

Patrones de Diseño

Ayudantía

Ing. de Software 2019-1

¿Por qué usar patrones de diseño?

Principalmente porque le otorgan a tu código:

- Legibilidad
- Limpieza
- Mutabilidad
- Agilidad
- Elegancia
- Todo lo que a uno no le enseñan en IIC2233.

How do I write Clean Code?

```
I will not write any more bad code  
I will not write any more bad code  
I will not write any more bad code  
I will not write any more bad code  
I will not write any more bad code  
I will not write any more bad code  
I will not write any more bad code  
I will not write any more bad code  
I will not write any more bad code  
I will not write any more bad code  
I will not write any more bad code  
I will not write any more bad code
```



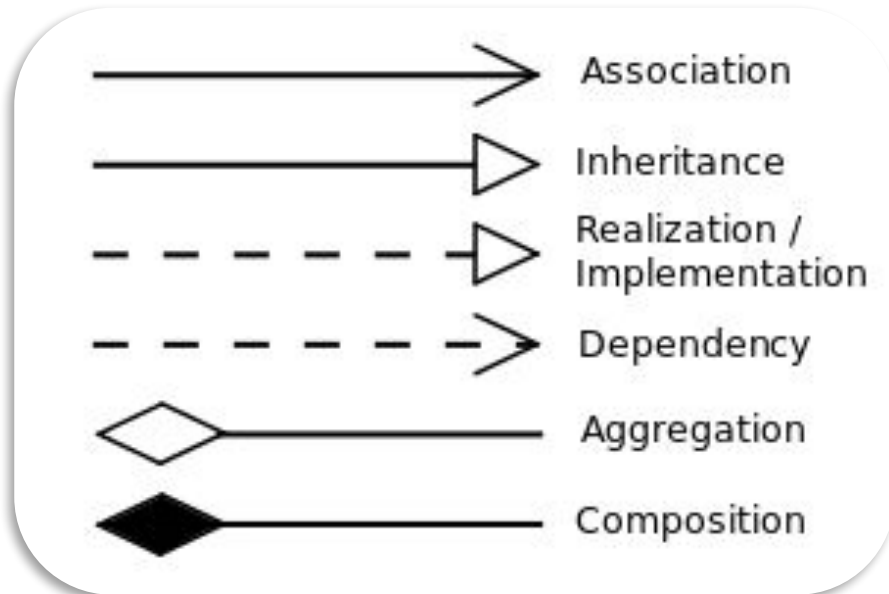
¿Cuántos patrones de diseño existen?

Muchos!

Pero en este curso se enseñan los 11 más comunes:

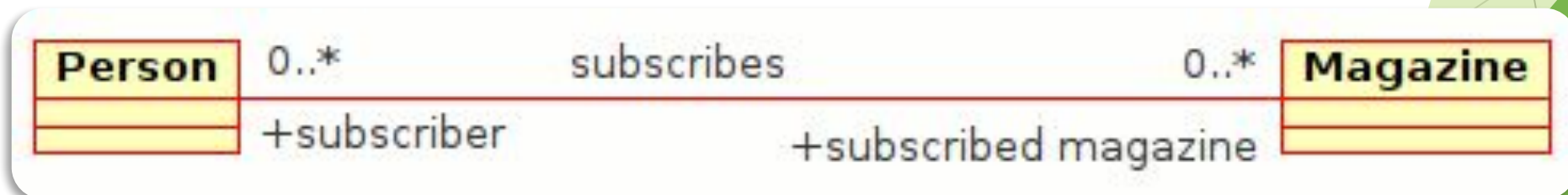
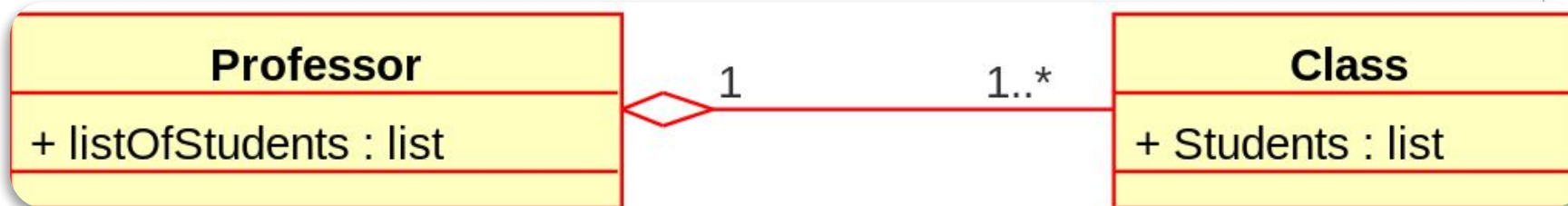
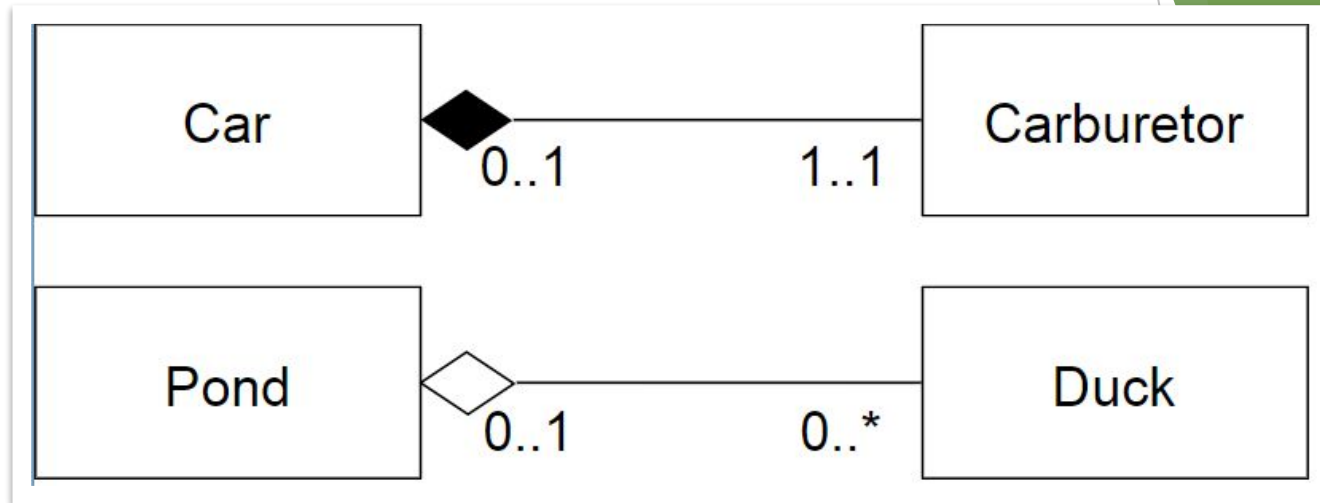
- Adaptador
- Fachada
- Singleton
- Observador
- Template Method
- Estrategy
- Decorador

