

Decorador (Decorator)

- ▶ Problema: anexar responsabilidades en forma dinámica
- ▶ Estas funcionalidades deben ejecutar antes o después de la ya existente
- ▶ Serie de envoltorios agregan funcionalidades al objeto básico
- ▶ Clase abstracta representa clase original y nuevas funciones. En decoradores ubicar llamadas a funciones antes o después

Otro problema que herencia no resuelve satisfactoriamente

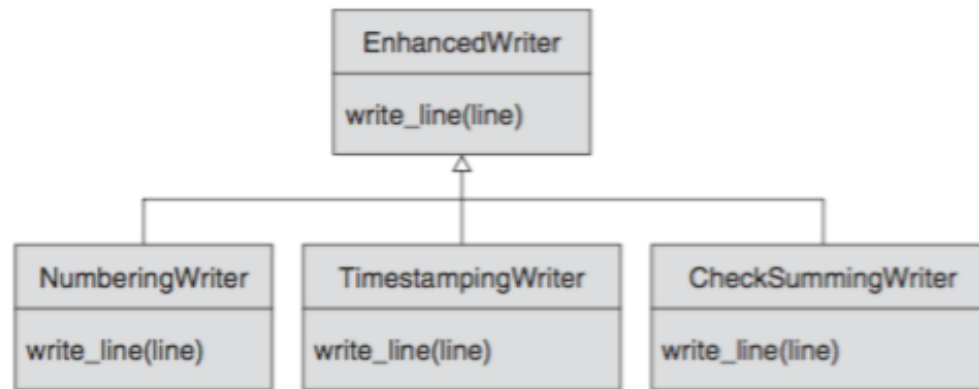
- ▶ Se quiere escribir un cierto texto en un archivo pero ...
 - ▶ a veces se quiere agregar un número de línea antes de escribirla
 - ▶ a veces se quiere poner un time stamp en la línea antes de escribirla

Solución naive

```
class EnhancedWriter
  attr_reader :check_sum
  def initialize(path)
    @file = File.open(path, "w")
    @check_sum = 0
    @line_number = 1
  end
  def write_line(line)
    @file.print(line)
    @file.print("\n")
  end
  def timestamping_write_line(data)
    write_line("#{Time.new}: #{data}")
  end
  def numbering_write_line(data)
    write_line("#{@line_number}: #{data}")
    @line_number += 1
  end
  def close
    @file.close
  end
end

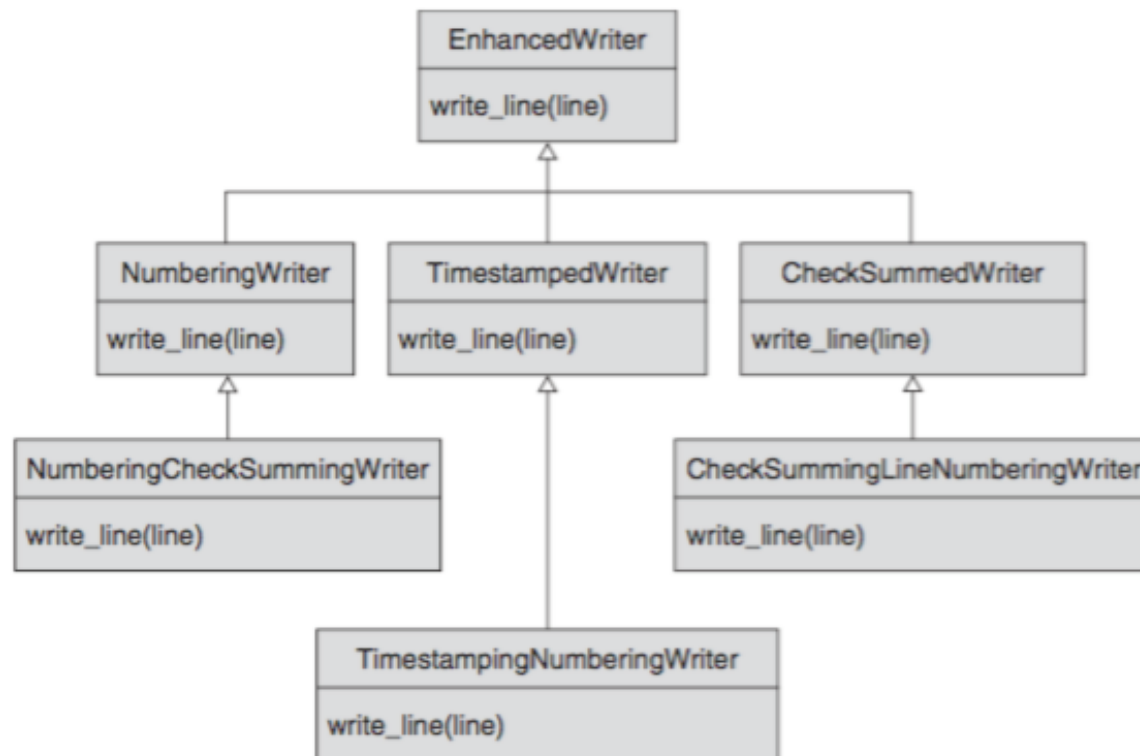
writer = EnhancedWriter.new('out.txt')
writer.write_line("A plain line")
writer.timestamping_write_line('with time stamp')
writer.numbering_write_line('with line number')
```

¿Por qué herencia no sirve ?



