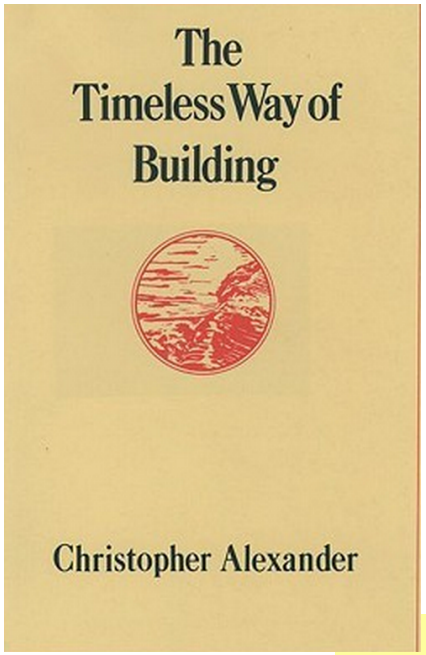
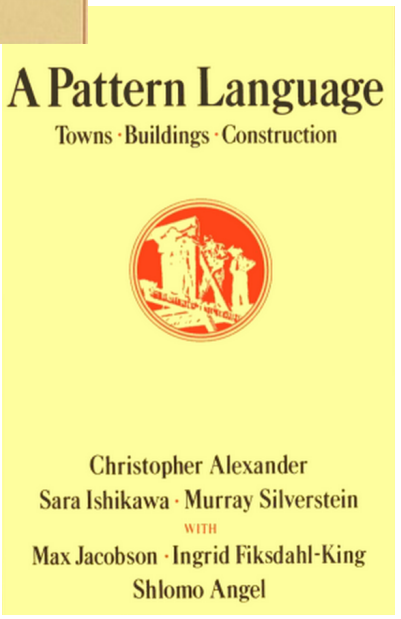
Patrones de Diseño

- Arquitecto Christopher Alexander
  - ¿Se puede hablar de calidad arquitectónica en términos objetivos ?
  - ¿Cómo podría uno saber que un diseñó particular es bueno ?
  - Alexander postula que hay una base objetiva
  - Una buena guía es observar las cosas que se repiten en edificios, calles y ciudades
  - define la idea de patrón como solución a un problema en un contexto





En palabras de Alexander

Drawings help people to work out intricate relationships between parts.

Christopher Alexander

Speaking as a builder, if you start something, you must have a vision of the thing which arises from your instinct about preserving and enhancing what is there.

Christopher Alexander

When they have a choice, people will always gravitate to those rooms which have light on two sides, and leave the rooms which are lit only from one side unused and empty.

Christopher Alexander

From a sequence of these individual patterns, whole buildings with the character of nature will form themselves within your thoughts, as easily as sentences.

Christopher Alexander

Everyone is aware that most of the built environment today lacks a natural order, an order which presents itself very strongly in places that were built centuries ago

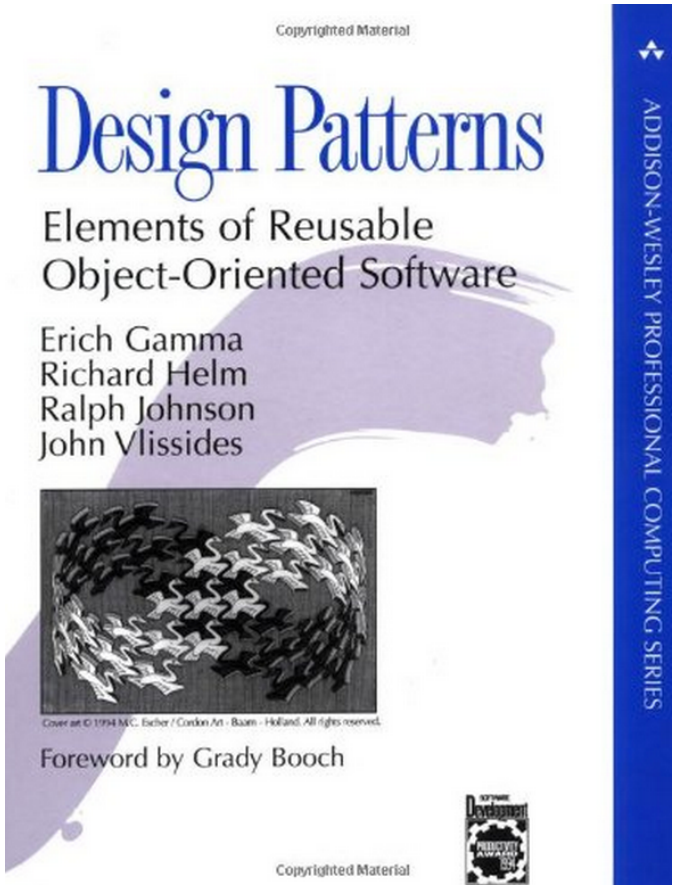
Christopher Alexander

Descripción de un Patrón

- Nombre
- Propósito
- Cómo logra el propósito
- Restricciones y fuerzas involucradas

De la Arquitectura al Software

- Gang of Four (Gamma, Helm, Johnson, Vlissides) escriben un libro que resulta clave
- The Elements of Reusable Object-Oriented Software
- Llevó la idea de Alexander al Software
- Propone forma estándar de describir un patrón
- Identifica y cataloga 23 patrones

