
class SlickButton
  attr_accessor :command
  def initialize(&block)
    @command = block
  end
  #
  # Lots of button drawing and management
  # code omitted...
  #
  def on_button_push
    @command.call if @command
  end
end

new_button = SlickButton.new do
  #
  # Create a new document...
  #
end
```

Ejemplo: Comandos para Manejo de Archivos

```
class Command
  attr_reader :description
  def initialize(description)
    @description = description
  end
  def execute
  end
end
```

```
class CreateFile < Command
  def initialize(path, contents)
    super("Create file: #{path}")
    @path = path
    @contents = contents
  end
  def execute
    f = File.open(@path, "w")
    f.write(@contents)
    f.close
  end
end
```

```
class DeleteFile < Command
  def initialize(path)
    super("Delete file: #{path}")
    @path = path
  end
  def execute
    File.delete(@path)
  end
end
```

```
class CopyFile < Command
  def initialize(source, target)
    super("Copy file: #{source} to #{target}")
    @source = source
    @target = target
  end
  def execute
    FileUtils.copy(@source, @target)
  end
end
```

Patrones Creacionales: Factories

```
class Duck
  def initialize(name)
    @name = name
  end
  def eat
    puts("Duck #{@name} is eating.")
  end
  def speak
    puts("Duck #{@name} says Quack!")
  end
  def sleep
    puts("Duck #{@name} sleeps quietly.")
  end
end


class Pond
  def initialize(number_ducks)
    @ducks = []
    number_ducks.times do |i|
      duck = Duck.new("Duck#{i}")
      @ducks << duck
    end
  end
  def simulate_one_day
    @ducks.each {|duck| duck.speak}
    @ducks.each {|duck| duck.eat}
    @ducks.each {|duck| duck.sleep}
  end
end


pond = Pond.new(3)
pond.simulate_one_day
```

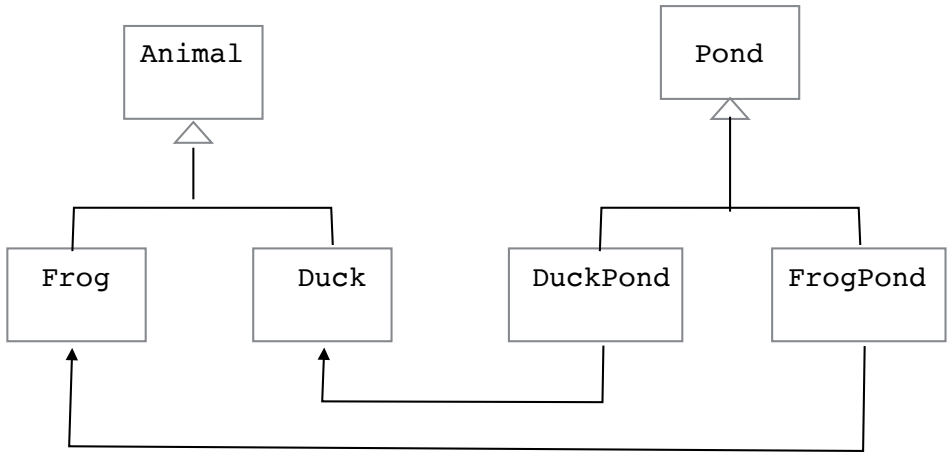
```
Duck Duck0 says Quack!
Duck Duck1 says Quack!
Duck Duck2 says Quack!
Duck Duck0 is eating.
Duck Duck1 is eating.
Duck Duck2 is eating.
Duck Duck0 sleeps quietly.
Duck Duck1 sleeps quietly.
Duck Duck2 sleeps quietly.
```

Pond solo sirve para patos!

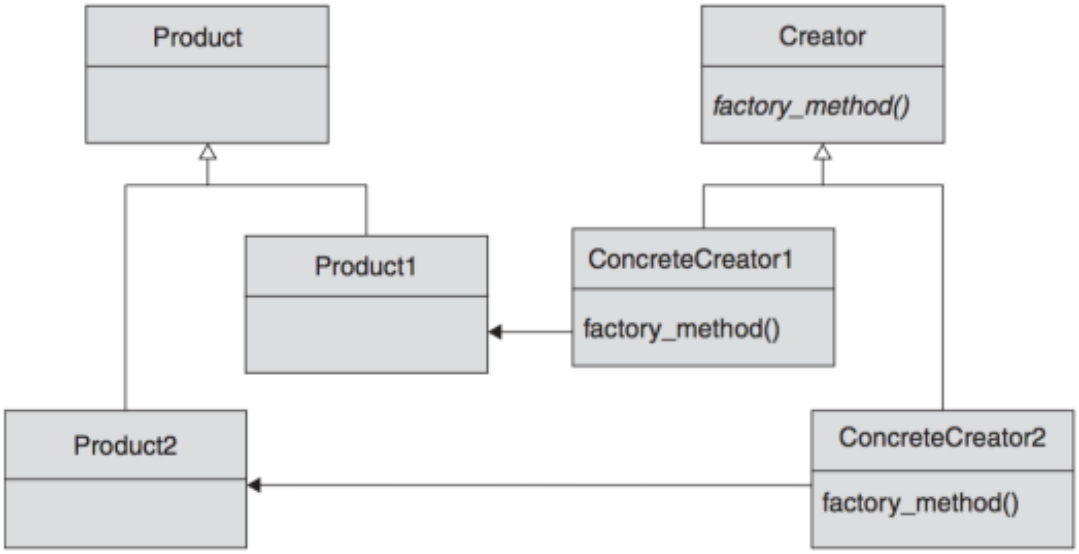
Una laguna de sapos ?

Mejor Solución