Pequeño repaso de UML

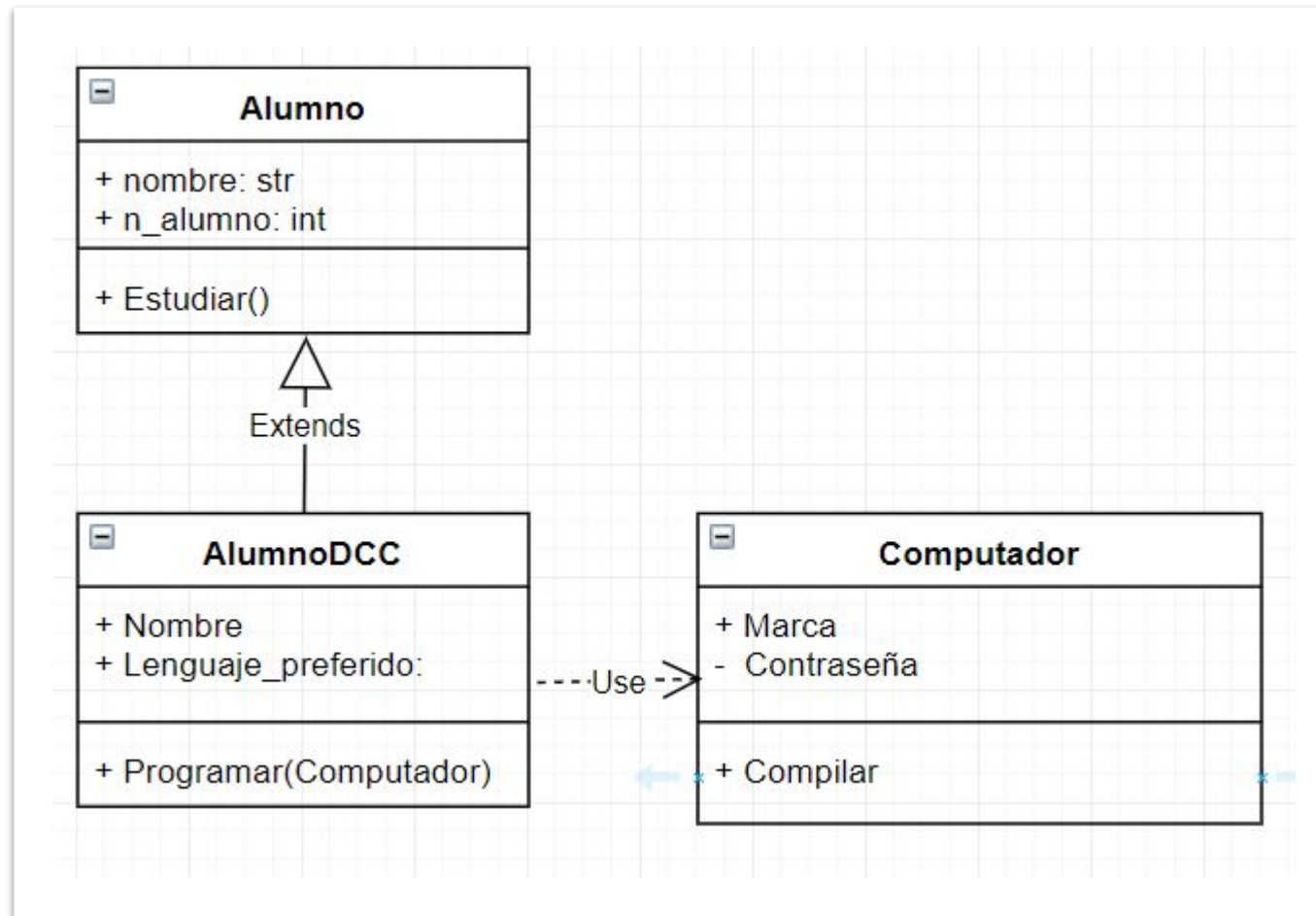


+	Public
-	Private
#	Protected
~	Package

Ejemplos

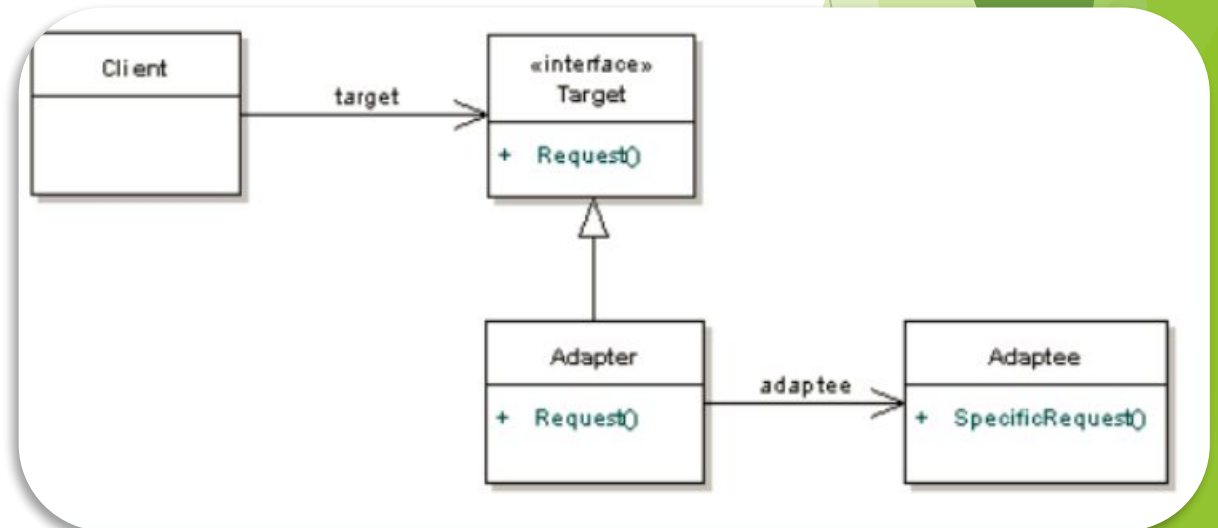
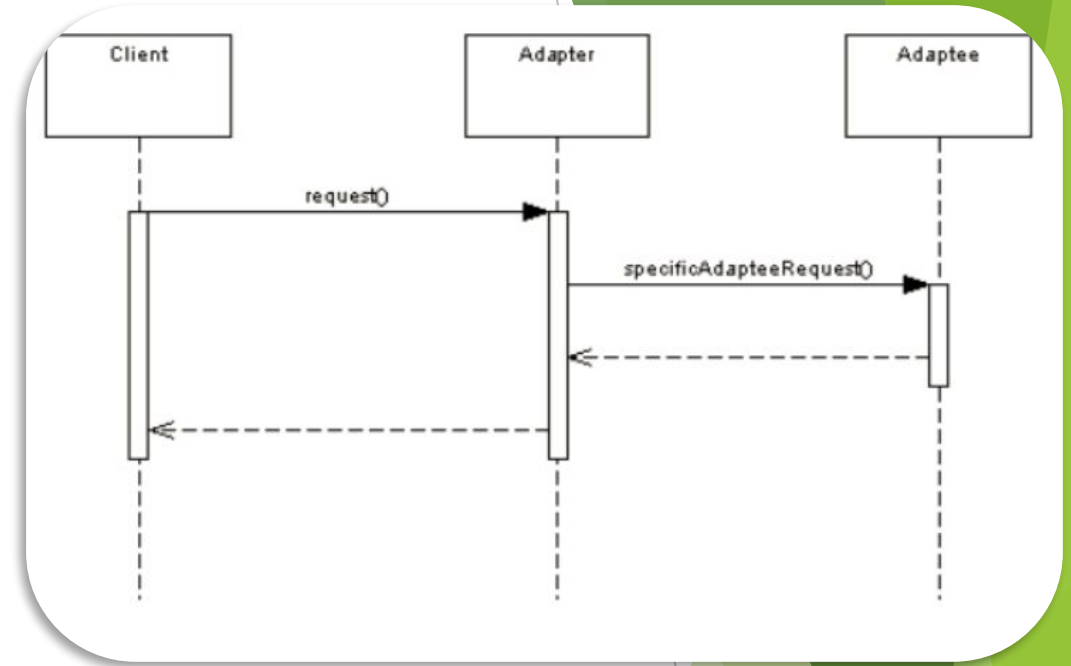