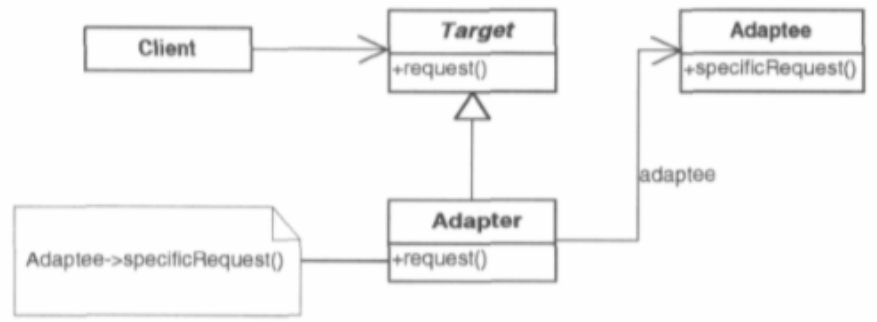


- 500.000 copias solo en Inglés
- traducido a 13 lenguajes

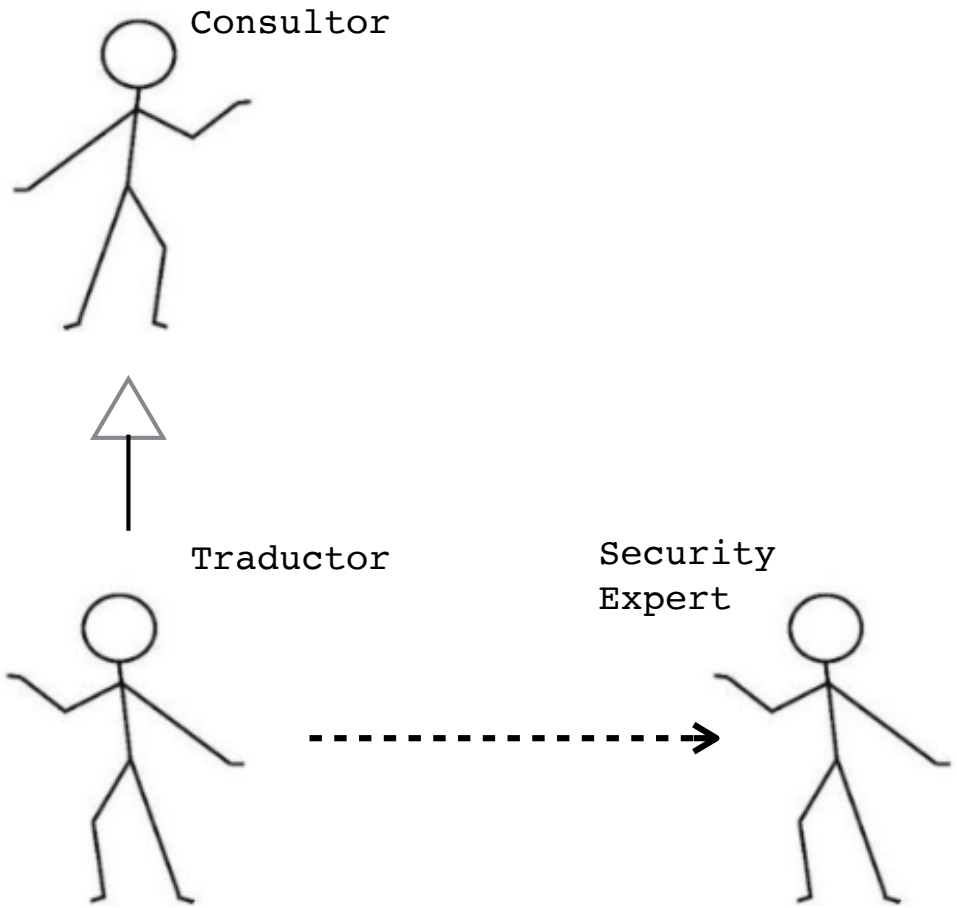
Ejemplo

The Adapter Pattern: Key Features

Intent	Match an existing object beyond your control to a particular interface.
Problem	A system has the right data and behavior but the wrong interface. Typically used when you have to make something a derivative of an abstract class we are defining or already have.
Solution	The Adapter provides a wrapper with the desired interface.
Participants and Collaborators	The Adapter adapts the interface of an Adaptee to match that of the Adapter's Target (the class it derives from). This allows the Client to use the Adaptee as if it were a type of Target.
Consequences	The Adapter pattern allows for preexisting objects to fit into new class structures without being limited by their interfaces.
Implementation	Contain the existing class in another class. Have the containing class match the required interface and call the methods of the contained class.



No solo en Software




Kensington


plugable
USB 2.0


WISMEC


KIWI
LMA-TM_EOS
AP306

¿ Por qué estudiarlos ?

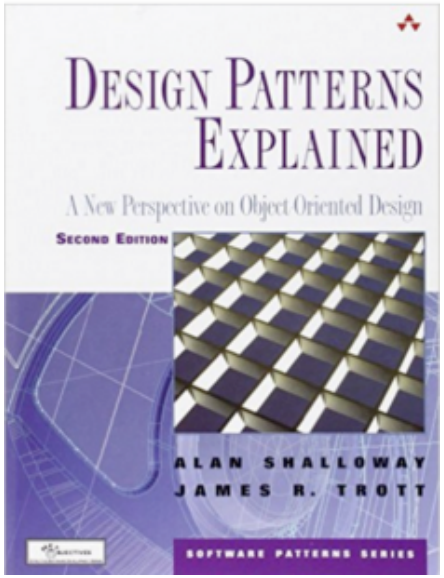
- Permiten reusar soluciones (experiencia de otros)
- Permiten establecer terminología común
- Nos dan una perspectiva de mas alto nivel para el diseño de software
- Mejora comunicación al interior del equipo
- Código es más fácil de modificar
- Permiten aprender estrategias generales de diseño

Algunas Estrategias en GoF

- Diseñe para interfaces
- Favorezca agregación por sobre herencia
- Identifique lo que cambia más y encapsúlelo

Libros


Design Patterns
Elements of Reusable
Object-Oriented Software
Erich Gamma
Richard Helm
Ralph Johnson
John Vlissides
Foreword by Grady Booch
ADDISON-WESLEY PROFESSIONAL COMPUTING SERIES