```
class Pond
  def initialize(number_animals)
    @animals = []
    number_animals.times do |i|
      animal = new_animal("Animal#{i}")
      @animals << animal
    end
  end
  def simulate_one_day
    @animals.each {|animal| animal.speak}
    @animals.each {|animal| animal.eat}
    @animals.each {|animal| animal.sleep}
  end
end


class DuckPond < Pond
  def new_animal(name)
    Duck.new(name)
  end
end

class FrogPond < Pond
  def new_animal(name)
    Frog.new(name)
  end
end


pond = FrogPond.new(3)
pond.simulate_one_day
```

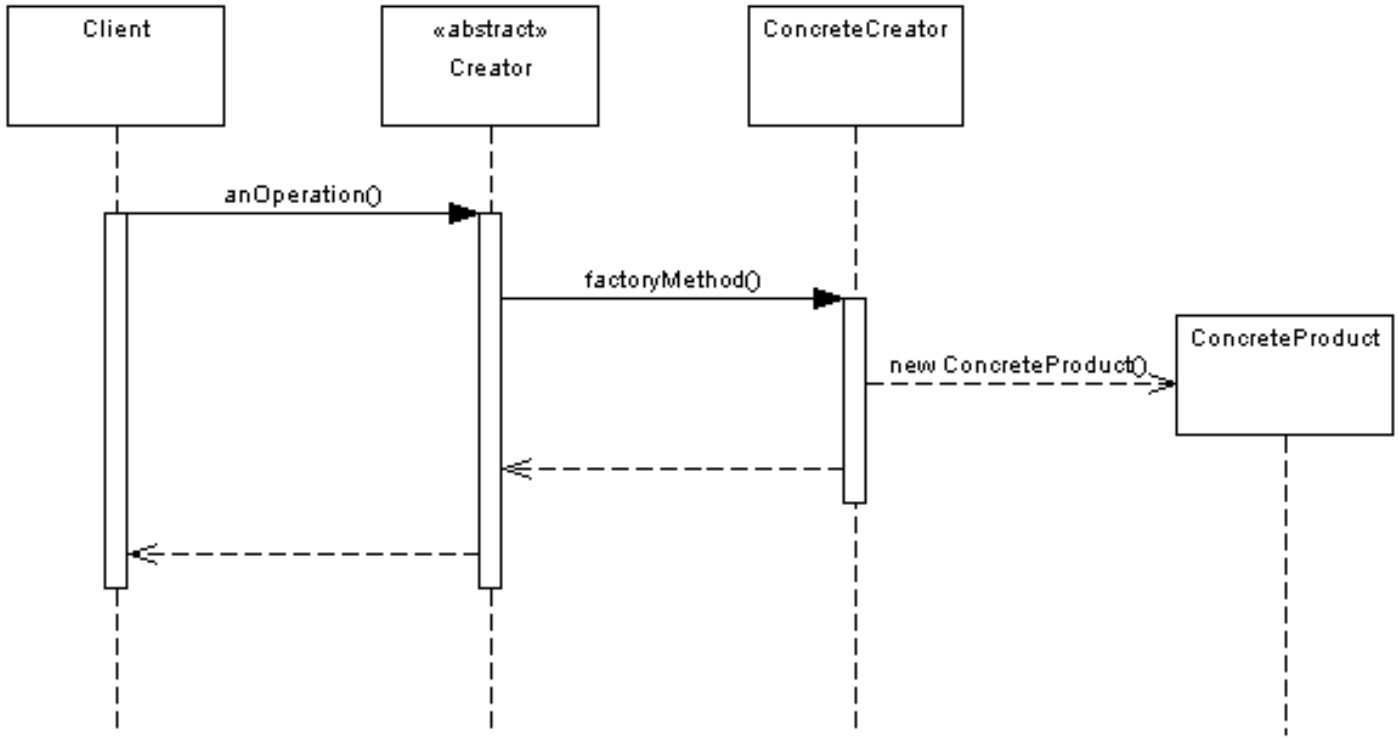


Factory Method



Este patrón es en realidad el patrón template method aplicado al problema de creación de objetos

Diagrama de Secuencia



Variación del Problema

- El lago debe poder manejar plantas y animales
- Las plantas tienen solo método grow