- ▶ Cómo lograr un texto numerado con checksum ?
- ▶ Cómo lograr uno con timestamp numerado ?
- ▶ etc

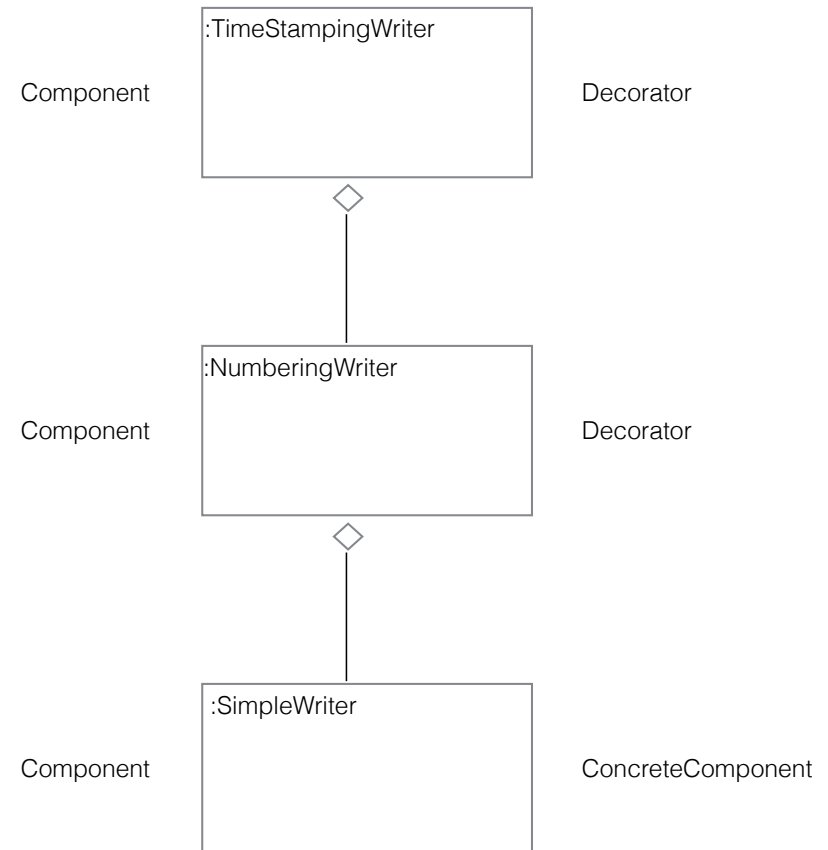
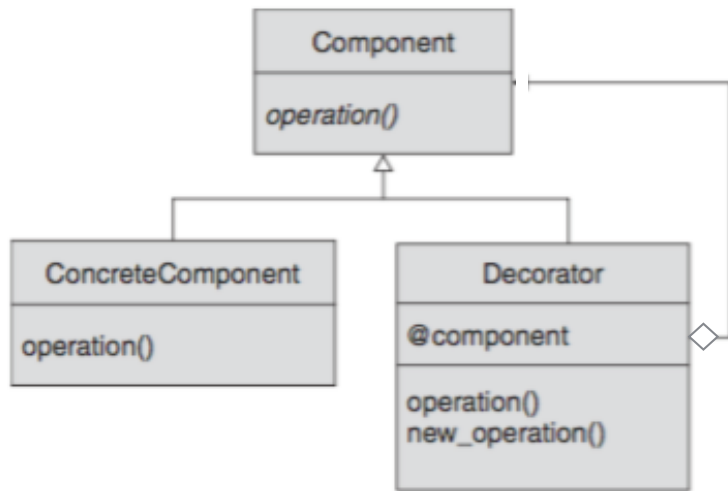
no va por aquí



La esencia del decorador

- ▶ Poder armar en runtime un objeto con la combinación de features deseada
- ▶ SimpleWriter simplemente escribe el texto en el archivo
- ▶ NumberingWriter agrega números al texto antes de pasarlo al Simple (decora a SimpleWriter)
- ▶ TimeStampingWriter agrega un time stamp a cada línea antes de pasarlo a simple

El patrón sintetizado



```

class SimpleWriter
  def initialize(path)
    @file = File.open(path, 'w')
  end
  def write_line(line)
    @file.print(line)
    @file.print("\n")
  end
  def pos
    @file.pos
  end
  def rewind
    @file.rewind
  end
  def close
    @file.close
  end
end