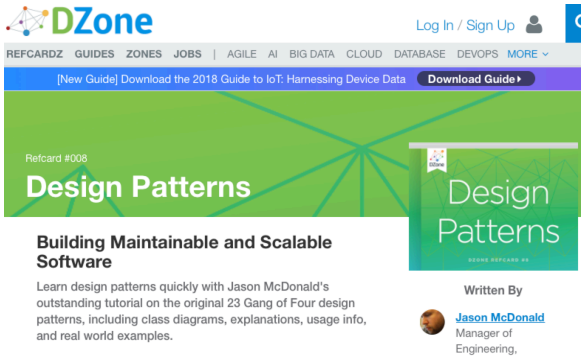

DESIGN PATTERNS
EXPLAINED
A New Perspective on Object Oriented Design
SECOND EDITION
ALAN SHALLOWAY
JAMES R. TROTT
SOFTWARE PATTERNS SERIES


O'REILLY
10th Anniversary
Updated for Java 8
Head First
Design Patterns
A Brain-Friendly Guide
Avoid those embarrassing coupling mistakes
Learn why everything your friends know about Factory pattern is probably wrong
Discover the secrets of the Patterns Guru
Load the patterns that matter straight into your brain
Find out how Starbuzz Coffee doubled their stock price with the Decorator pattern
See why Jim's love life improved when he cut down his inheritance
Eric Freeman & Elisabeth Robson
with Kathy Sierra & Bert Bates


Addison-Wesley Professional Ruby Series
DESIGN
PATTERNS
IN RUBY
Foreword by Obie Fernandez
Professional Ruby Series Editor
RUSS OLSEN

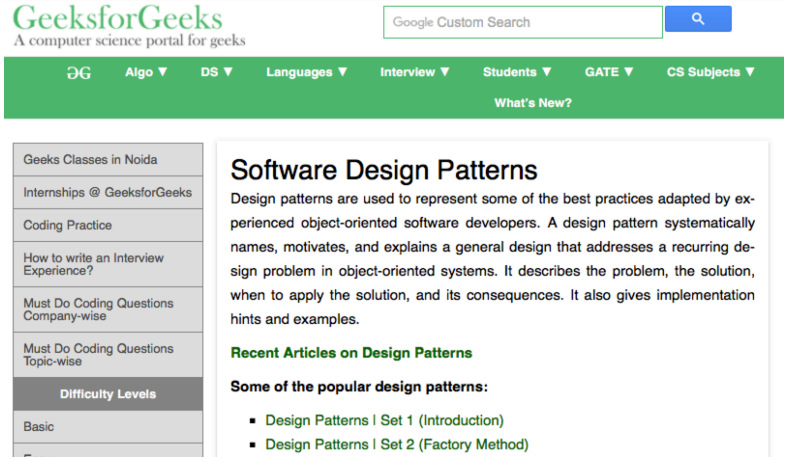
OODesign.com

http://www.oodesign.com



DZone.com

https://dzone.com/refcardz/design-patterns

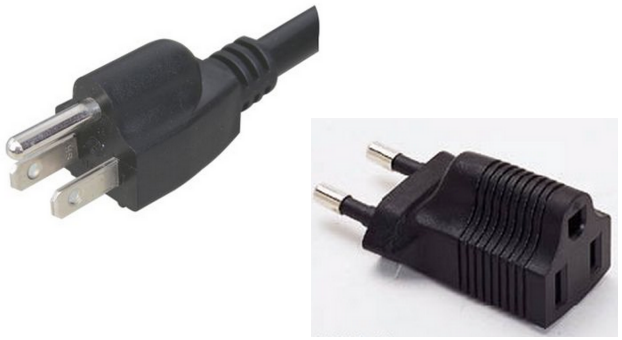


Geeks for Geeks

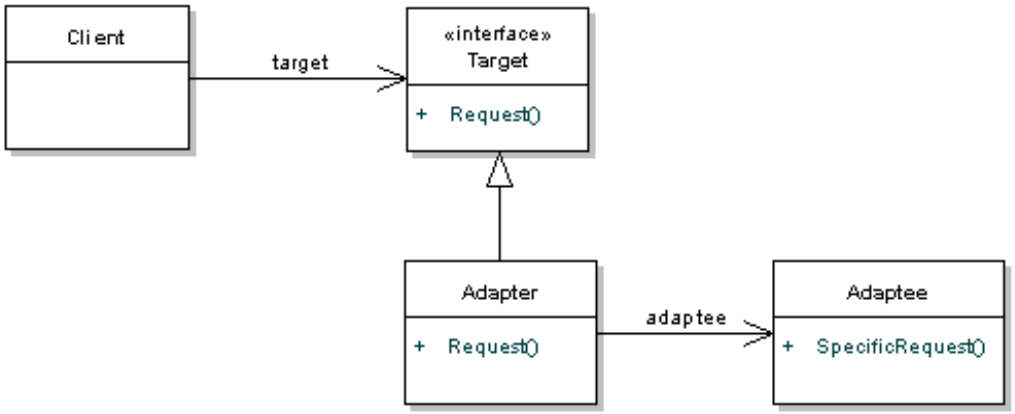
http://www.geeksforgeeks.org/software-design-patterns/