Ejemplos



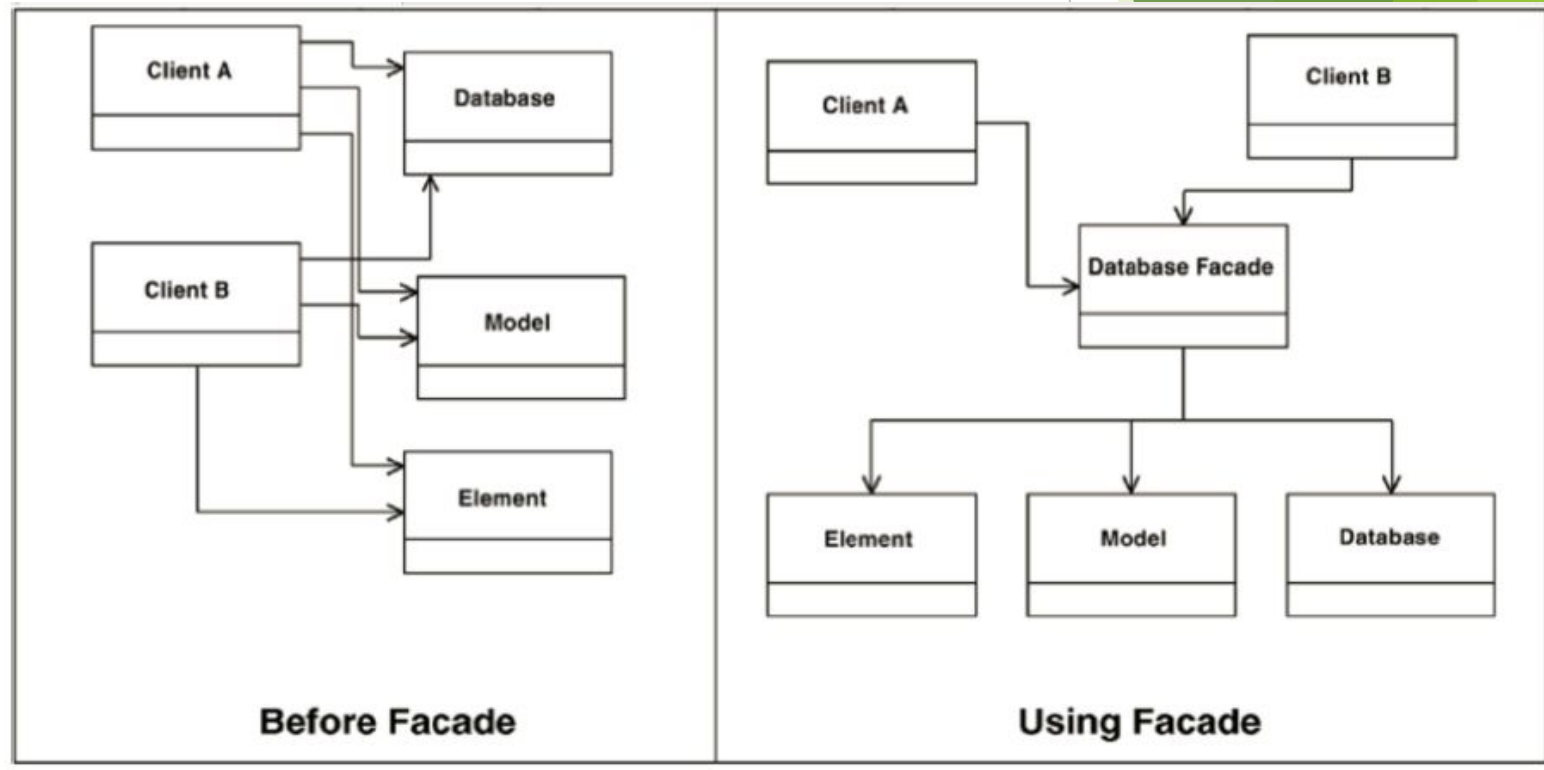
Adaptador

- Es un patrón **estructural**.
- La gracia de este patrón es que logra la interacción entre dos objetos incompatibles.
- Por ejemplo: Tengo un programa que funciona leyendo archivos JSON, pero mi base de datos está en XML.
- Solución: Programar un adaptador que rediriga el método leer_json() a leer_xml() y retorne el resultado al programa principal.

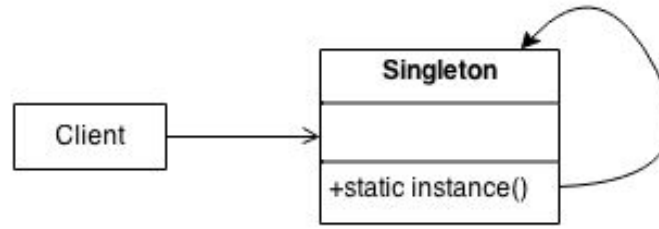


Fachada

- También es un patrón **estructural**.
- El objeto que sirve como fachada sirve como mediador de toda interacción entre el cliente y el backend.
- Por ejemplo: Ruby on Rails!



Singleton



- Es un patrón creacional.
- Este patrón restringe a que una clase solo pueda ser instanciada una vez. Es útil en sistemas complejos en los que se podría cometer el error de instanciar dos veces a un objeto que debiese ser único.
- Por ejemplo: El tablero en un juego de ajedrez, o un controlador de Logs.

```
class SimpleLogger

  # Lots of code deleted...

  @@instance = SimpleLogger.new

  def self.instance
    return @@instance
  end

  private_class_method :new

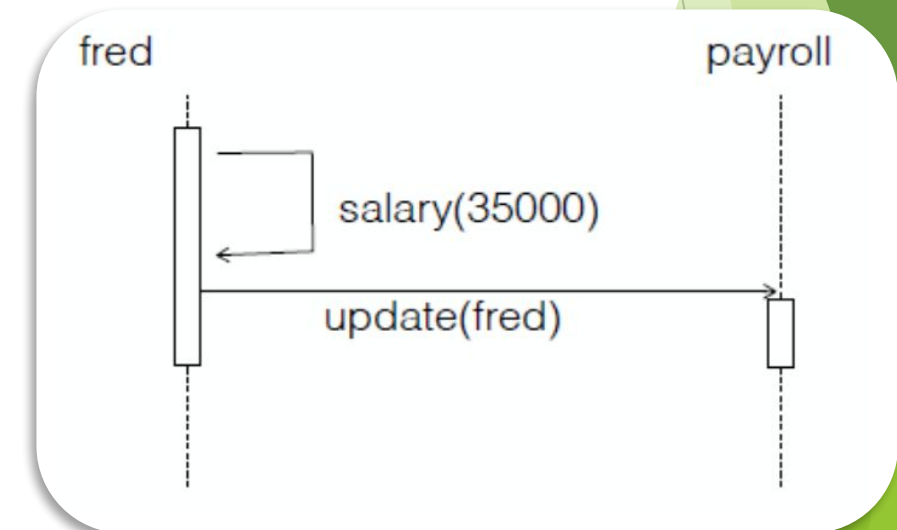
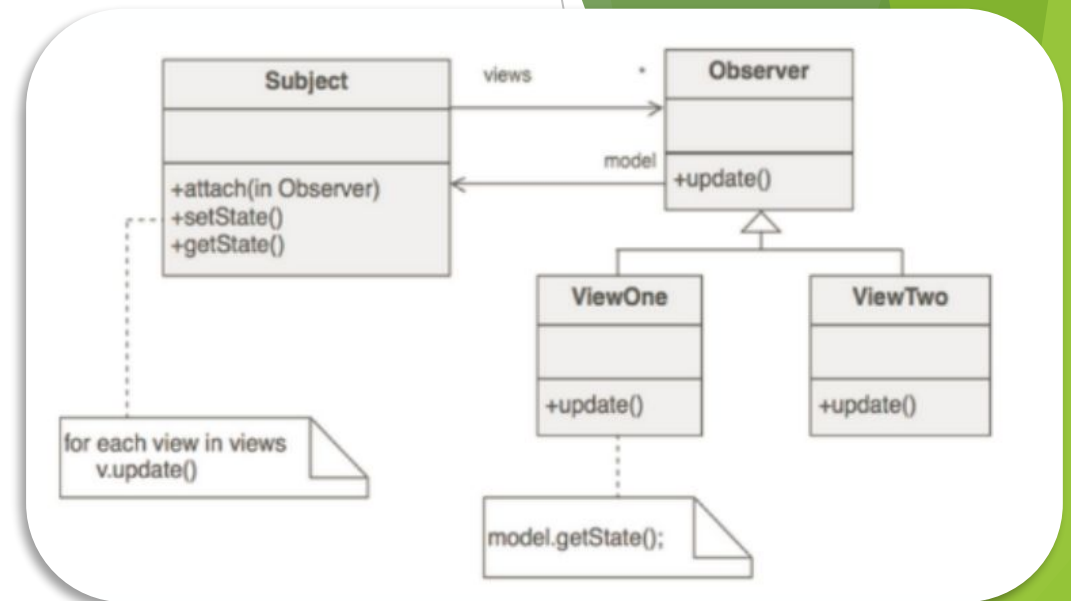
end
```

```
logger1 = SimpleLogger.instance # Returns the logger
logger2 = SimpleLogger.instance # Returns exactly the same logger

SimpleLogger.instance.info('Computer wins chess game.')
```