```

```

class WriterDecorator
  def initialize(real_writer)
    @real_writer = real_writer
  end
  def write_line(line)
    @real_writer.write_line(line)
  end
  def pos
    @real_writer.pos
  end
  def rewind
    @real_writer.rewind
  end
  def close
    @real_writer.close
  end
end

```

```

class NumberingWriter < WriterDecorator
  def initialize(real_writer)
    super(real_writer)
    @line_number = 1
  end
  def write_line(line)
    @real_writer.write_line("#{@line_number}: #{line}")
    @line_number += 1
  end
end

```

```

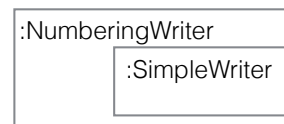
class TimeStampingWriter < WriterDecorator
  def write_line(line)
    @real_writer.write_line("#{Time.new}: #{line}")
  end
end

```

```

writer = NumberingWriter.new(SimpleWriter.new('final.txt'))
writer.write_line

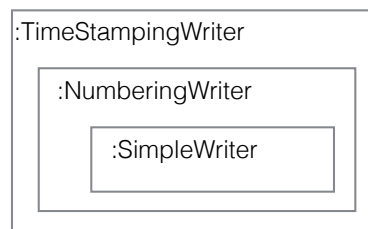
```