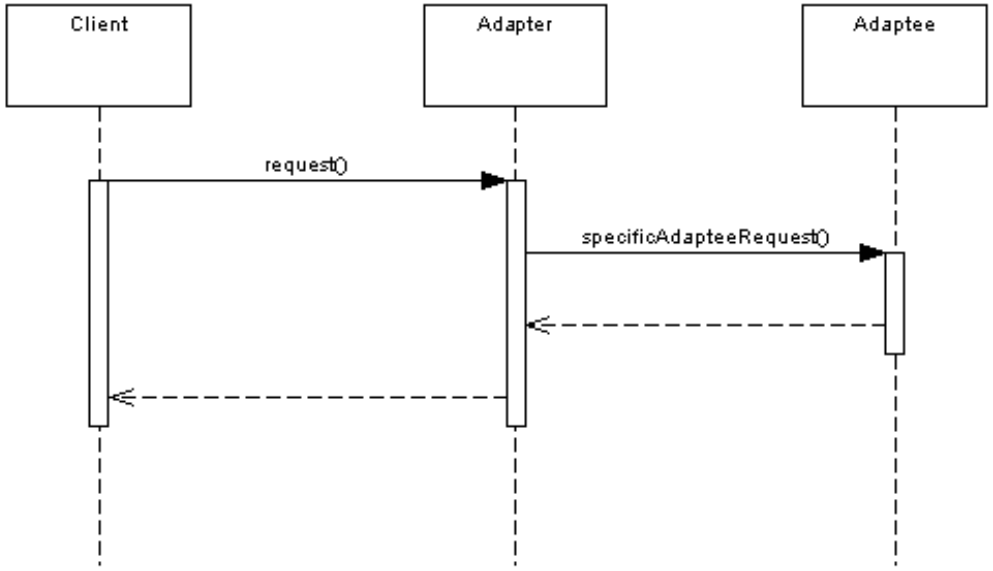
Adaptador

- Intención - Hacer calzar un objeto que no controlamos a nuestra interfaz
- Problema - Sistema tiene datos y comportamiento correctos pero interfaz incorrecta
- Solución - Adaptador provee un envoltorio con interfaz deseada
- Implementación - incorporar la clase existente en otra que tiene la interfaz deseada y hacer que los métodos de adaptador llamen a métodos del adaptado



