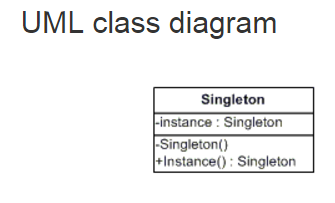
**Creational Patterns**

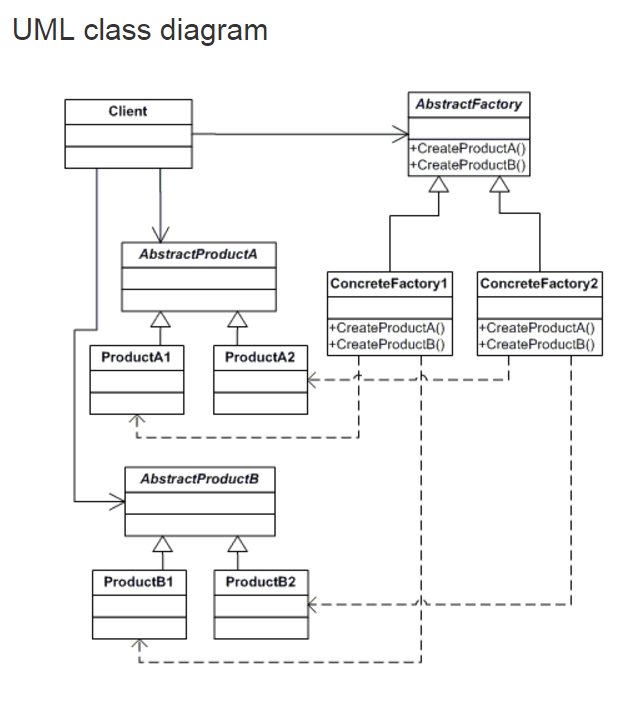
**SINGLETON PATTERN** –

patrón de diseño diseñado para restringir la creación de objetos pertenecientes a una clase o el valor de un tipo a un único objeto (una sola instancia creada en memoria). ANTI-SOLID.



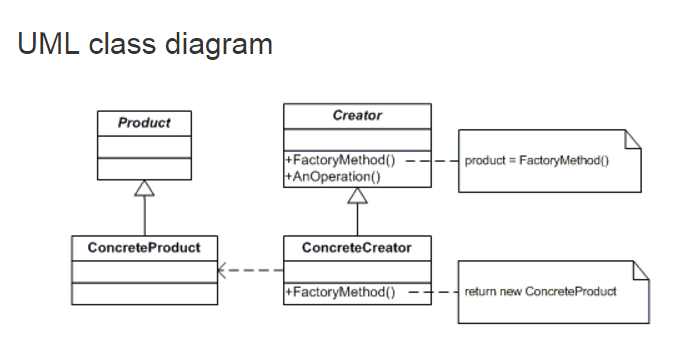
A**BSTRACT\_FACTORY PATTERN** –

Creates an instance of several families of clases. Provide an interface for creating families of related or dependent objects without specifying their concrete classes.



**FACTORY METHOD PATTERN**

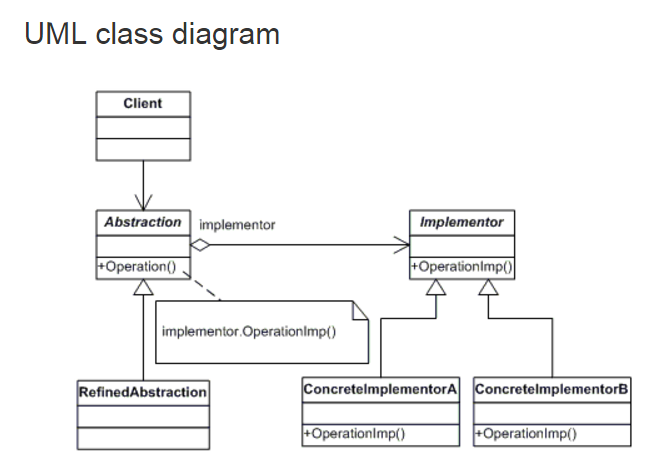
Creates an instance of several derived clases. Define an interface for creating an object, but let subclasses decide which class to instantiate. Factory Method lets a class defer instantiation to subclasses.