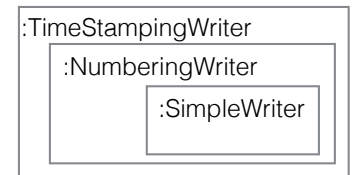
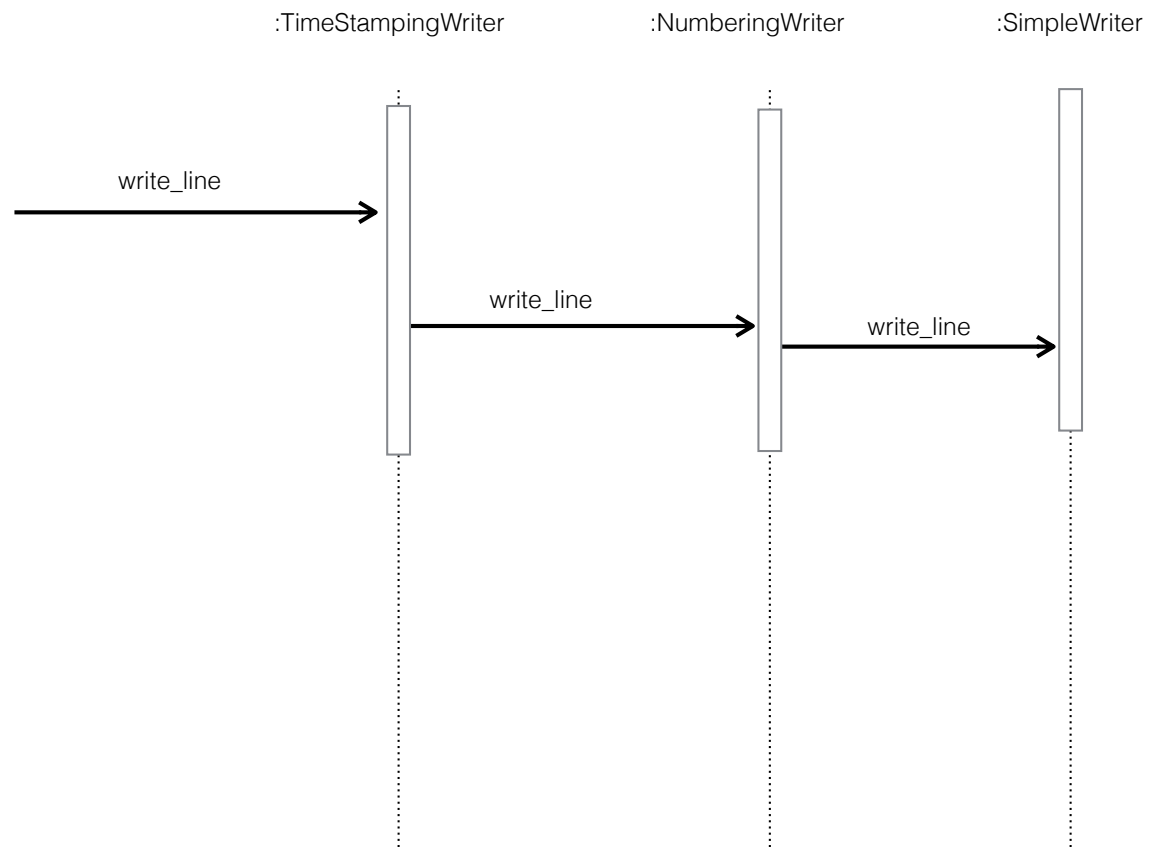
```

writer = TimeStampingWriter.new(
  NumberingWriter.new(
    SimpleWriter.new('final.txt')))
writer.write_line

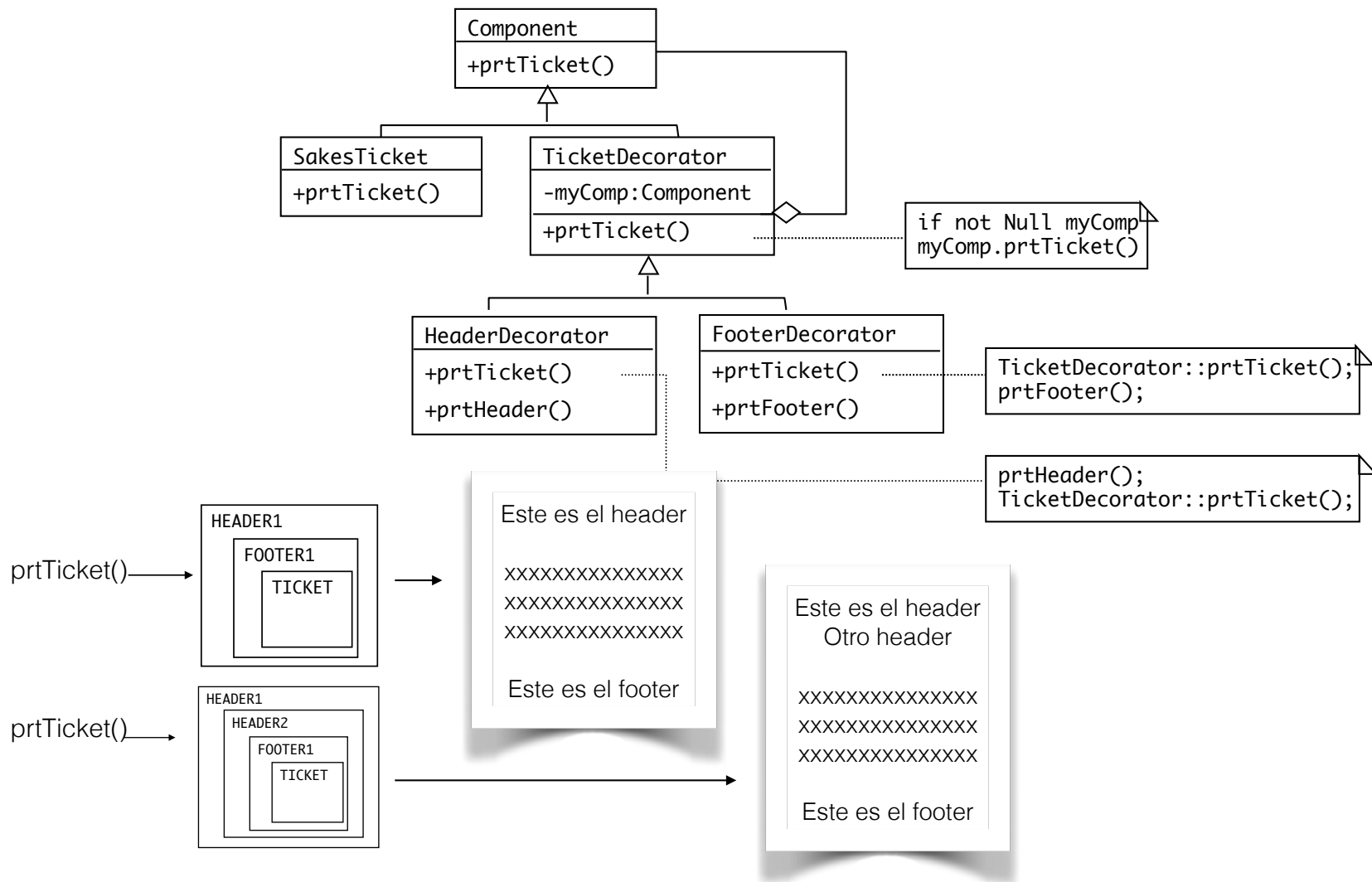
```



