

## Capítulo 4

### Patrones y Patrones de Diseño (ii)

---

Diseño y Programación Orientado a Objetos  
Ingeniería Informática  
Ingeniería Técnica de Informática de Sistemas y Gestión  
Optativa (6 créditos)

<http://www.info-ab.uclm.es/assignaturas/42579>



## Bibliografía

---

- Design Patterns. Elements of Reusable Object-Oriented Software  
Gamma, e., Helm, R., Johnson, R., Vlissides, J.  
Addison Wesley, 1994 (Edición de 2003 en castellano)
- UML y Patrones. Introducción al análisis y diseño orientado a objetos  
Larman, G.  
Prentice Hall. 1999
- <http://webs.teleprogramadores.com/patrones/>
- <http://www.dofactory.com/patterns/patterns.aspx>
- <http://www.jacana.org.uk/pattern/>



## Patrones de creación. Resumen

---

- Hay dos formas de parametrizar un sistema: uno es con la herencia y otro es con la composición. Ejemplo del primer método es el **Factory Method** y ejemplo del segundo son los patrones **Abstract Factory**, **Builder** y **Prototype**
- **Abstract Factory** produce objetos de varias clases
- **Builder** construye un producto complejo incrementalmente
- En **Prototype** hace que su objeto fábrica construya un producto copiando un objeto prototípico
- Muchas veces los diseños comienzan usando el patrón **Factory Method** y luego evolucionan hacia los otros patrones de creación



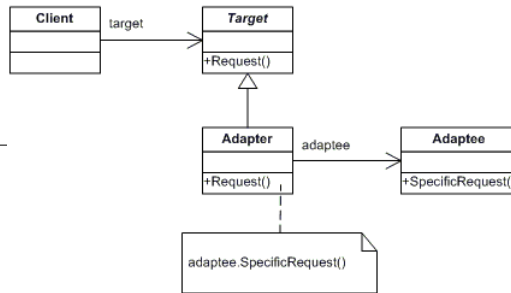
## Patrones de diseño. Estructural

---

- Adapter
- Bridge
- Composite
- Decorator
- Façade
- Flyweight
- Proxy



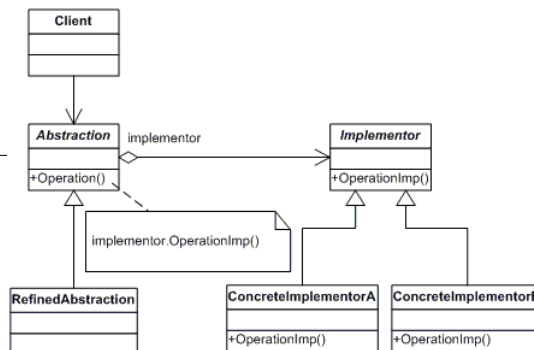
## Adapter



- **Target:** Define una interfaz de dominio específico que el cliente utiliza
- **Adapter** Adapta interfaces de Adaptee y Target
- **Adaptee:** Define una interfaz que existiendo necesita adaptación
- **Client:** utiliza objetos ajustándose a la interfaz ofrecida por Target



## Bridge

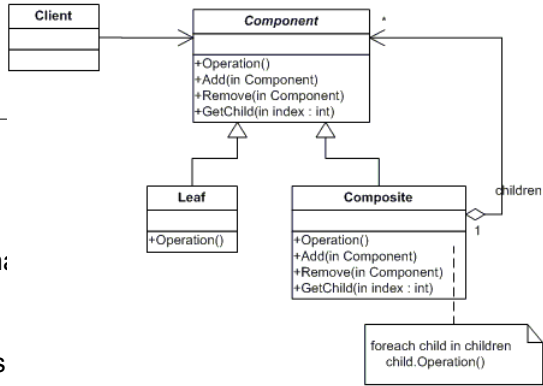


- **Abstraction:** define la interfaz de la abstracción
- **RefinedAbstraction:** Extiende la interfaz definida por la Abstraction
- **Implementor:** define la interfaz de las clases de implementación, no tiene porque corresponderse con el interfaz de Abstraction. Aquí se suelen proporcionar primitivas, en Abstraction operaciones de más alto nivel
- **ConcreteImplementor:** Implementa la interfaz de Implementor