**Structural Patterns**

**BRIDGE PATTERN**

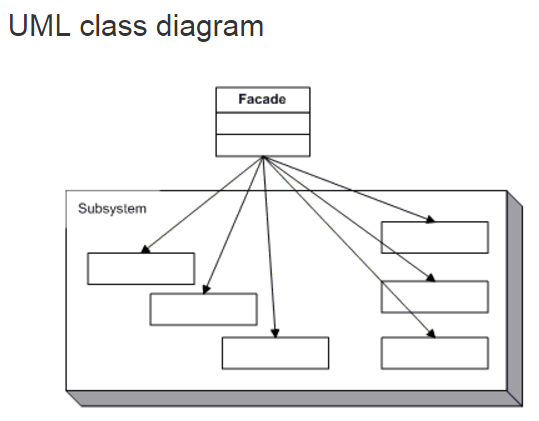
El patrón Bridge, también conocido como Handle/Body, es una técnica usada en programación para desacoplar una abstracción de su implementación, de manera que ambas puedan ser modificadas independientemente sin necesidad de alterar por ello la otra.



**FACADE PATTERN**

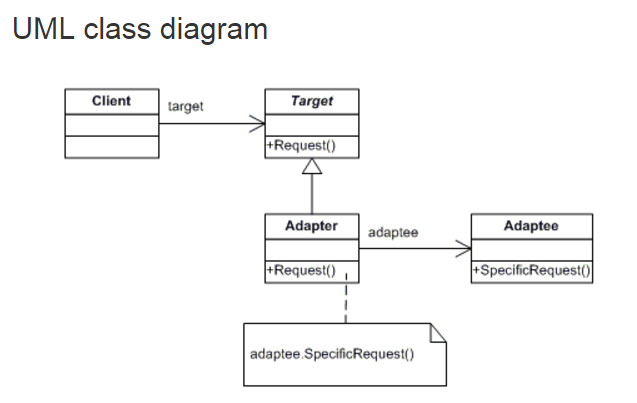
Provide a unified interface to a set of interfaces in a subsystem. Façade defines a higher-level interface that makes the subsystem easier to use.

Ejemplo: Se usa mucho para servicios web; encapsular, en una sola llamada /respuesta desde el cliente, varias llamadas "internas" en el facade a los servicios más pequeños que se necesitan para el response.



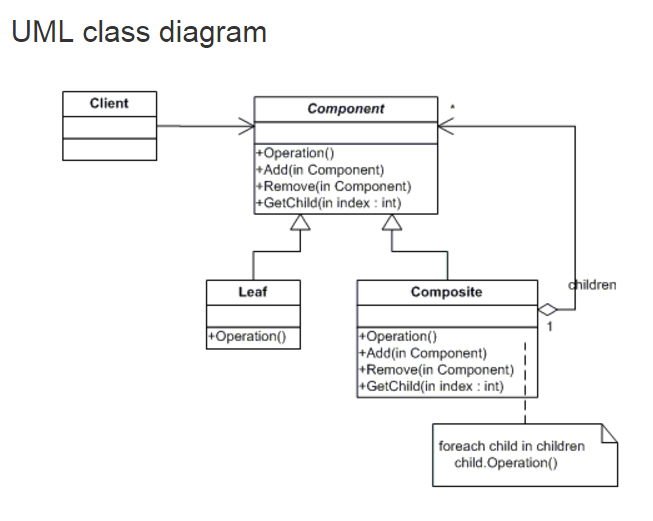
**ADAPTER PATTERN**

Convert the interface of a class into another interface clients expect. Adapter lets classes work together that couldn't otherwise because of incompatible interfaces. These incompatible classes may come from different libraries or frameworks.



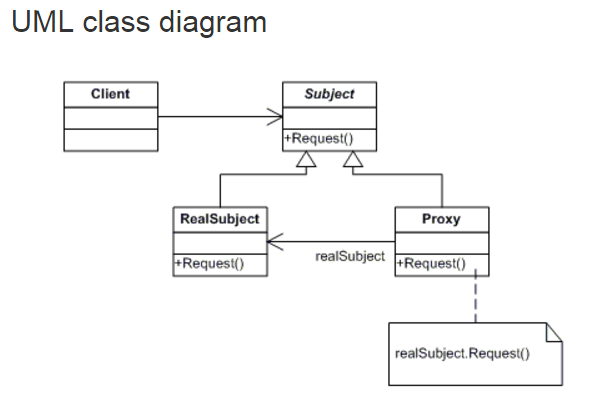
**COMPOSITE PATTERN**

Compose objects into tree structures to represent part-whole hierarchies. Composite lets clients treat individual objects and compositions of objects uniformly.



**PROXY**

Provide a surrogate or placeholder for another object to control access to it.