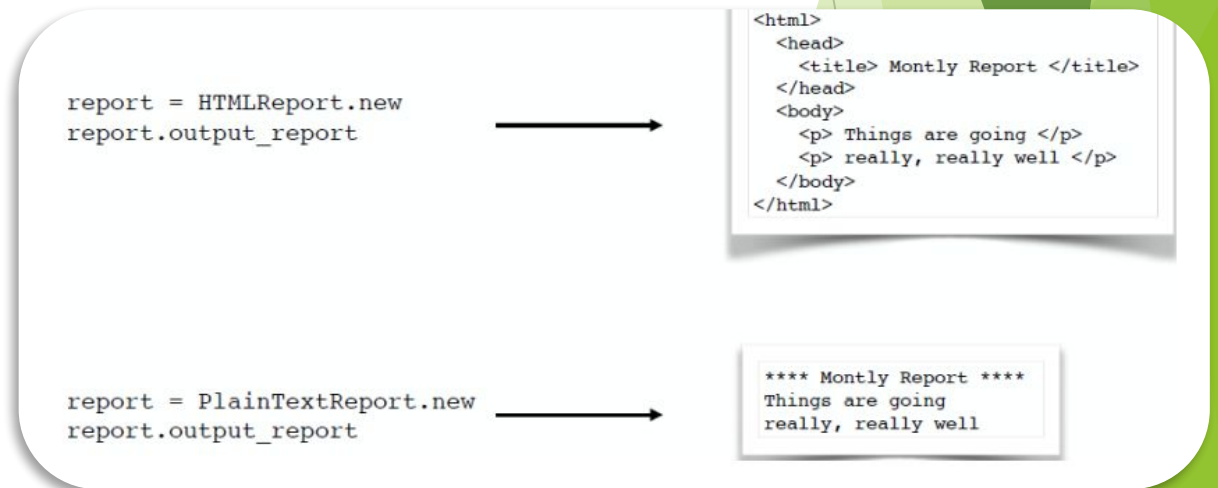
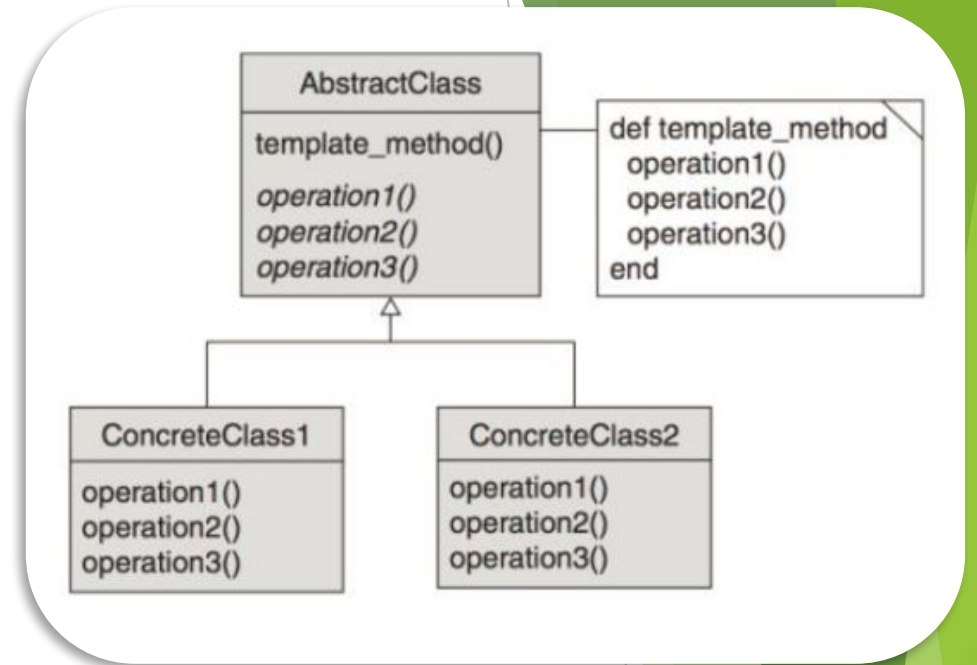
Observador

- Es un patrón de comportamiento.
- Muy útil en cuanto a sistemas de monitoreo y actualización de reportes.
- Por ejemplo: El historial de cobros de sueldo. En este caso el cliente, al cobrar un cheque, llama al método *update()* de la clase *payroll*, avisando sobre la acción que se acaba de ejecutar.
- El observador de cierta forma espera a que le lleguen notificaciones, en vez de estar verificando las cosas el mismo.



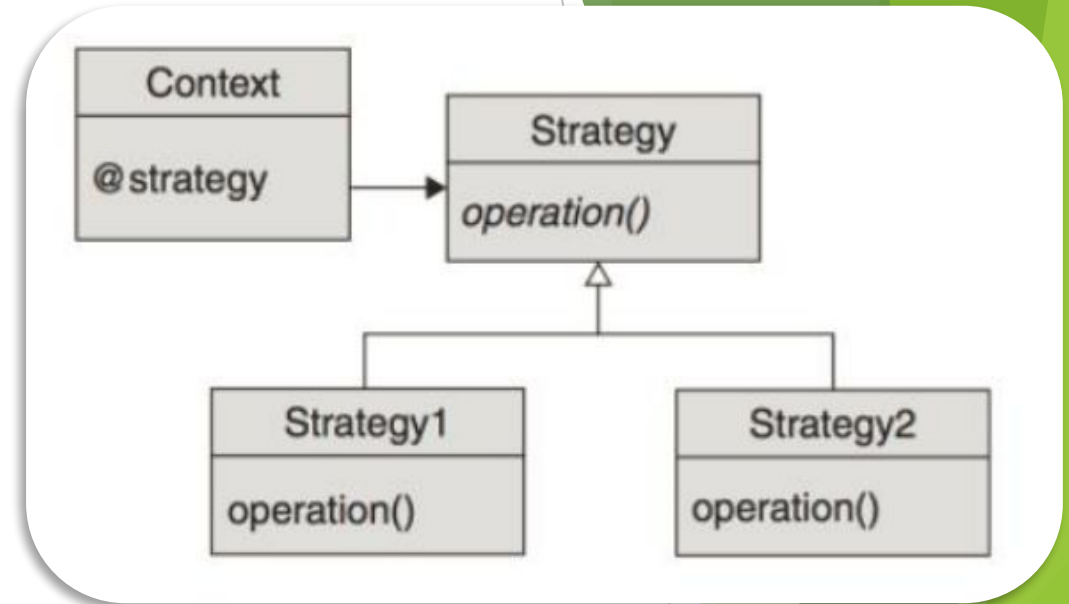
Template Method

- También es un patrón de comportamiento
- Se implementa cuando se quiere hacer lo mismo pero en distintos formatos.
- Por ejemplo: ReporteHTML y ReportePlainText, heredan de Reporte.