```
class Pond
  def initialize(number_animals, number_plants)
    @animals = []
    number_animals.times do |i|
       animal = new_animal("Animal#{i}")
       @animals << animal
    end
    @plants = []
    number_plants.times do |i|
       plant = new_plant("Plant#{i}")
      @plants << plant
    end
  end
  def simulate_one_day
    @plants.each {|plant| plant.grow }
    @animals.each {|animal| animal.speak}
    @animals.each {|animal| animal.eat}
    @animals.each {|animal| animal.sleep}
  end
end
```

Factory Method Parametrizado

```
class Pond
  def initialize(number_animals, number_plants)
    @animals = []
    number_animals.times do |i|
      animal = new_organism(:animal, "Animal#{i}")
      @animals << animal
    end
    @plants = []
    number_plants.times do |i|
      plant = new_organism(:plant, "Plant#{i}")
      @plants << plant
    end
  end
# ... end

class DuckWaterLilyPond < Pond
  def new_organism(type, name)
    if type == :animal
      Duck.new(name)
    elsif type == :plant
      WaterLily.new(name)
    else
      raise "Unknown organism type: #{type}"
    end
  end
end
```

y si no son solo lagunas ?

- Queremos simular distintos habitats: lagunas y junglas
- En cada habitat hay distintos tipos de habitantes
- Patrón Fábrica Abstracta, capaz de crear un conjunto de objetos de clases relacionadas
- Un caso más real podría ser una fábrica que genera objetos de UI para MacOS otra para Windows, etc

El Habitat

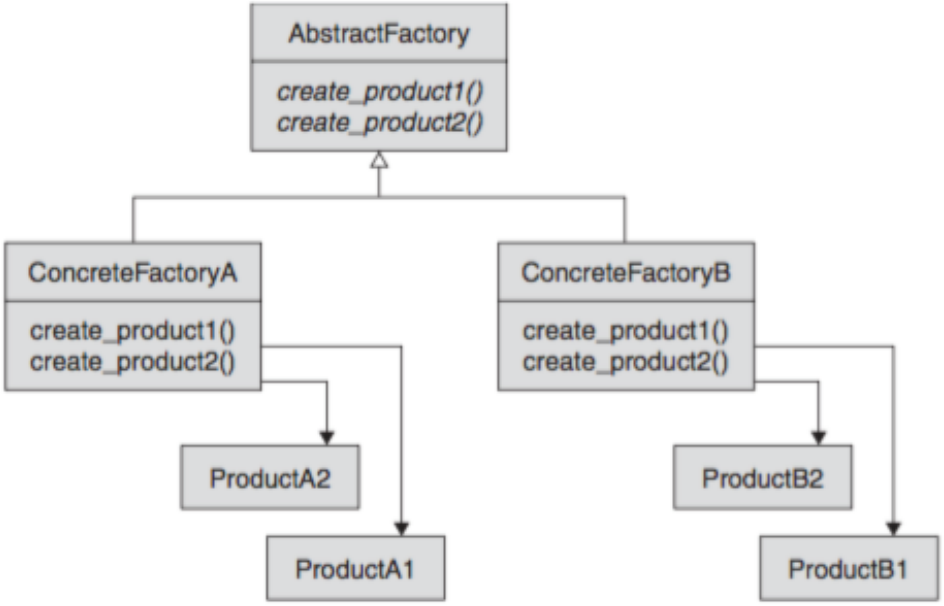
```
class PondOrganismFactory
  def new_animal(name)
    Frog.new(name)
  end
  def new_plant(name)
    Algae.new(name)
end end
```