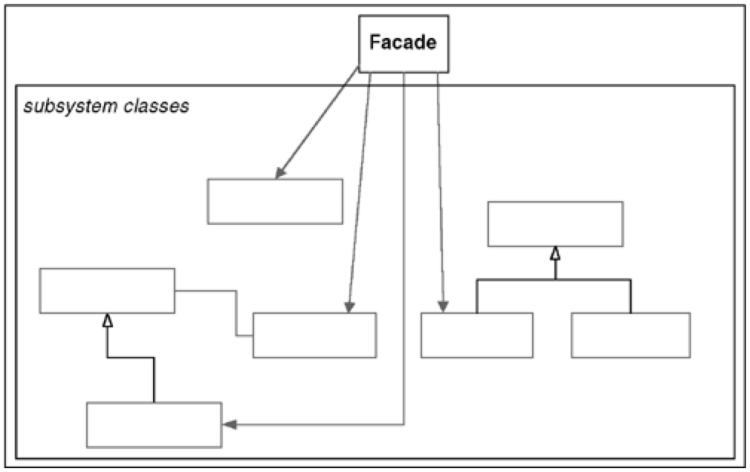
Adaptador en UML

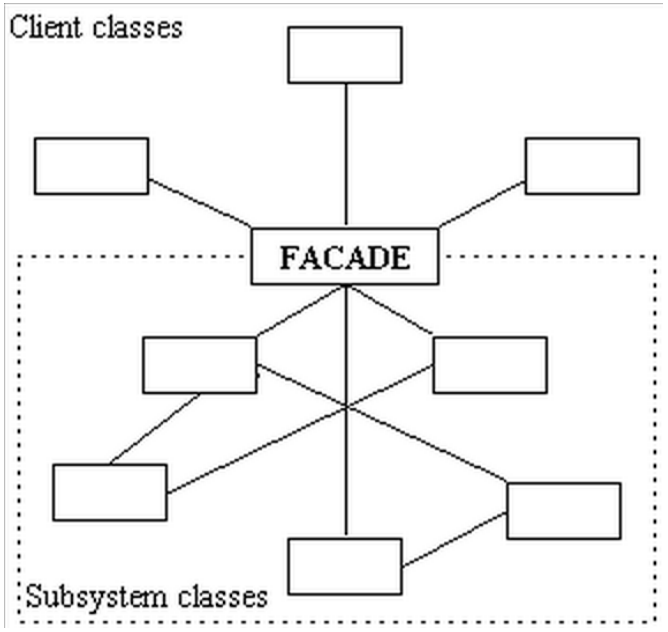




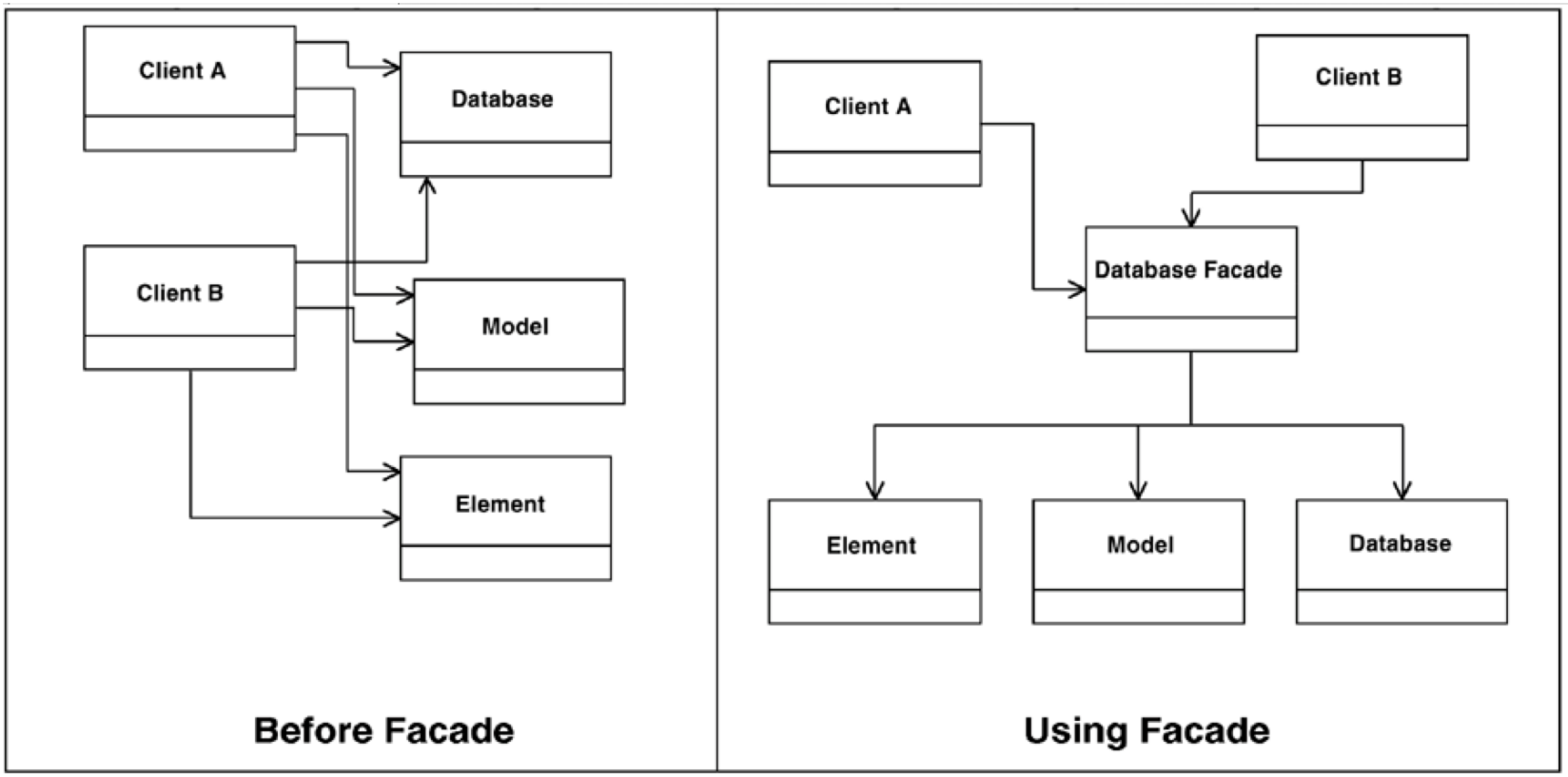
Fachada

- Intención - simplificar el uso de un sistema existente definiendo una fachada
- Problema - solo se requiere usar parte del sistema o interactuar con el en forma limitada
- Solución - la fachada presenta una nueva interfaz simplificada para usar el sistema
- Consecuencias - la fachada no es completa, hay funcionalidades que no estarán disponibles
- Implementación - definir una nueva clase (o clases) con la interfaz requerida. Esta clase usa el sistema existente





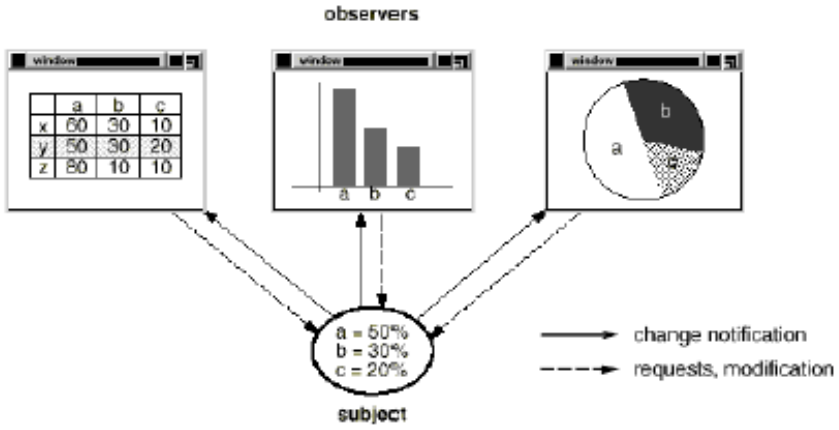
Uso de Fachada para Simplificar uso de un set de Clases a Clase Cliente



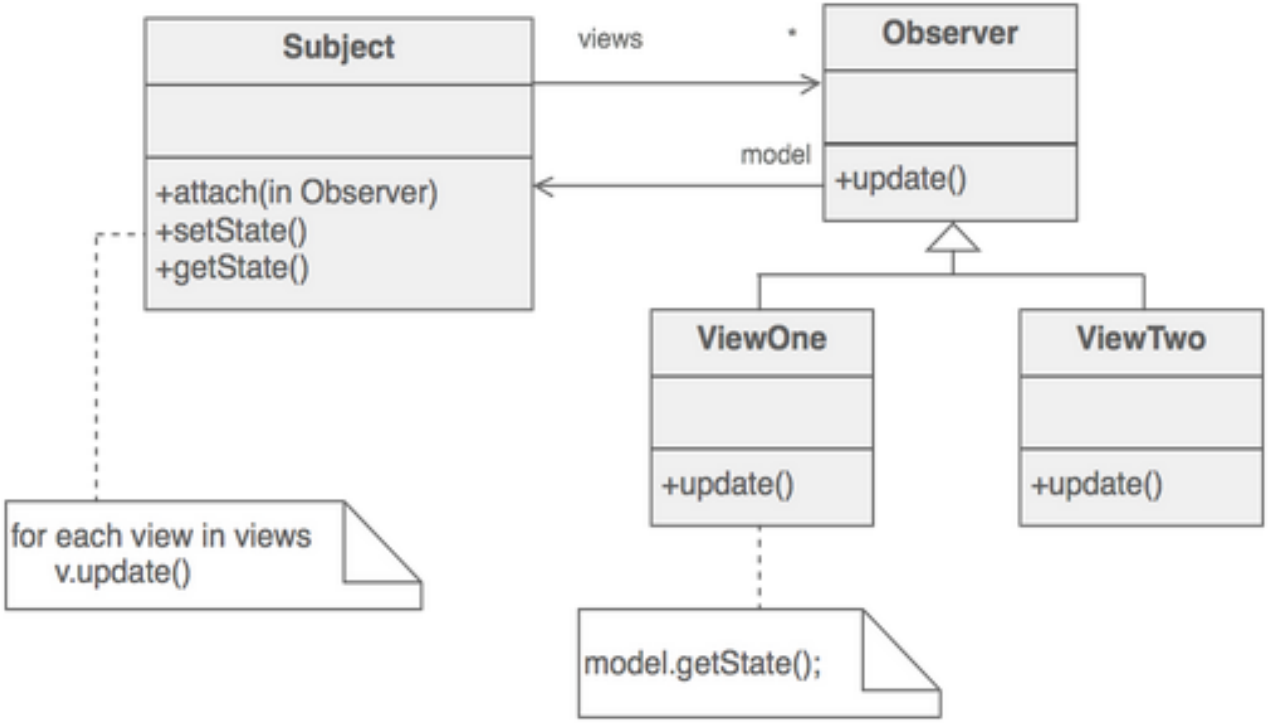
El “Observador”