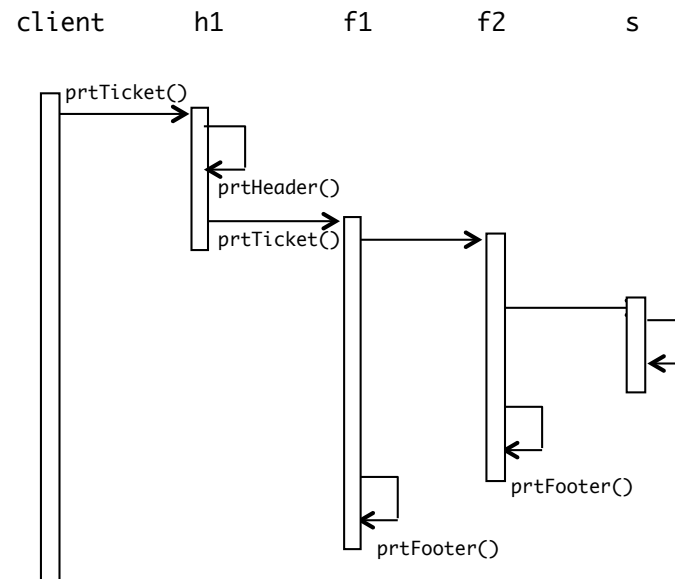
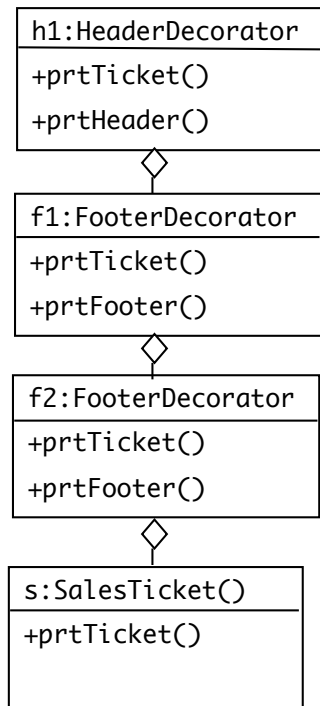
Funcionamiento



Otro Ejemplo



¿ Como funciona ?

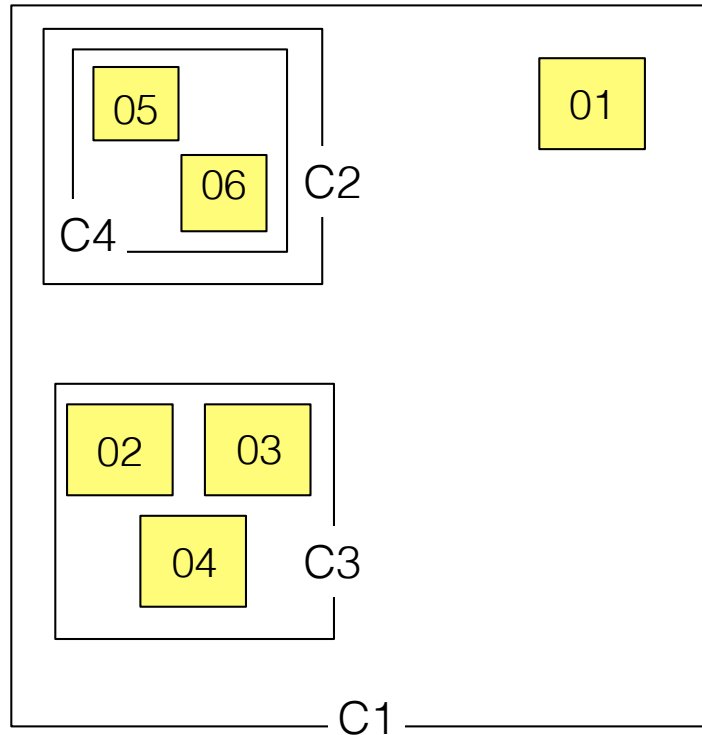
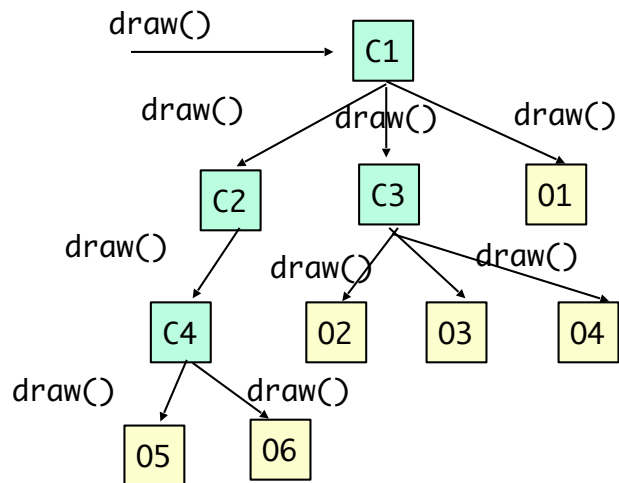
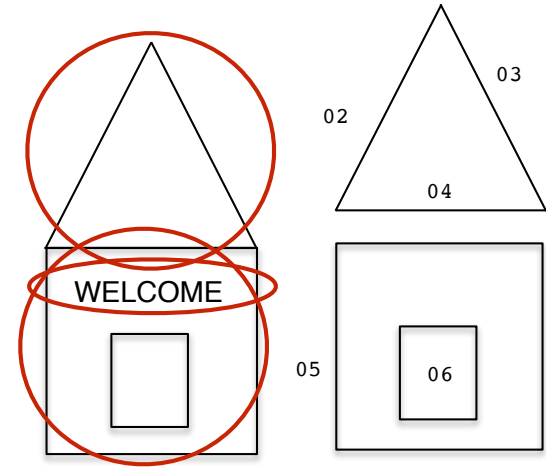
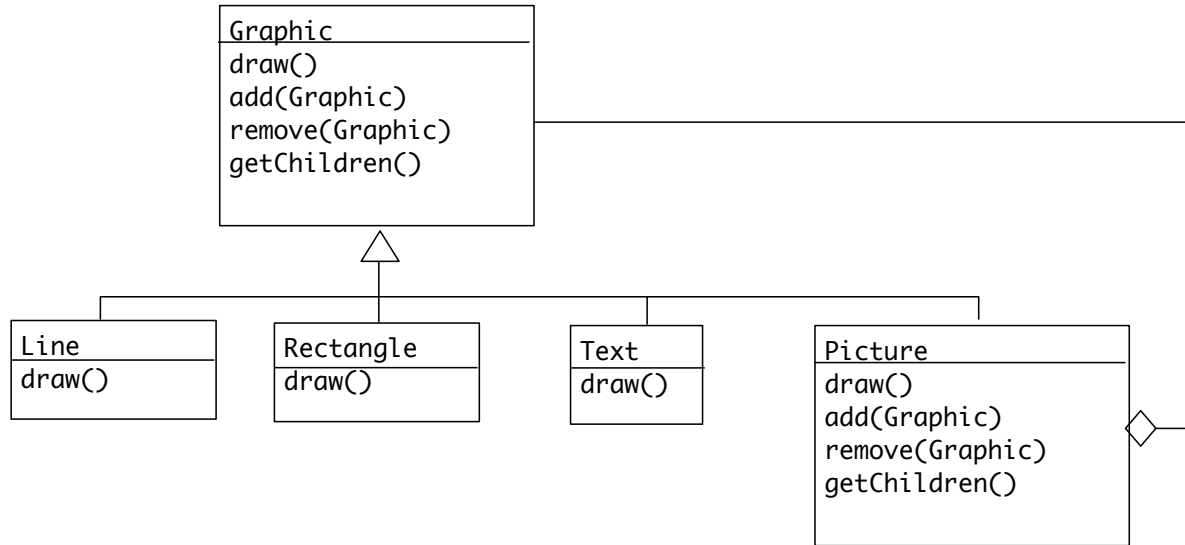