Estrategia

- Es un patrón de **comportamiento**.
- A diferencia de Template Method, acá existe la clase Strategy que posee el método implementado de diversas formas.
- Por ejemplo: Soy un estudiante que posee una calculadora. Al calcular `sin(t)`, no ejecuto `self.sin(t)` sino que ejecuto `self.calculadora.sin(t)`. Luego, este calculo tendrá diferentes resultados dependiendo de si la Calculadora es una DegreeCalculator o una RadianCalculator.
- El ejemplo de los templates también se puede hacer con este patrón.



```
class Report
  attr_reader :title, :text
  attr_accessor :formatter

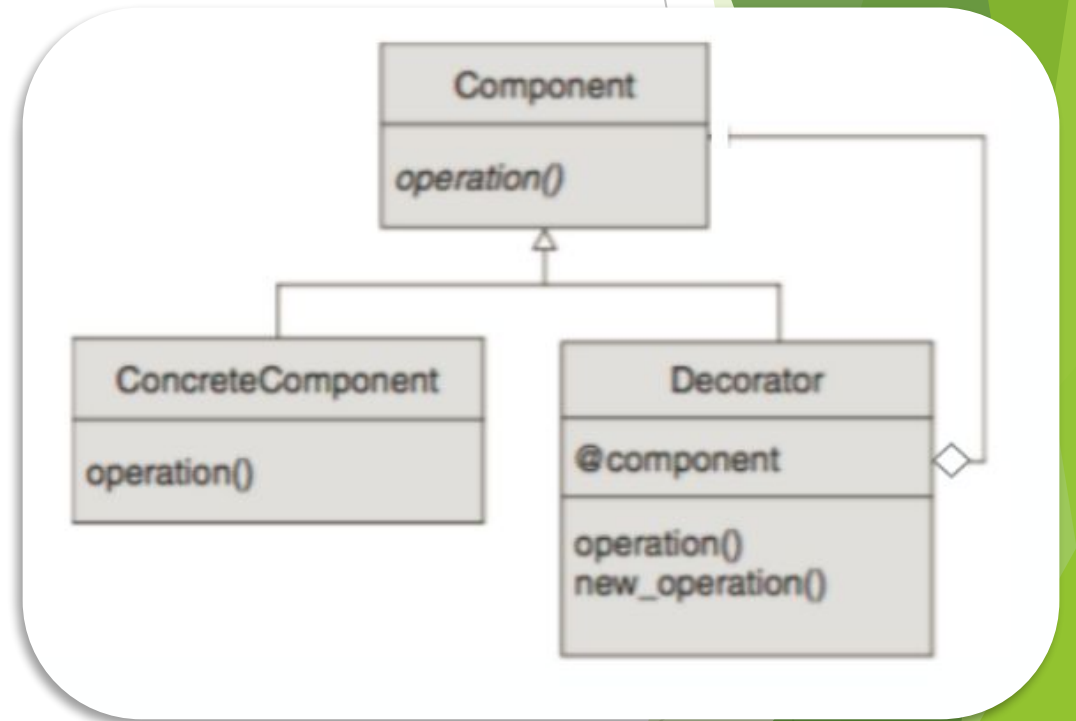
  def initialize(formatter)
    @title = 'Monthly Report'
    @text = [ 'Things are going', 'really, really well.' ]
    @formatter = formatter
  end

  def output_report
    @formatter.output_report( @title, @text )
  end
end

class HTMLFormatter
  def output_report( title, text )
    puts("<html>")
  end
end
```

Decorador

- Es un patrón **estructural**.
- Sirve para envolver (*wrapper*) a otras clases de manera anidada, sin editar lo que hay dentro de ellas.
- En general se trata de que el orden de anidación no influya en el resultado final.
- Por ejemplo: decoradores de Python.



```

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 CheckSummingWriter < WriterDecorator
  attr_reader :check_sum
  def initialize(real_writer)
    @real_writer = real_writer
    @check_sum = 0
  end
  def write_line(line)
    line.each_byte {|byte| @check_sum = (@check_sum + byte) % 256 }
    @check_sum += "\n"[0] % 256
    @real_writer.write_line(line)
  end
end

```

```

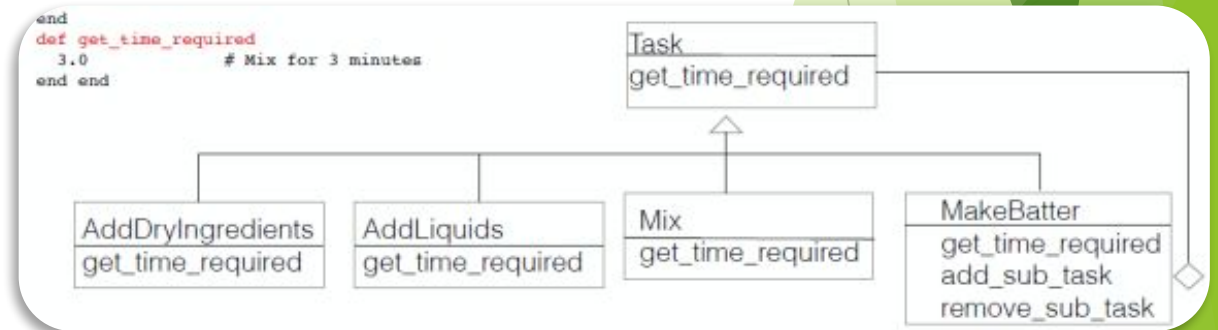
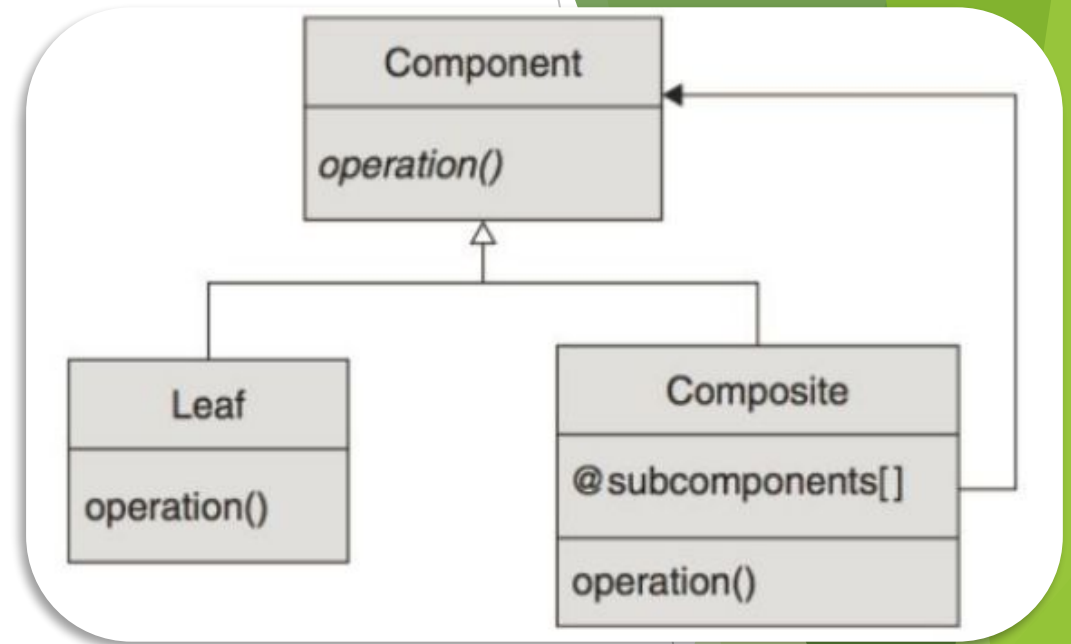
class TimeStampingWriter < WriterDecorator
  def write_line(line)
    @real_writer.write_line("#{Time.new}: #{line}")
  end
end