- Una serie de objetos (observadores) quieren monitorear de cerca (ser informados) los cambios que experimenta un determinado objeto (observado)

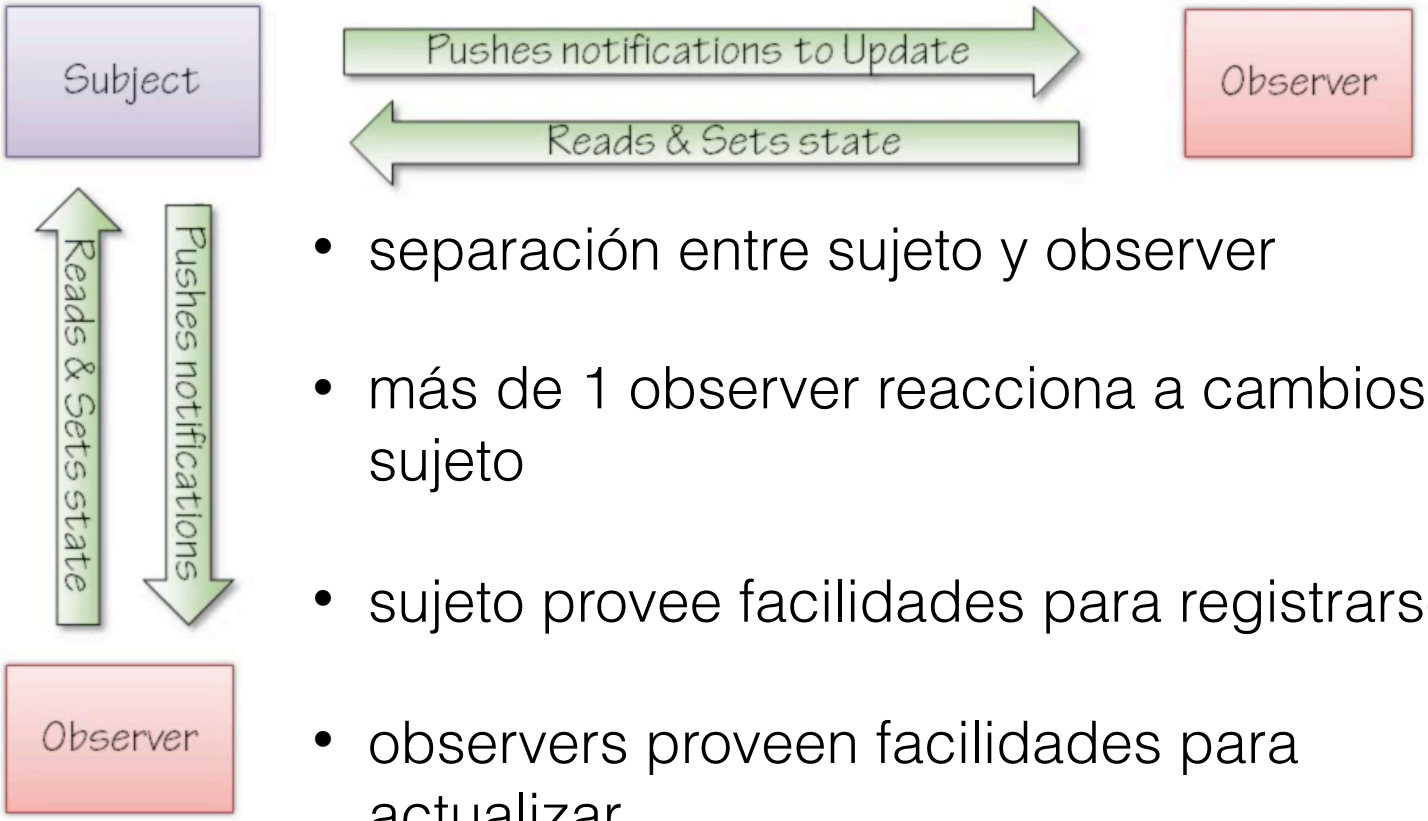
- Es una relación de tipo 1 a N



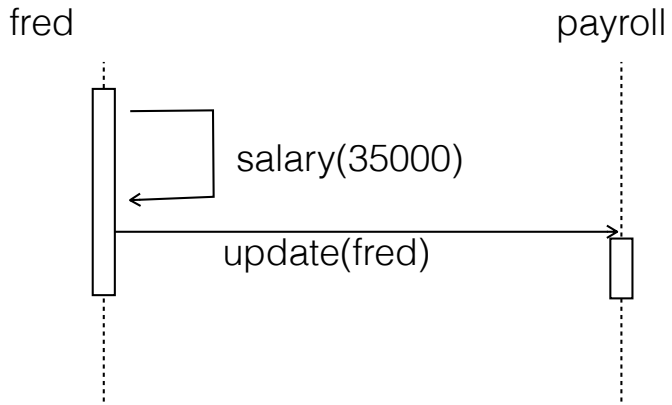
Uso común: Modelo y Vistas



Esencia del patrón Observer



- separación entre sujeto y observer
- más de 1 observer reacciona a cambios del sujeto
- sujeto provee facilidades para registrarse
- observers proveen facilidades para actualizar




```
class Employee
   attr_reader :name, :title
   attr_reader :salary
   def initialize( name, title, salary, payroll)
     @name = name
     @title = title
     @salary = salary
     @payroll = payroll
   end
   def salary=(new_salary)
     @salary = new_salary
     @payroll.update(self)
   end
end
```

```
class Payroll
  def update( changed_employee )
    puts("Cut a new check for #{changed_employee.name}!")
    puts("His salary is now #{changed_employee.salary}!")
  end
end
```