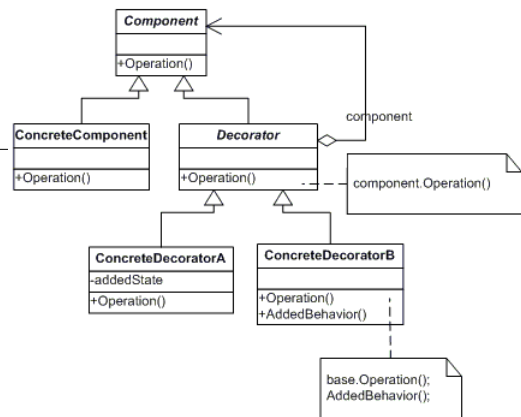
# Composite

- **Component:** declara la interfaz de los objetos involucrados en la composición, declara una interfaz para acceder y manejar a sus hijos
- **Leaf:** representa objetos hoja en la composición
- **Composite:** define el comportamiento de los componentes que tienen hijos, almacena los componentes hijo, implementa operaciones relacionadas con los hijos
- **Client:** manipula los objetos de la composición a través de Component



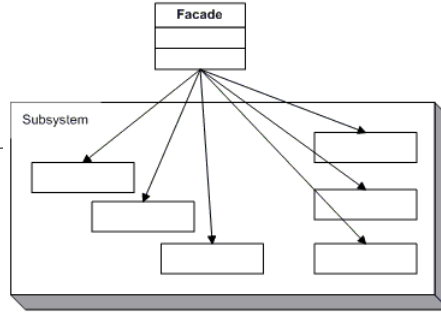
# Decorator

- **Component:** Define la interfaz que ofrecen los objetos que poseen responsabilidades añadidas dinámicamente
- **ConcreteComponent:** Define un objeto al que responsabilidades adicionales pueden serle añadidas
- **Decorator:** Mantiene referencia a un objeto Component y define una interfaz que conforma la interfaz del Component
- **ConcreteDecorator:** Añade las responsabilidades al Component

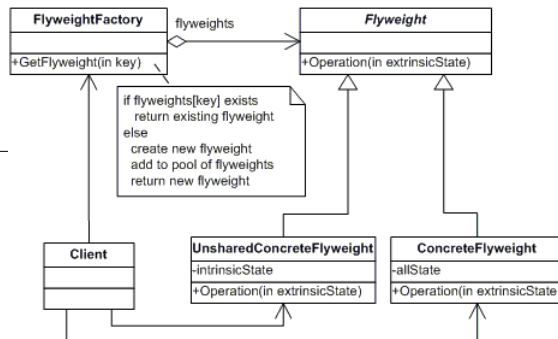


## Façade

- **Façade:** Conoce que clases son responsables de que peticiones y así, delega las peticiones a los objetos apropiados
- **Subsystem:** Implementa la funcionalidad del subsistema, realiza el trabajo solicitado por el objeto Façade y no conoce, ni mantiene referencia alguna del objeto Façade



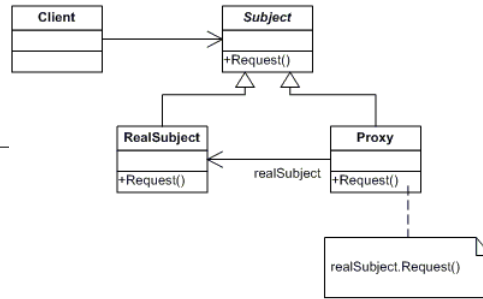
## Flyweight



- **Flyweight:** declaran una interface a través de la que flyweights pueden recibir y actuar sobre estados externos
- **ConcreteFlyweight:** Implementa la interfaz del flyweight y añade almacenamiento para el estado interno
- **UnsharedConcreteFlyweight:** no todas las subclases de Flyweight necesitan ser compartidas
- **FlyweightFactory:** crea y gestiona los objetos Flyweight
- **Client:** mantiene una referencia a objetos flyweights



## Proxy



- **Proxy**: tiene referencia al objeto RealSubject, tiene una interfaz idéntica a la de Subject así un proxy puede sustituirse por una RealSubject, y controla el acceso al RealSubject y puede ser el responsable de su creación y borrado
- **Subject**: define una interfaz común para RealSubject y Proxy
- **RealSubject**: define el objeto real que Proxy representa