```

Decorador

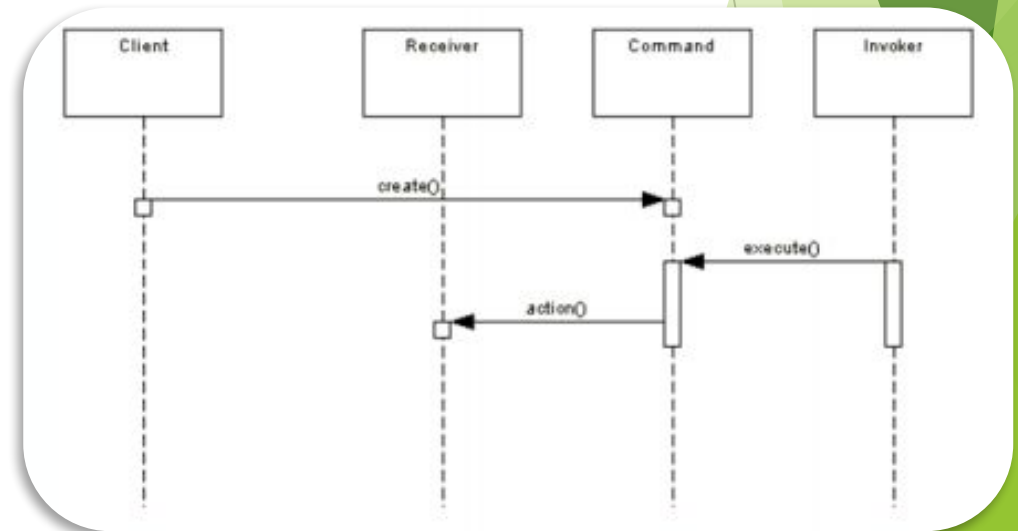
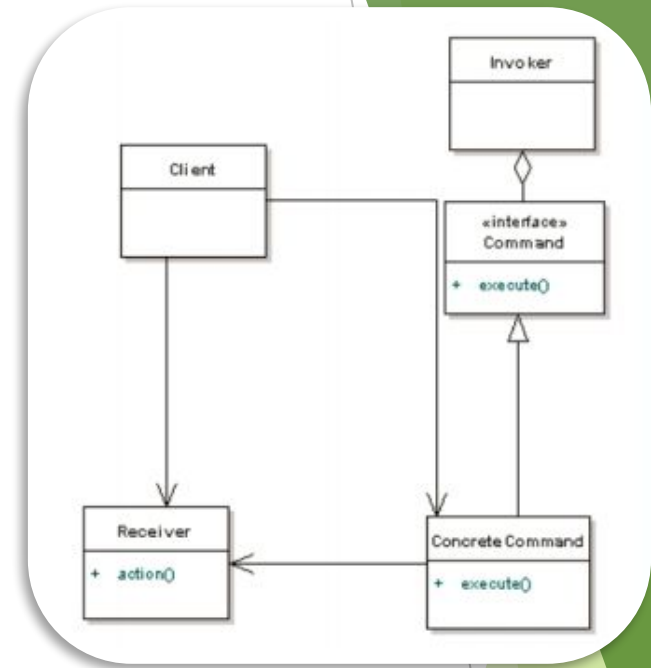
Composite

- Es un patrón **estructural**.
- Logra organizar objetos de manera jerárquica.
- Por ejemplo: Hacer una torta, armar un juguete, etc.
- Es posible crear métodos cuyo resultado sea acumulativo. Por ejemplo el tiempo que demora en cocinar la torta, que depende del tiempo de cada subtarea.



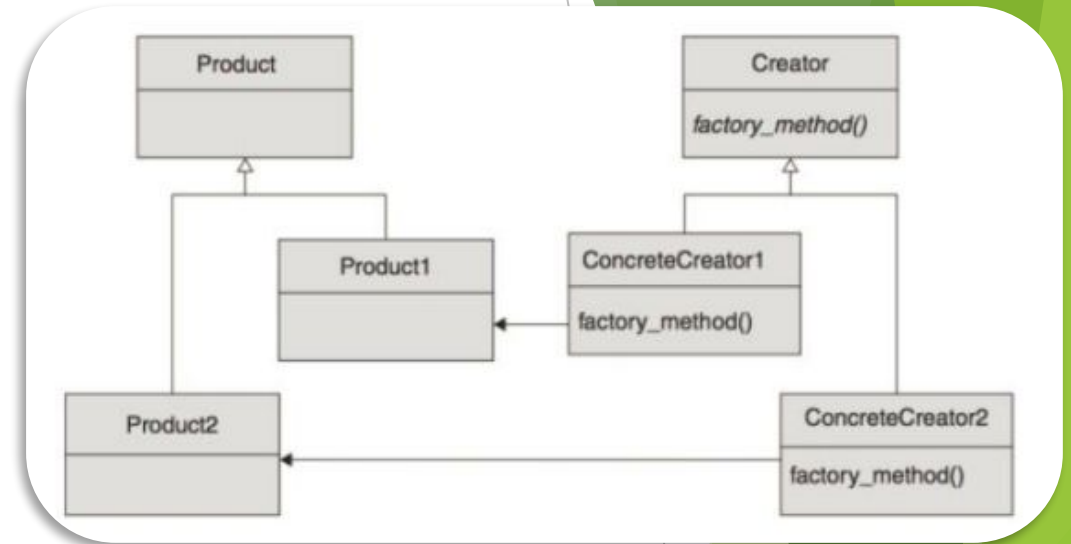
Comando

- Es un patrón de comportamiento.
- La idea es representar cada acción que pueda ejecutar el usuario como un objeto.
- Ejemplos de uso: Poder poner acciones en cola, poder implementar ctrl+Z, asignar atributos a las acciones, programar un scheduler, etc.



Factory Method

- Es un patrón creacional.
- Cada objeto es creado por otra clase creadora.
- Ventajas: La clase creadora puede controlar cuanto se crea, cuando se crea, llevar registros, etc. Además, no es necesario saber la clase del objeto si se manda a hacer a la clase Creador.
- Ojo: Una clase creadora solo crea una clase predeterminada de objetos.

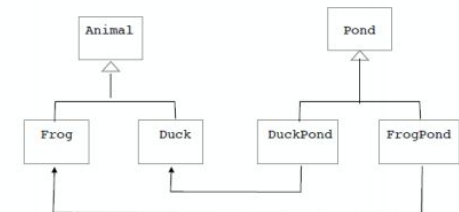


```
class Pond
def initialize(number_animals)
  @animals = []
  number_animals.times do |i|
    animal = new_animal("Animal#{i}")
    @animals << animal
  end
end
def simulate_one_day
  @animals.each {|animal| animal.speak}
  @animals.each {|animal| animal.eat}
  @animals.each {|animal| animal.sleep}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



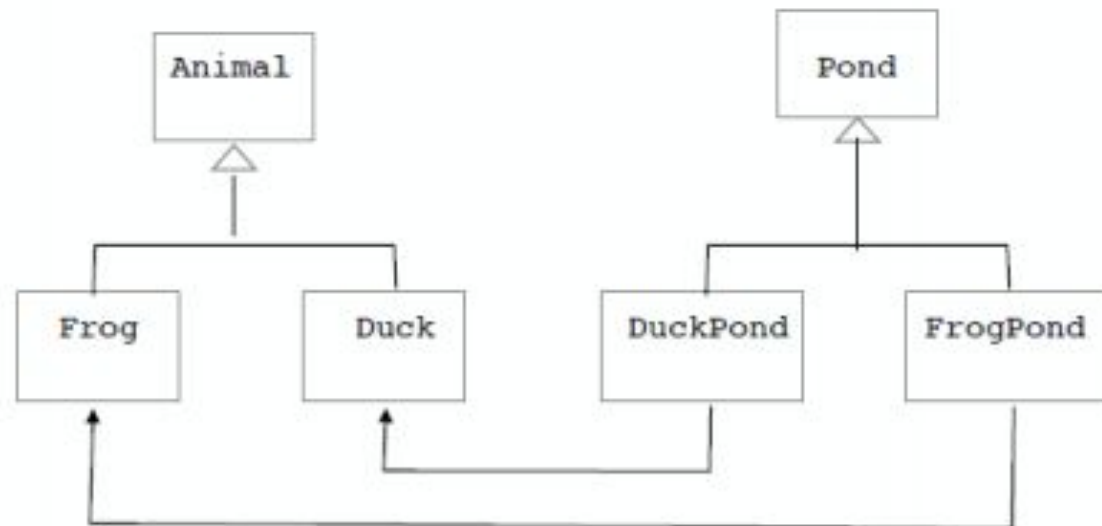
Factory Method