**Behavioral Patterns**

**COMMAND**

Encapsulate a request as an object, thereby letting you parameterize clients with different requests, queue or log requests, and support undoable operations.

The classes and objects participating in this pattern are:

Command - declares an interface for executing an operation

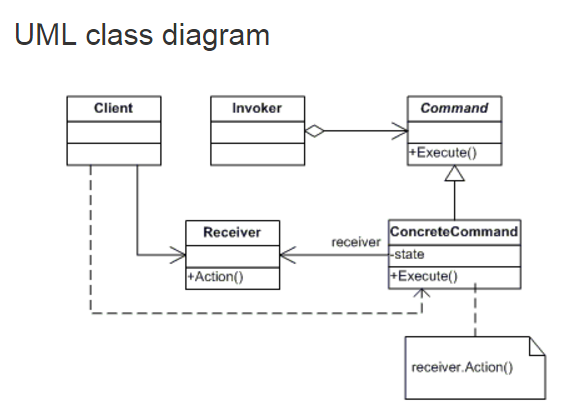
ConcreteCommand - defines a binding between a Receiver object and an action / implements Execute by invoking the corresponding operation(s) on Receiver

Client (Main) - creates a ConcreteCommand object and sets its receiver

Invoker - asks the command to carry out the request

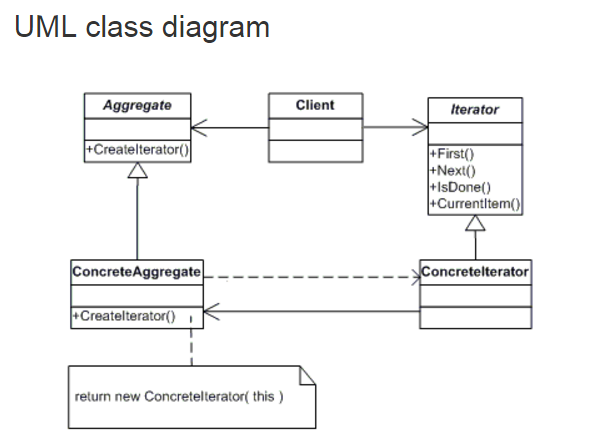
Receiver - knows how to perform the operations associated with carrying out the request.

La intermediación de comando desacopla emisor y receptor



**ITERATOR**

Sequentially access the elements of a Collection. Provide a way to access the elements of an aggregate object sequentially without exposing its underlying representation.

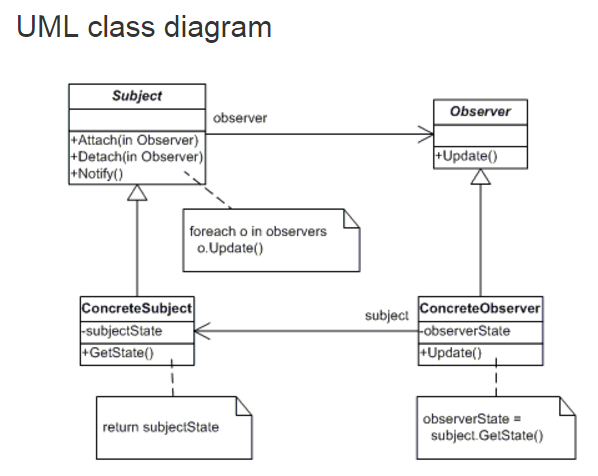


 The classes and objects participating in this pattern are:

* **Iterator**  **(AbstractIterator)**
  + defines an interface for accessing and traversing elements.
* **ConcreteIterator**  **(Iterator)**
  + implements the Iterator interface.
  + keeps track of the current position in the traversal of the aggregate.
* **Aggregate**  **(AbstractCollection)**
  + defines an interface for creating an Iterator object
* **ConcreteAggregate**  **(Collection)**
  + implements the Iterator creation interface to return an instance of the proper ConcreteIterator

**OBSERVER**

A way of notifying change to a number of classes. Define a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically.



The classes and objects participating in this pattern are:

* Subject  (Stock)
  + knows its observers. Any number of Observer objects may observe a subject
  + provides an interface for attaching and detaching Observer objects.
* ConcreteSubject  (IBM)
  + stores state of interest to ConcreteObserver
  + sends a notification to its observers when its state changes
* Observer  (IInvestor)
  + defines an updating interface for objects that should be notified of changes in a subject.
* ConcreteObserver  (Investor)
  + maintains a reference to a ConcreteSubject object
  + stores state that should stay consistent with the subject's
  + implements the Observer updating interface to keep its state consistent with the subject's