```
class JungleOrganismFactory
  def new_animal(name)
    Tiger.new(name)
  end
  def new_plant(name)
    Tree.new(name)
  end
end
```

```
class Habitat
  def initialize(number_animals, number_plants, organism_factory)
    @organism_factory = organism_factory
    @animals = []
    number_animals.times do |i|
      animal = @organism_factory.new_animal("Animal#{i}")
      @animals << animal
    end
    @plants = []
    number_plants.times do |i|
      plant = @organism_factory.new_plant("Plant#{i}")
      @plants << plant
    end
  end
  # Rest of the class...
```

```
jungle = Habitat.new(1, 4, JungleOrganismFactory.new)
jungle.simulate_one_day
pond = Habitat.new( 2, 4, PondOrganismFactory.new)
pond.simulate_one_day
```

En UML



- Dos factories concretas, cada una capaz de crear su propio set de productos
- Es la misma idea del patrón Estrategy pero aplicado al problea de crear objetos

Una empresa produce 3 productos distintos: A, B y C. Sin embargo cada uno de ellos los produce en dos versiones, una proveniente de la fábrica local de alta calidad y una proveniente de una fábrica en China que los produce a bajo costo. Sólo hay una fábrica en cada uno de estos sitios. Queremos simular esta fábrica usando uno o mas patrones de diseño de los aprendidos en clase.

Es muy probable que Ud estime conveniente usar la fábrica abstracta para producir las dos fábricas concretas que se describen. En este caso sería necesario asegurar que hay solo una fábrica concreta de cada una de ellas (hay un patrón para ello)

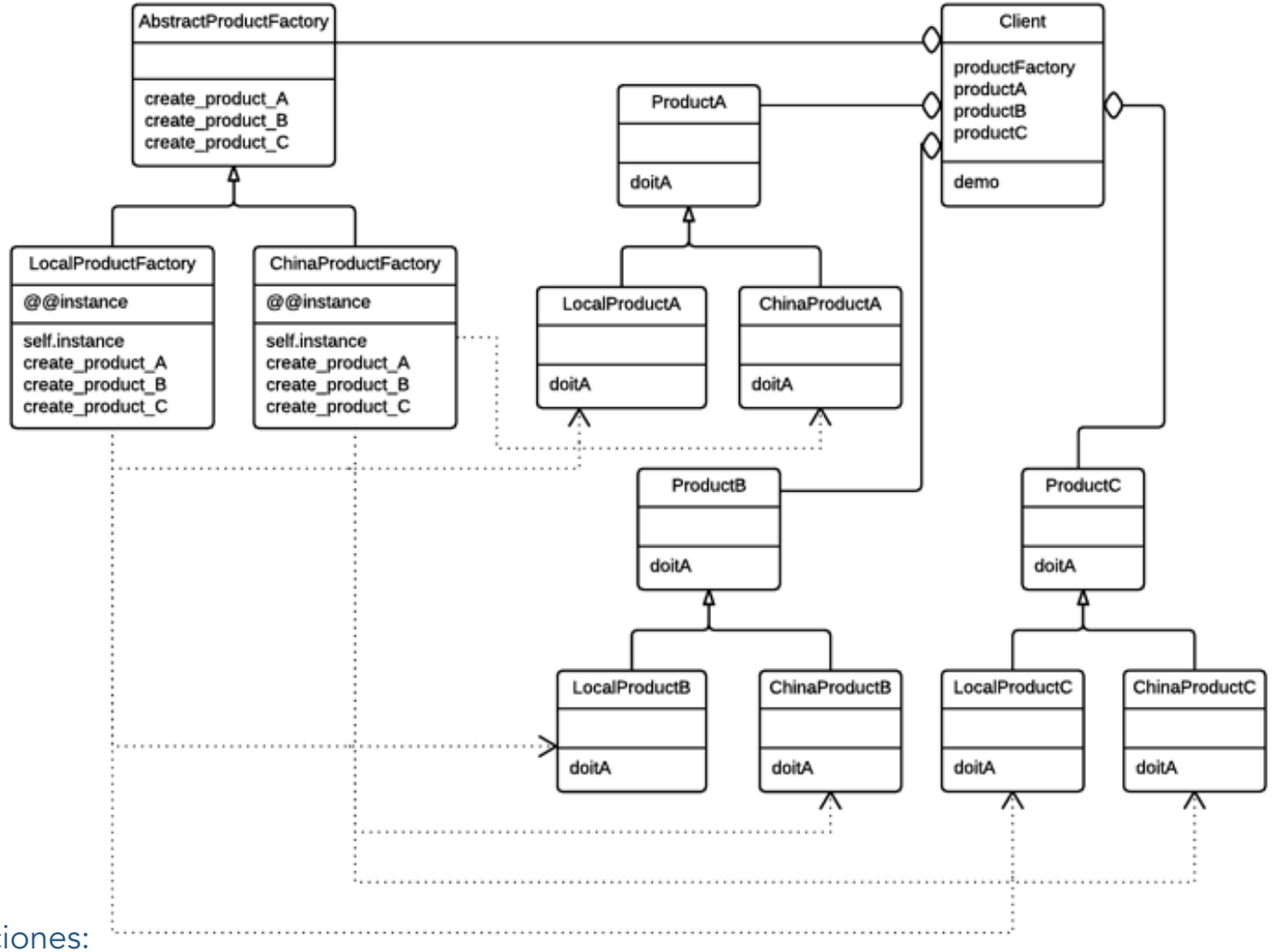
Los productos A incluyen método doitA, los productos B un doitB y los C un doitC.

a) Haga un diagrama de clases UML que muestre en forma gráfica la solución

b) Escriba el código de la fábrica abstracta y de las dos fábricas concretas

c) Escriba una clase Cliente que se inicializa con una fábrica concreta la que utiliza para crear instancias de objetos A, B y C. Cliente contiene además un método demo que invoca doitA, doitB y doitC sobre estos objetos.

d) Escriba un segmento de código que crea un cliente para luego invocar el método demo, primero con una fábrica local y luego con la fábrica china.



Observaciones:

- Las fábricas concretas son implementadas como singleton y por eso aparece el atributo de clase instance y el método de clase instance
- Hay una fábrica abstracta de la cual heredan las dos fábricas concretas
- Hay una clase abstracta para cada producto y clases concretas para los productos producidos en las fábricas concretas
- Las líneas punteadas indican dependencia. Así los 3 productos concretos locales todos dependen de la fábrica local y los 3 productos concretos chinos dependen de la fábrica china