## Patrones estructurales. Resumen (i)

- Los patrones **Adapter** y **Bridge** promueven la flexibilidad al proporcionar un nivel de indirección a otro objeto. Ambos implican reenviar peticiones a este objeto desde una interfaz distinta a la suya propia. Pero el patrón **Adapter** se centra en resolver incompatibilidades entre dos interfaces existentes, mientras que el patrón **Bridge** une una implementación con sus implementaciones
- Podemos ver un **Façade** como un **Adapter**
- **Composite** y **Decorator** usan la composición recursiva para organizar un número indeterminado de objetos. El patrón **Decorator** se utiliza para añadir responsabilidades a objetos sin crear subclases. El patrón **Composite** estructura subclases



## Patrones estructurales. Resumen (ii)

---

- **Decorator** y **Proxy** también tiene una estructura similar. Proxy compone un objeto y proporciona una interfaz idéntica a los clientes, pero **Proxy** no asigna o quita propiedades dinámicamente, su propósito es proporcionar un sustituto
- En el patrón **Proxy**, el sujeto define la principal funcionalidad y el proxy proporciona el acceso al mismo



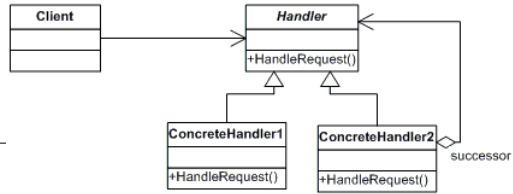
## Patrones de diseño. Comportamiento

---

- Chain of responsibility
- Command
- Interpreter
- Iterator
- Mediator
- Memento
- Observer
- State
- Strategy
- Template Method
- Visitor



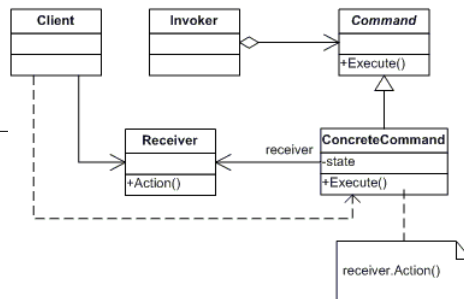
## Chain of responsibility



- **Handler**: define una interfaz para manejar las peticiones  
opcionalmente implementa enlaces de sucesión de atención de tales peticiones
- **ConcreteHandler**: recibe las peticiones y las atiende si es posible, de otra forma pasa la petición a su sucesor
- **Client**: hace las peticiones a los objetos ConcreteHandler pertenecientes a la cadena



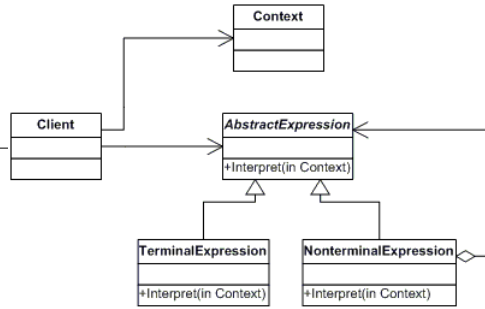
## Command



- **Command**: declara una interface para la ejecución de una operación
- **ConcreteCommand**: define un vínculo entre un objeto Receiver y una acción, implementa Execute() invocando al método correspondiente de Receiver
- **Cliente**: Crea un objeto ConcreteCommand y establece su receptor
- **Invoker**: pide a **Command** que lleve a cabo su petición
- **Receiver**: sabe cómo realizar las operaciones asociadas con la puesta en marcha de la petición



## Interpreter



- **AbstractExpression:** declara una interfaz para la ejecución de una operación
- **TerminalExpression:** Implementa una operación de Interpretación asociada con los símbolos terminales asociados con la gramática
- **NonterminalExpression:** implementa una operación de interpretación asociada con los símbolos no terminales asociados con la gramática
- **Context:** contiene información global para el interprete
- **Client:** construye un árbol sintáctico abstracto que representa una sentencia particular en el lenguaje que la gramática define