El Patrón Composite

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

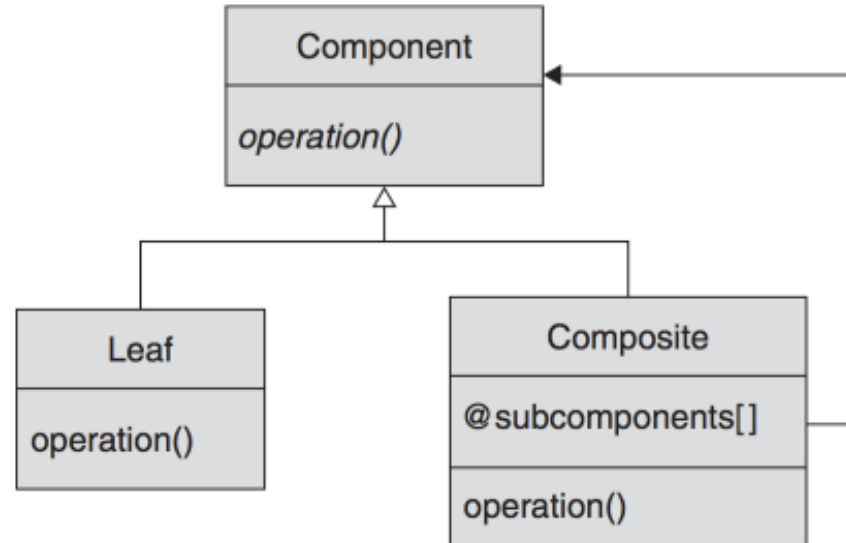


Ejemplo con Objetos Gráficos



WELCOME
01

Estructura del Patrón

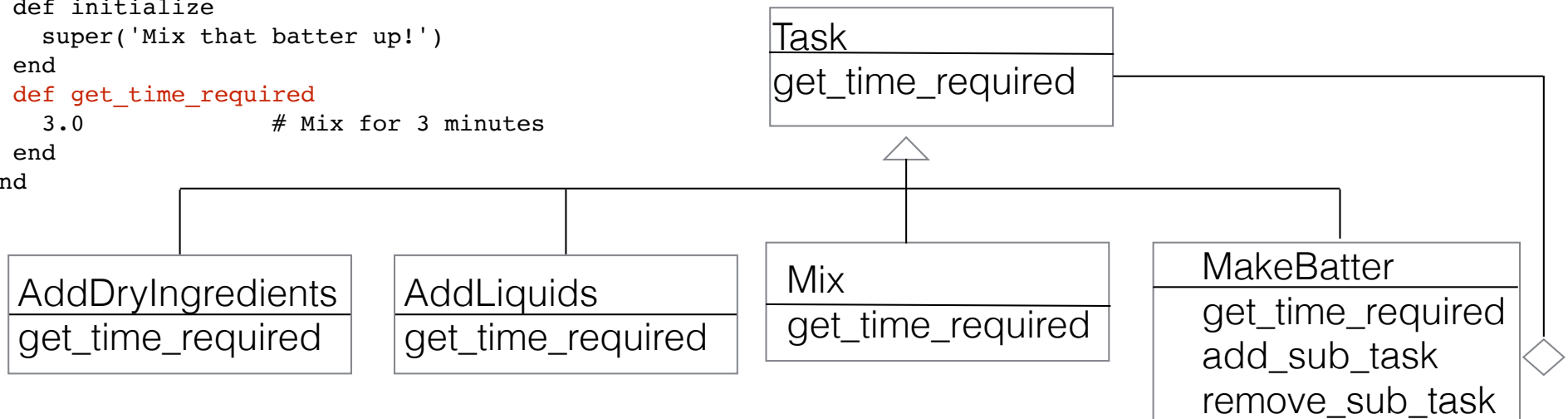


- ▶ Component contiene la interfaz base de todos los objetos (qué hay en común entre objetos simples y compuestos)
- ▶ Las "hojas" representan los objetos elementales y deben implementar la interfaz de Component