Implementación completa ...

```
class Employee

def initialize( name, title, salary )
  @name = name
  @title = title
  @salary = salary
  @observers = []
end

def salary=(new_salary)
  @salary = new_salary
  notify_observers
end

def notify_observers
  @observers.each do |observer|
    observer.update(self)
  end
end

def add_observer(observer)
  @observers << observer
end

def delete_observer(observer)
  @observers.delete(observer)
end

end
```

```
class Payroll
  def update( changed_employee )
    puts("Cut a new check for #{changed_employee.name}!")
    puts("His salary is now #{changed_employee.salary}!")
  end
end

class TaxMan
  def update( changed_employee )
    puts("Send #{changed_employee.name} a new tax bill!")
  end
end
```

```
payroll = Payroll.new
fred.add_observer( payroll )

tax_man = TaxMan.new
fred.add_observer(tax_man)

fred = Employee.new('Fred', 'Crane Operator', 30000.0)

fred.salary=80000
```

```
Cut a new check for Fred!
His salary is now 80000.0!
Send Fred a new tax bill!
```

Un problema y dos patrones

- Supongamos que queremos construir un generador de reportes mensuales de avance de un proyecto
- Informe tiene un título y un cuerpo
- Se ha decidido que formato del reporte debería ser HTML

Solución Directa "Naive"

```
class Report
  def initialize
    @title = 'Monthly Report'
    @text =  [ 'Things are going', 'really, really well.' ]
  end

  def output_report
    puts('<html>')
    puts('  <head>')
    puts("    <title>#{@title}</title>")
    puts('  </head>')
    puts('  <body>')
    @text.each do |line|
      puts("    <p>#{line}</p>")
    end
    puts('  </body>')
    puts('</html>')
  end

end

 report = Report.new
 report.output_report
```



```
<html>
  <head>
    <title> Montly Report </title>
  </head>
  <body>
    <p> Things are going </p>
    <p> really, really well </p>
  </body>
</html>
```

Requisitos cambian ...

- en realidad sería lindo poder emitir reportes en texto plano
- podríamos manejarlos con "ifs" pero ... quizá mas tarde en pdf u otros formatos
- la estructura de la solución es la misma
  - generar encabezados requeridos por formato
  - generar título
  - generar cuerpo
  - generar elementos finales requeridos por formato

Template Method

- El patrón de diseño *template method* permite capturar exactamente este problema
- El método plantilla captura la estructura de la generación del reporte usando operaciones genéricas

```
def output_report
    output_start
    output_head
    output_body_start
    output_body
    output_body_end
    output_end
end
```

```
class Report
  def initialize
    @title = 'Monthly Report'
    @text =  ['Things are going', 'really, really well.']
  end

  def output_report
    output_start
    output_head
    output_body_start
    output_body
    output_body_end
    output_end
  end

  def output_body
    @text.each do |line|
      output_line(line)
    end
  end

  def output_start
    raise 'Called abstract method: output_start'
  end

  def output_head
    raise 'Called abstract method: output_head'
  end

  def output_body_start
    raise 'Called abstract method: output_body_start'
  end

  def output_line(line)
    raise 'Called abstract method: output_line'
  end

  def output_body_end
    raise 'Called abstract method: output_body_end'
  end

  def output_end
    raise 'Called abstract method: output_end'
  end

end
```

```
class HTMLReport < Report
  def output_start
    puts('<html>')
  end
  def output_head
    puts('  <head>')
    puts("    <title>#{@title}</title>")
    puts('  </head>')
  end

  def output_body_start
    puts('<body>')
  end

  def output_line(line)
    puts("  <p>#{line}</p>")
  end

  def output_body_end
    puts('</body>')
  end

  def output_end
    puts('</html>')
  end

end
```

```
class PlainTextReport < Report
  def output_start
  end
  def output_head
    puts("**** #{@title} ****")
    puts
  end

  def output_body_start
  end


  def output_line(line)
    puts(line)
  end

  def output_body_end
  end

  def output_end
  end

end
```

y para producir los informes ...

```
report = HTMLReport.new
report.output_report
```

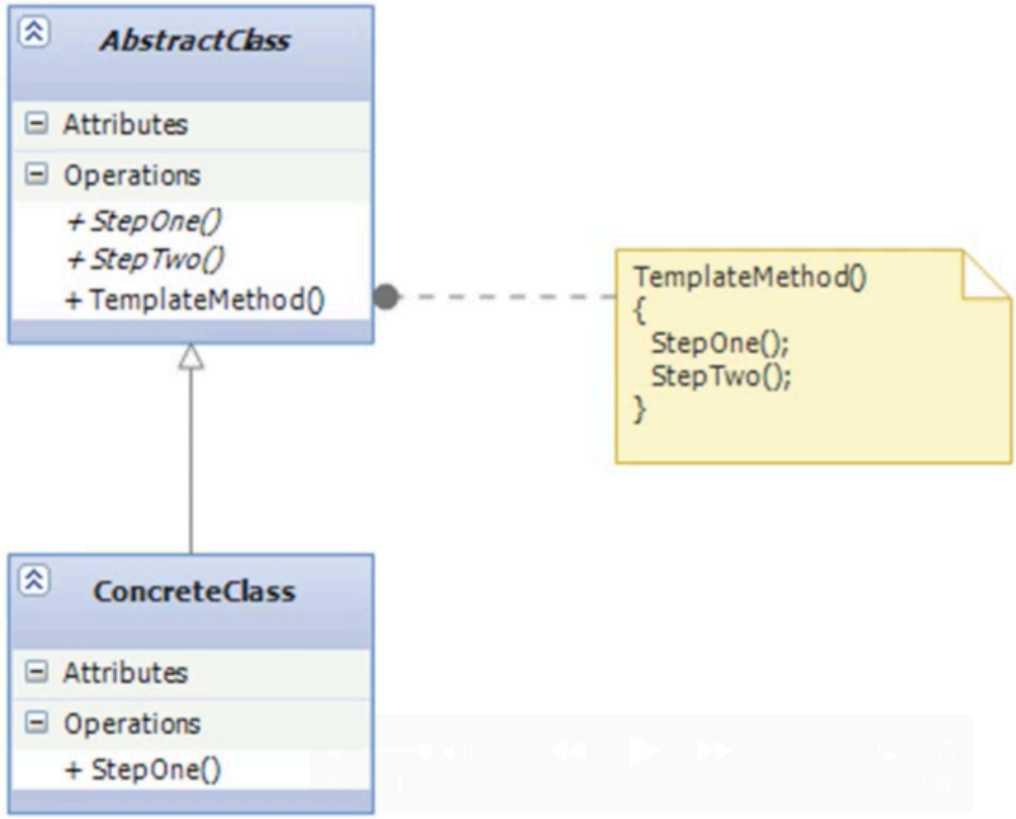
```
<html>
  <head>
    <title> Montly Report </title>
  </head>
  <body>
    <p> Things are going </p>
    <p> really, really well </p>
  </body>
</html>
```

```
report = PlainTextReport.new
report.output_report
```

```
**** Montly Report ****
Things are going
really, really well
```