```
class AbstractProductFactory
 def create_product_A
   puts "You should implement this method in the concrete factory"
 end
 def create_product_B
   puts "You should implement this method in the concrete factory
 end
 def create_product_C
   puts "You should implement this method in the concrete factory"
 end
end

class LocalProductFactory  < AbstractProductFactory
 @@instance = LocalProductFactory.new    #requerido para singleton
 private_class_method :new               #requerido para singleton
 def self.instance                       #requerido para singleton
   return @@instance
 end
 def create_product_A
   LocalProductA.new
 end
 def create_product_B
   LocalProductB.new
 end
 def create_product_C
   LocalProductC.new
 end
end

class ChinaProductFactory  < AbstractProductFactory
 @@instance = ChinaProductFactory.new    #requerido para singleton
 private_class_method :new               #requerido para singleton
 def self.instance                       #requerido para singleton
   return @@instance
 end
 def create_product_A
   ChinaProductA.new
 end
 def create_product_B
   ChinaProductB.new
 end
 def create_product_C
   ChinaProductC.new
 end
end
```

```
class Client
 attr_accessor :productFactory
 def initialize(productFactory)
   @productFactory = productFactory
   @prod_A = @productFactory.create_product_A
   @prod_B = @productFactory.create_product_B
   @prod_C = @productFactory.create_product_C
 end
 def demo
   @prod_A.doitA
   @prod_B.doitB
   @prod_C.doitC
 end
end


cliente1 = Client.new(LocalProductFactory.instance)
cliente1.demo
#output de doitA de un producto A de la clase LocalProductA
#output de doitB de un producto B de la clase LocalProductB
#output de doitC de un producto C de la clase LocalProductC
cliente2 = Client.new(ChinaProductFactory.instance)
cliente2.demo
#output de doitA de un producto A de la clase ChinaProductA
#output de doitB de un producto B de la clase ChinaProductB
#output de doitC de un producto C de la clase ChinaProductC
```

Observaciones:

- nótese que cuando se crea el cliente se le pasa como parámetro una fábrica
- la fábrica no puede ser creada con new porque ese método está deshabilitado, sino con el método de clase instance (singleton)