- ▶ Composite es también una componente pero contiene subcomponentes

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
```



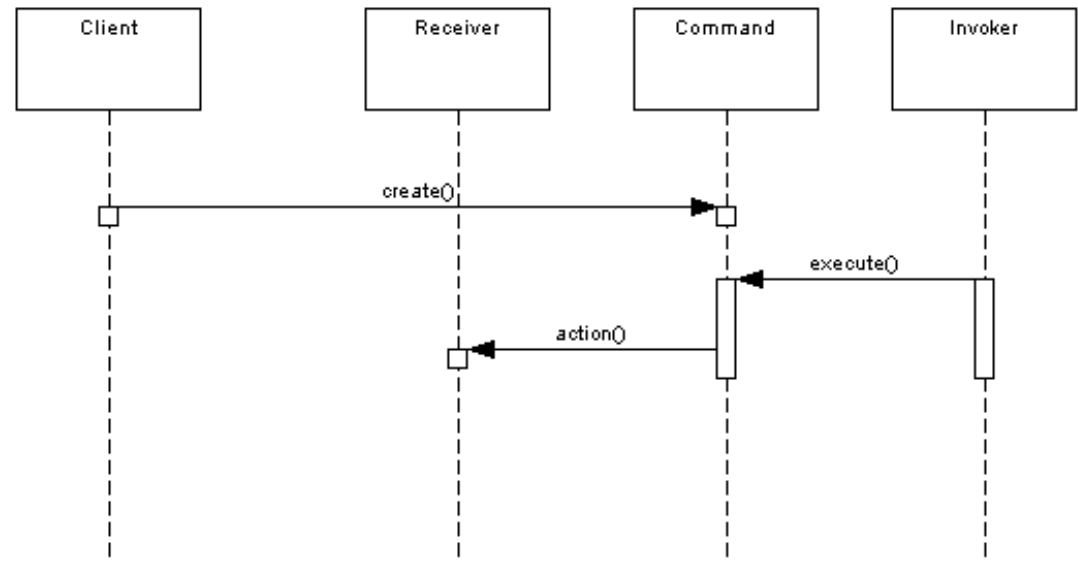
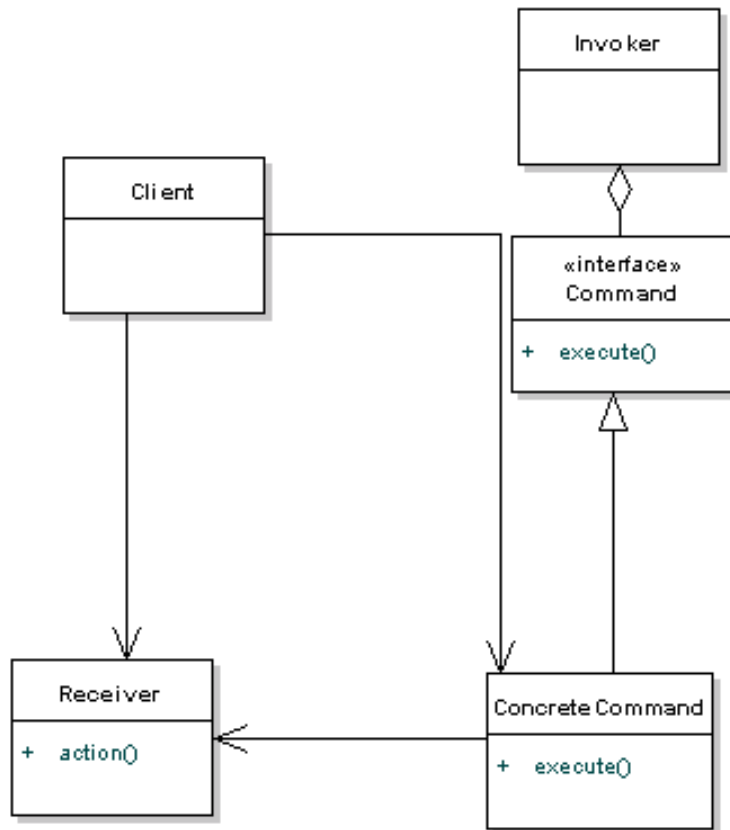
Transformando Acciones en Objetos

- ▶ 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

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



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
```