Esencia del Patrón *Template Method*

- Se tiene un algoritmo o un proceso que comprende varios pasos
- Patrón permite variación en cómo se implementan los pasos manteniendo la estructura del proceso
- Subclases especializan el proceso

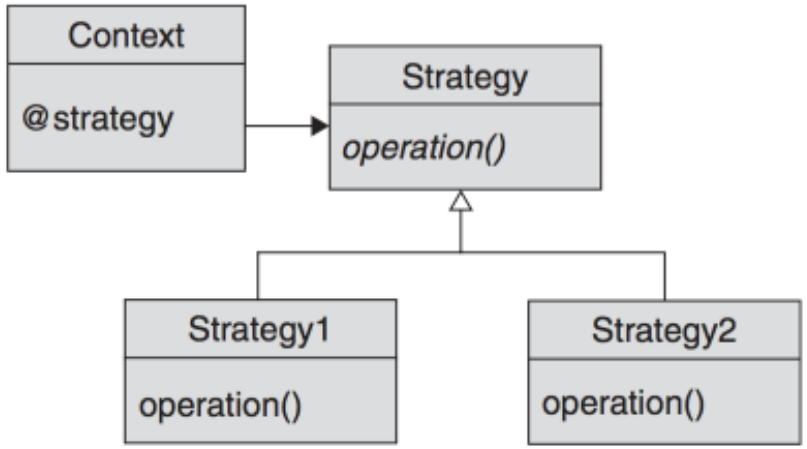


El patrón Strategy

- El problema que resuelve es básicamente el mismo
- La solución usa delegación en lugar de herencia (en el libro Gof se insiste mucho en esto)
- Delegación implica traspasar la responsabilidad de la operación a otro objeto que trabaja para el primero

La estrategia se ecapsula como clase

- Puede haber una estrategia general y otras subclases
- Estrategia general puede contener lo que es común en todos los casos
- En UML



```
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
```

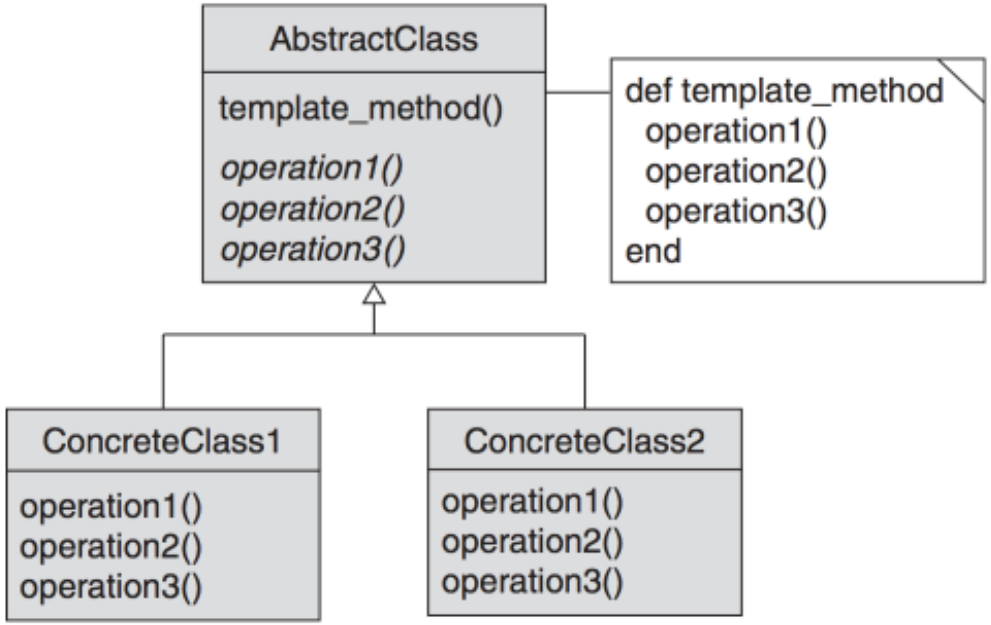
```
class HTMLFormatter
  def output_report( title, text )
    puts('<html>')
    puts('  <head>')
    puts("    <title>#{title}</title>")
    puts('  </head>')
    puts('  <body>')
    text.each do |line|
      puts("    <p>#{line}</p>" )
    end
    puts('  </body>')
    puts('</html>')
  end
end
```

```
class PlainTextFormatter
  def output_report(title, text)
    puts("***** #{title} *****")
    text.each do |line|
      puts(line)
    end
   end
end
```

```
report = Report.new(HTMLFormatter.new)
report.output_report

report = Report.new(PlainTextFormatter.new)
report.output_report
```

Estructura del Patrón en UML



Observaciones

- métodos de template no necesariamente deben ser abstractos
- por ejemplo se podrían haber dejado los métodos que no hacen nada como default y así PlainTextReport no tendría que redefinirlos
- un posible "pero" es que usa mecanismos de herencia para lograr su propósito