```
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
```

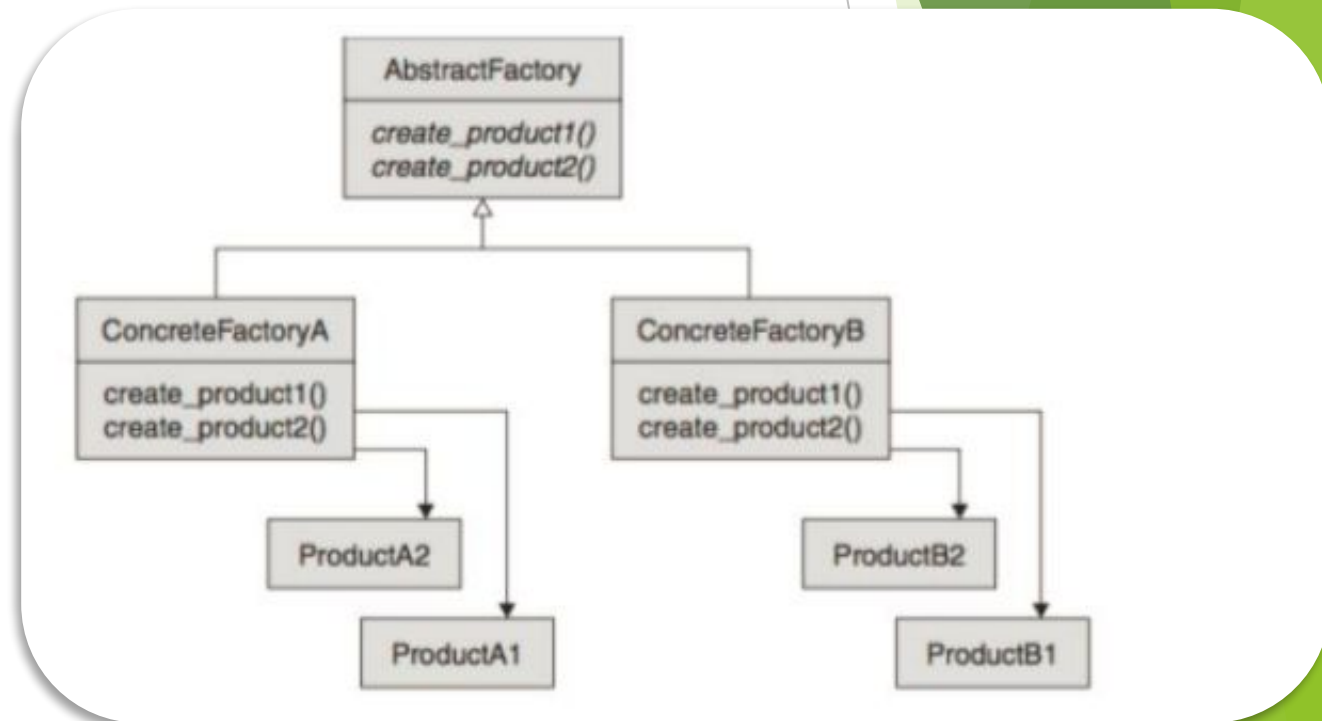
```
class FrogPond < Pond
  def new_animal(name)
    Frog.new(name)
  end
end
```

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



Abstract Factory

- Es un patrón creacional.
- Diferencia con la fábrica normal es que acá las clases creadoras pueden crear múltiples clases.
- Se ocupa cuando existen maneras de crear objetos de manera combinada, como el producto A1, A2, B1 y B2.



```

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...
end

```

```

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

```

Abstract Factory

Ejercicios

Pregunta 2 - 20 pts (Patrones de Diseño)

Se quiere implementar una clase Podcast que mantiene una lista de episodios y un método `add_episode` que permite incorporar nuevos episodios. Normalmente un Podcast mantiene un cierto número de suscriptores que deben ser informados cada vez que se agrega un nuevo episodio. Queremos que funcione mas o menos así:

```
>> developer_tea = Podcast.new
>> s1 = Subscriber.new("Jaime")
>> s2 = Subscriber.new("Rodrigo")
>> developer_tea.add_subscriber(s1)
>> developer_tea.add_subscriber(s2)
>> developer_tea.add_episode (Episode.new("T1-E3"))
```

```
Jaime: there is a new episode !
```

```
Rodrigo: there is a new episode !
```

```
>> developer_tea.remove_subscriber(s1)
>> developer_tea.add_episode (Episode.new("T1-E4"))
```

```
Rodrigo: there is a new episode !
```













Pregunta 2 - 20 pts (Patrones de Diseño)

Se quiere implementar una clase Podcast que mantiene una lista de episodios y un método `add_episode` que permite incorporar nuevos episodios. Normalmente un Podcast mantiene un cierto número de suscriptores que deben ser informados cada vez que se agrega un nuevo episodio. Queremos que funcione mas o menos así:

- a) (15 pts) Identifique el patrón de diseño relevante y escriba el código Ruby necesario (Solo lo estrictamente necesario para que funcione como el ejemplo). Debe escribir
- La clase Podcast con los métodos necesarios
 - La clase Subscriber con los métodos necesarios
 - La clase Episode

4. (0.7 pts) Relacione los nombres de los siguientes patrones *GoF* con su descripción correspondiente (utilice la letra que lista cada patrón):

- a. Decorator
- b. Strategy
- c. Factory Method
- d. Adapter
- e. Observer
- f. Composite
- g. Command

-  Encapsula un objeto para exponer una nueva interfaz de este
-  Permite crear objetos de distintas familias sin especificar sus clases
-  Clientes interactúan con colecciones de objetos y objetos individuales uniformemente
-  Permite notificar a otros objetos cuando el estado de uno cambia
-  Encapsula comportamiento intercambiable dinámicamente
-  Simplifica la interfaz de un grupo de interfaces
-  Encapsula una petición en un objeto
-  Añade funcionalidad a un objeto dinámicamente
-  Subclases deciden qué clase en concreto se crea
-  Encapsula un objeto para controlar su acceso
-  Garantiza que una clase tenga solamente una instancia

Pregunta 1 - 20 pts (Patrones de Diseño)

Un software de gestión bastante complejo incluye una componente destinada a generar reportes impresos. El problema es que el software debe manejar distintos formatos de reporte dependiendo si este está destinado a Chile, USA, Brasil, o alguno de otros 10 países donde se utiliza. Cada reporte incluye títulos de nivel 1, 2 y 3, párrafos, tablas y gráficos. Cada tipo de reporte incluye un set particular de dichos elementos.

- a) (5 pts) ¿Cual es el patrón de diseño más apropiado para este problema ? Fundamente su respuesta
- b) (10 pts) Haga un diagrama de clases UML que ilustre su solución para solo 2 tipos de reporte (Chile y USA) y 3 tipos de elementos (párrafos, tablas y gráficos). Escriba el código Ruby de las distintas clases del diagrama (solo lo mínimo necesario)
- c) (5 pts) Escriba un trozo de código Ruby que permita demostrar la forma de usar lo anterior para generar los reportes de Chile y USA mencionados

Respuestas