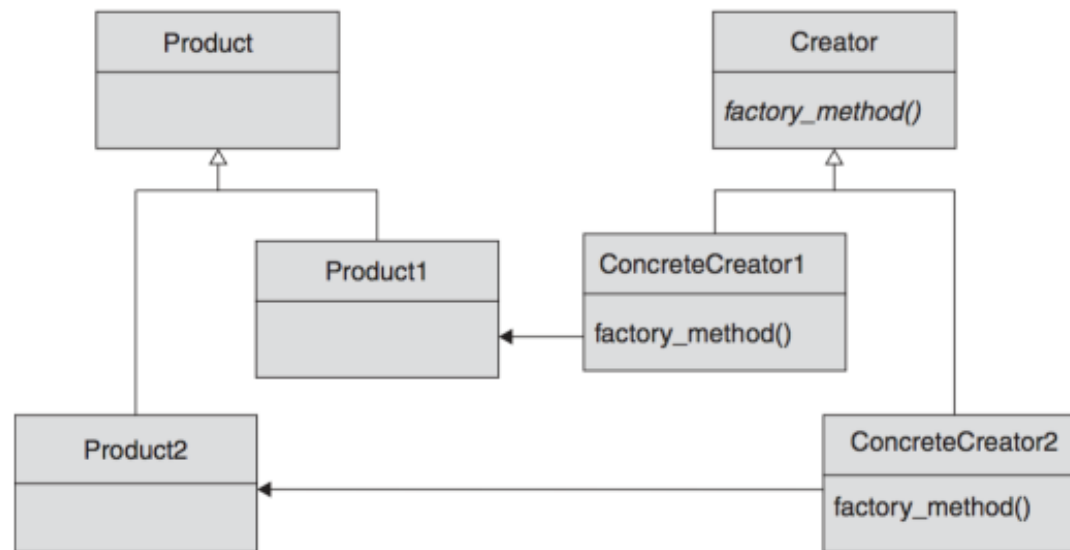
```
class DeleteFile < Command
  def initialize(path)
    super("Delete file: #{path}")
    @path = path
  end
  def execute
    File.delete(@path)
  end
end
```

```
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
```

```
acciones = [CreateFile.new("file1.txt", "contenido"),
             CreateFile.new("file2.txt", "otra cosa")]
acciones.each {|command| command.execute}
```

Factory Method

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



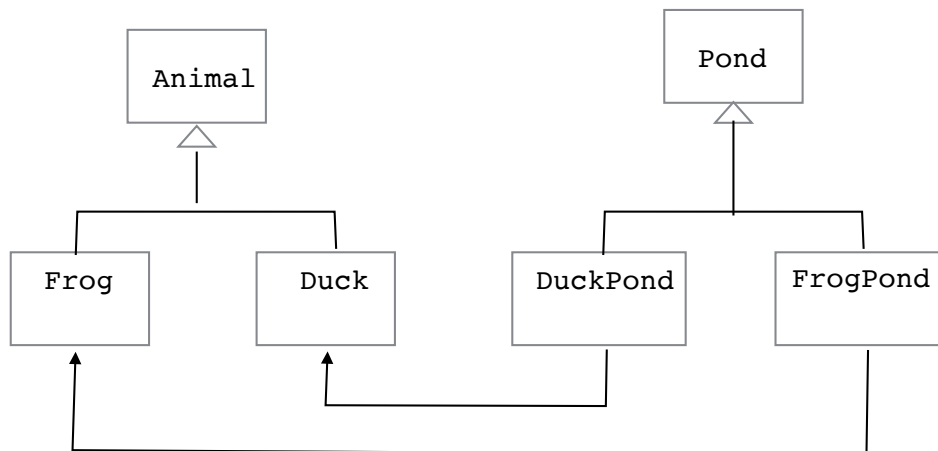
La laguna de patos

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

```
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
pond = DuckPond.new(2)
pond.simulate_one_day
```

De la laguna a un habitat

```
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...
```

```
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
```

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

La Fábrica Abstracta

- ▶ fábricas concretas, cada una capaz de crear su propio set de productos
- ▶ Es la misma idea del patrón Estrategy pero aplicado al problema de crear objetos