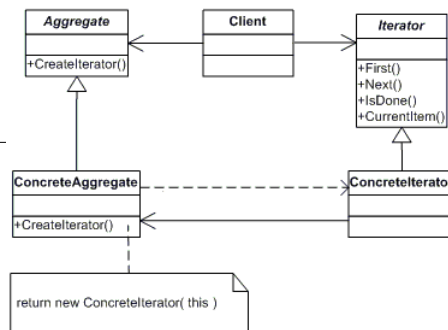


Curso 2003/04

Diseño y Programación  
Orientada a Objetos

17

## Iterator



- **Iterator** define una interfaz para atravesar y acceder a los elementos
- **ConcreteIterator:** Implementa la interfaz del iterator, mantiene un puntero al conjunto de elementos
- **Aggregate:** define un interfaz para crear un objeto iterator
- **ConcreteAggregator:** Implementa la interfaz de creación del Iterator devolviendo un objeto de ConcreteIterator apropiado

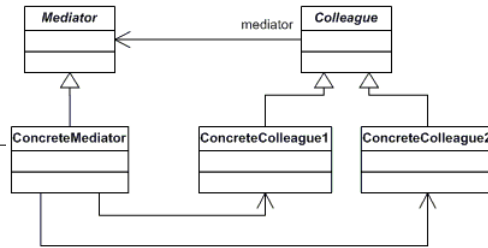


Curso 2003/04

Diseño y Programación  
Orientada a Objetos

18

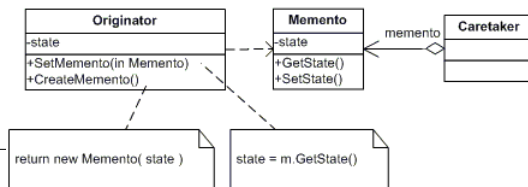
## Mediator



- **Mediator**: define una interfaz para la comunicación entre objetos Colleagues
- **ConcreteMediator**: Implementa un comportamiento cooperativo coordinado objetos Colleague
- **Colleague** classes: cada objeto Colleague conoce su objeto Mediator, cada colleague comunica con su mediator cuando tiene que comunicar con otro colleague



## Memento

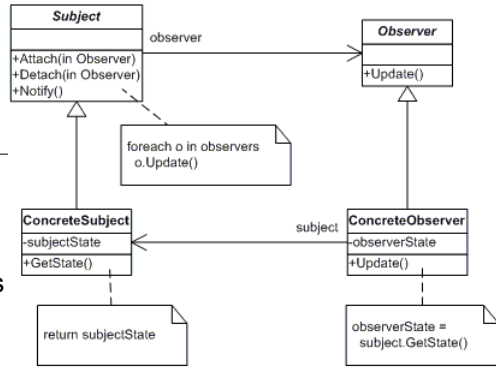


- **Memento**: Almacena el estado interno del objeto Originator, protege del acceso por parte de objetos distintos al Originator
- **Originator**: crea un objeto memento, que contiene una foto fija de su estado interno
- **Caretaker**: es el responsable de mantener la seguridad del objeto memento, no opera o examina el contenido de memento



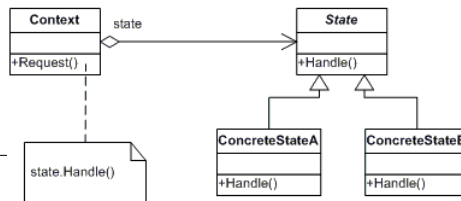
# Observer

- **Subject**: conoce sus observadores y proporciona una interfaz para gestionarlos
- **ConcreteSubject**: almacena el estado de interés y envía una notificación a sus observadores cuando su estado cambia
- **Observer**: define una interfaz para los objetos a los que debe notificarse el cambio de estado en ConcreteSubject
- **ConcreteObserver**: mantiene una referencia a un objeto ConcreteSubject, mantiene información consistente relacionada con el estado implementando el método Update()

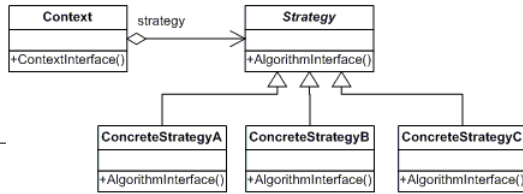


# State

- **Context**: define la interfaz de interés a clientes
- **State**: define una interfaz para encapsular el comportamiento asociado con un estado particular del contexto
- **ConcreteState**: cada subclase implementa un comportamiento asociado con un estado de Context