```
class Podcast
  attr_reader :episodes, :subscribers
  def initialize()
    @episodes = []
    @subscribers = []
  end
  def add_subscriber(new_subscriber)
    subscribers << new_subscriber
  end
  def remove_subscriber(leaving_subscriber)
    subscribers.delete(leaving_subscriber)
  end
  def add_episode(new_episode)
    episodes << new_episode
    subscribers.each do |subscriber|
      subscriber.update(self)
    end
  end
end
```

```
class Subscriber
  attr_reader :name
  def initialize(name)
    @name = name
  end
  def update (podcast)
    puts "#{name}: there is a new episode ! "
  end
end
```

Finalmente la clase Episode no requiere nada especial

```
class Episode
  attr_reader :name
  def initialize(name)
    @name = name
  end
end
```

- a. Decorator
- b. Strategy
- c. Factory Method
- d. Adapter
- e. Observer
- f. Composite
- g. Command

d. Encapsula un objeto para exponer una nueva interfaz de este
__ Permite crear objetos de distintas familias sin especificar sus
clases

f. Clientes interactúan con colecciones de objetos y objetos
individuales uniformemente

e. Permite notificar a otros objetos cuando el estado de uno
cambia

b. Encapsula comportamiento intercambiable dinámicamente

__ Simplifica la interfaz de un grupo de interfaces

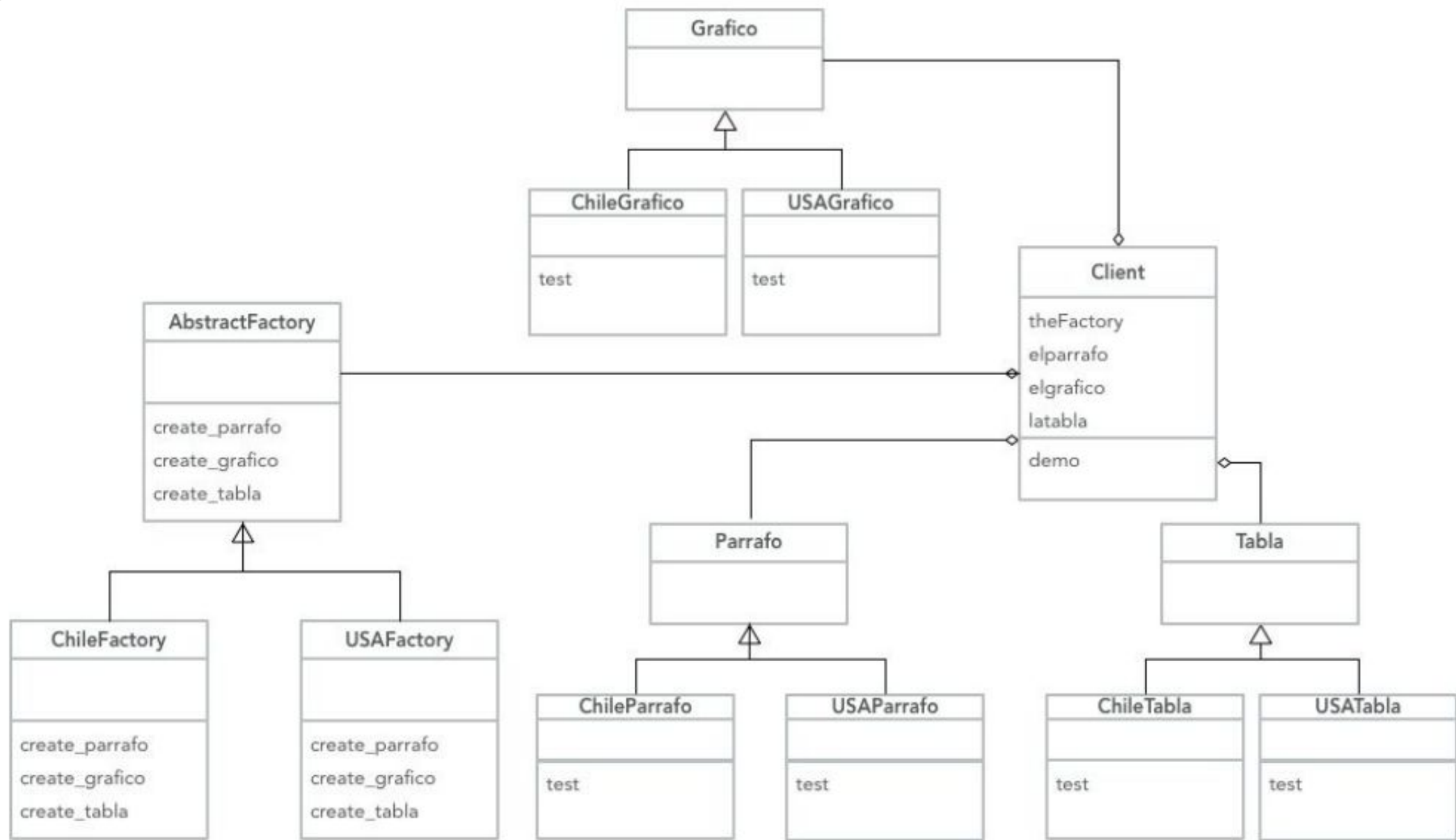
g. Encapsula una petición en un objeto

a. Añade funcionalidad a un objeto dinámicamente

c. Subclases deciden qué clase en concreto se crea

__ Encapsula un objeto para controlar su acceso

__ Garantiza que una clase tenga solamente una instancia




```
class AbstractFactory
  def create_parrafo
    puts "You should implement this method in the concrete factory"
  end
  def create_grafico
    puts "You should implement this method in the concrete factory"
  end
  def create_tabla
    puts "You should implement this method in the concrete factory"
  end
end
```

```
class ChileFactory < AbstractFactory
  def create_parrafo
    ChileParrafo.new
  end
  def create_grafico
    ChileGrafico.new
  end
  def create_tabla
    ChileTabla.new
  end
end
```

```
class USAFactory < AbstractFactory
  def create_parrafo
    USAParrafo.new
  end
  def create_grafico
    USAGrafico.new
  end
  def create_tabla
    USATabla.new
  end
end
```

```
Class ChileParrafo < Parrafo
  ...
  def test
    puts "parrafo chileno"
  end
Class ChileGrafico < Grafico
  ...
  def test
    puts "grafico chileno"
  end

...
end
Class ChileTabla < Tabla
...
  def test
    puts "tabla chilena"
  end

end
```

```
Class USAParrafo < Parrafo
...
  def test
    puts "parrafo gringo"
  end

end
```

```
Class USAParrafo < Parrafo
...
  def test
    puts "parrafo gringo"
  end

end
```

```
Class USAGrafico < Grafico
...
  def test
    puts "grafico gringo"
  end

end
Class USATabla < Tabla
...
  def test
    puts "tabla gringa"
  end

end
```

```
class Client
  attr_accessor :thefactory
  def initialize(afactory)
    @theFactory = aFactory
    @elparrafo = @theFactory.create_parrafo
    @elgrafico = @theFactory.create_grafico
    @latabla = @theFactory.create_tabla
  end
  def demo
    @elparrafo.test
    @elgrafico.test
    @latabla.test
  end
end
```

```
cliente1 = Client.new(ChileFactory.new)
cliente1.demo
```

```
parrafo chileno
grafico chileno
tabla chilena
```

```
cliente2 = Client.new(USAFactory.new)
cliente2.demo
```

```
parrafo gringo
grafico gringo
tabla gringa
```


Patrones de Diseño

Ayudantía

Ing. de Software 2019-1