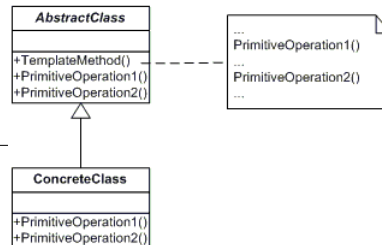
# Strategy



- **Strategy**: declara una interface común para dar soporte a todos los algoritmos Context utiliza esta interfaz para llamar al algoritmo definido por un ConcreteStrategy
- **ConcreteStrategy**: Implementa el algoritmo usando la interfaz Strategy
- **Context**: se configura con un objeto ConcreteStrategy, sobre el que mantiene una referencia, puede definir una interfaz que permita a Strategy acceder a sus datos



# Template Method



- **AbstractClass**: define operaciones primitivas abstractas cuya concreción se delega a las subclases, estas primitivas se utilizan en el cuerpo de un algoritmo esqueleto
- **ConcreteClass**: Implementa las operaciones primitivas abstractas anteriores

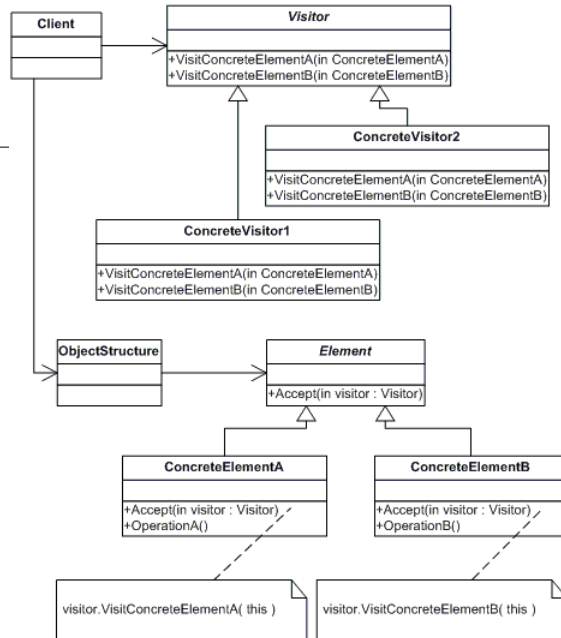


## Visitor

- **Visitor**: Establece las operaciones a realizar
- **ConcreteVisitor**: implementa dichas operaciones
- **Element**: define un método `Accept()` que toma un `Visitor` como argumento
- **ConcreteElement**: implementa el método `Accept()`



Curso 2003/04



Diseño y Programación  
Orientada a Objetos

25

## Patrones de comportamiento. Resumen (i)

- Encapsular aquello que puede variar es el tema de muchos patrones de comportamiento:
  - Un objeto **Estrategia** encapsula un algoritmo
  - Un objeto **Estado** encapsula el comportamiento dependiente del estado
  - Un objeto **Mediator** encapsula el protocolo entre objetos
  - Un objeto **Iterator** encapsula el modo en que se accede y se recorren los componentes de un objeto agregado
- Los patrones de comportamiento describen aspectos de un programa que es probable que cambien
- El patrón **Chain of Responsibility** describe el modo de comunicación entre un número indefinido de objetos



Curso 2003/04

Diseño y Programación  
Orientada a Objetos

26

## Patrones de comportamiento. Resumen (ii)

---

- Varios patrones introducen un objeto que siempre se usa como argumento (**Visitor**, **Command**, **Memento**)
- El **Mediator** y el **Observer** son patrones rivales. La diferencia entre ellos es que el patrón **Observer** distribuye la comunicación introduciendo objetos Observador y Sujeto, mientras que un objeto **Mediator** encapsula la comunicación entre objetos
- El patrón **Command** permite el desacoplamiento usando un objeto Orden que define un enlace entre un emisor y un receptor
- El patrón **Observer** desacopla a los emisores de los receptores
- El patrón **Mediator** desacopla los objetos haciendo que se refieran unos a otros indirectamente a través del Mediator



Curso 2003/04

Diseño y Programación  
Orientada a Objetos

27

## Capítulo 4 Patrones y Patrones de Diseño (ii)

---

Diseño y Programación Orientado a Objetos  
Ingeniería Informática  
Ingeniería Técnica de Informática de Sistemas y Gestión  
Optativa (6 créditos)

<http://www.info-ab.uclm.es/assignaturas/42579>

