# 

# Ingeniería del Software: Laboratorio

### Cuaderno de prácticas. P2 - Diseño

## CONTENIDOS

El presente cuaderno de prácticas debe servir como guía al alumno para la asimilación y estudio de los siguientes contenidos:

* Patrones de Diseño

## RECOMENDACIÓN DE HERRAMIENTAS

* Argo UML (<http://argouml.tigris.org/>): Herramienta OpenSource.
* Otras alternativas: Herramientas de pago (Poseidon, Rational, …).

## Repaso Teórico. Introducción

El origen de los patrones se remonta a los años 70, Christopher Alexander, en sus trabajos, los aplicó para identificar y resolver, en un marco descriptivo formal, problemas esenciales en el domino de la arquitectura.

Muchas de sus ideas se trasladaron de este campo al de desarrollo software por el paralelismo que existe entre ambas materias. Por tanto se puede afirmar que Christopher Alexander ha servido, en realidad, de catalizador de ciertas tendencias “constructivas” utilizadas en el diseño de sistemas software.

Según Christopher Alexander “*Cada patrón describe un problema que ocurre una y otra vez en nuestro entorno, para describir después el núcleo de la solución a este problema, de tal manera que esa solución pueda ser usada más de un millón de veces sin hacerlo ni siquiera dos veces de la misma forma*.”

Entre 1990 y 1994, la Banda de los Cuatro (GoF, Gang of Four), formada por los autores Erich Gamma, Richard Helm, Ralph Johnson y John Vlissides, realiza el primer catálogo de patrones de diseño (Un total de 23) que incluyen en su libro *Design Patters: Elements of Reusable Object-Oriented Software* (1995).

La Banda de los Cuatro, no fueron los inventores, ni los únicos implicados, pero gracias a ellos estalló la fiebre de empezar a aplicar los patrones en los desarrollos software.

Los patrones de diseño se dividen en tres tipos: Patrones de Creación, Estructurales y Comportamiento.

Por último, mencionar a **Craig Larman**, que definió los Patrones Generales de Software para Asignar Respnsabilidades (GRASP, General Responsibility Assigment Software Patterns muy utilizados como buenas prácticas durante el diseño software. Los patrones GRASP vienen definidos en su libro *UML y Patrones: Introducción al Análisis y Diseño Orientado a Objetos y Proceso Unificado*.

Lo patrones de GRASP, no compiten con los patrones de diseño, sino que nos guían para encontrar los patrones de diseño (que son más concretos).

En esta asignatura únicamente se verán un par de patrones de diseño que ayudaran al alumno entender el funcionamiento y aplicación de los mismos.

## Repaso Teórico. Patrón Observador. Base teórica

Patrón Observador (Observer Pattern).

También conocido como Patrón Editor-Subscriptor (Publisher - Subscriptor Pattern) o Dependientes (Dependents Pattern).

Es un patrón de diseño de comportamiento y se aplica a nivel de objeto o de componente.

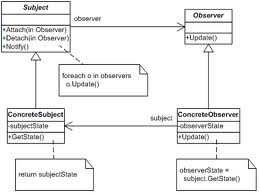
Este patrón permite definir dependencias de uno-a-muchos de forma que los cambios en un objeto se comuniquen a los objetos que dependen de él.

El Patrón Observador se suele utilizar cuando:

* Existe al menos un emisor en el mensaje.
* Uno o más receptores de mensajes podrían variar dentro de una aplicación o entre aplicaciones.
* Si se produce un cambio en un objeto se requiere el cambio de otro y no se sabe cuántos se necesitan cambiar. (Evitar Alto Acoplamiento).

Java tiene un API que proporciona la interfaz Observer y la clase Observable. Esto permite que muchos objetos reciban eventos de otro objeto en lugar de los sistemas de eventos básicos que sólo permiten notificar a un único objeto.

Su estructura en diagrama de clases es:



A continuación se pasa a detallar el rol que juega cada una de las clases.

* Sujeto (Subject). Lo observado. Interfaz que define como pueden interactuar los observadores con el sujeto. Define métodos para añadir y quitar observadores y avisarles de que se han producido cambios en el sujeto.
* SujetoConcreto (ConcreteSubject). Implementa la interfaz Sujeto. Contiene una lista de observadores a los que avisa cuando cambia su estado.
* Observador (Observer). Interfaz para actualizar los objetos ante cambios en un sujeto.
* ObservadorConcreto (ConcreteObserver). Mantiene una referencia a un objeto del SujetoConcreto. Implementa la interfaz Observador y define los métodos para responder a los mensajes recibidos del sujeto.

Las consecuencias de aplicar este patrón son:

* Desacoplamiento entre sujetos y observadores, convirtiéndolos en entidades reutilizadas por separado.
* Es un medio muy flexible de distribuir la información desde un objeto a muchos, de forma dinámica en tiempo de ejecución y sin que las clases implicadas sean conscientes del resto.
* El sujeto puede incluir cierta información en el mensaje de actualización de forma que cada observador pueda decidir si el cambio de estado le afecta o no.
* Un sujeto puede ser a su vez un observador respecto de otros.
* Un problema asociado es que un pequeño cambio en el sujeto puede provocar mucho procesamiento en los observadores.

## EJERCICIO TUTORIZADO. Patrón Observador

Se requiere desarrollar una aplicación informática que controle los diferentes cambios del valor de las acciones en la Bolsa para poder gestionar la compra venta de las mismas.

El valor de las acciones en bolsa está continuamente cambiando, en este tipo de situaciones es de vital importancia la detección de los distintos cambios en el momento que estos ocurran para cada una de las acciones. Por tanto, para resolver este problema se debe aplicar el Patrón Observador.

Para ello lo primero que se tiene que realizar es crear una clase Valor, que será aquella que contendrá la información de los distintos valores que tendrá en cada momento cada una de las acciones que coticen en bolsa.

La clase Valor tendrá:

* Atributos privados.
  + titulo del tipo String.
  + cotizacion del tipo double.
* Métodos públicos.
  + Constructor. Valor().
  + getTitulo() y setTitulo(). Para coger y devolver el título de la acción.
  + getCotizacion() y setCotizacion().Para coger y devolver el valor de la acción.
  + Además se deberá implementar un método que te pase el valor al tipo String.

Para aplicar el Patrón Observador se deberá adaptar la solución genérica del patrón a este problema en concreto. Para esto se deberán construir dos interfaces (Sujeto y Observador) y dos clases (SujetoConcreto y ObservadorConcreto).

La interfaz Sujeto tendrá:

* Métodos públicos.
  + getValor() y setValor(). Para coger y devolver el valor de la acción.
  + añadirObservador() y eliminarObservador().Para añadir o eliminar los distintos Observadores Concretos.
  + notificarObservadores().Se encarga de notificar el ObjservadorConcreto asignado al Sujeto Concreto.

La clase Sujeto Concreto tendrá:

* Atributos privados.
  + Un vector de observadores.
* Métodos públicos.
  + getValor() y setValor(). Para coger y devolver el valor de la acción.
  + añadirObservador() y eliminarObservador().Para añadir o eliminar los distintos Observadores Concretos.
  + notificarObservadores().Se encarga de notificar el ObjservadorConcreto asignado al Sujeto Concreto.

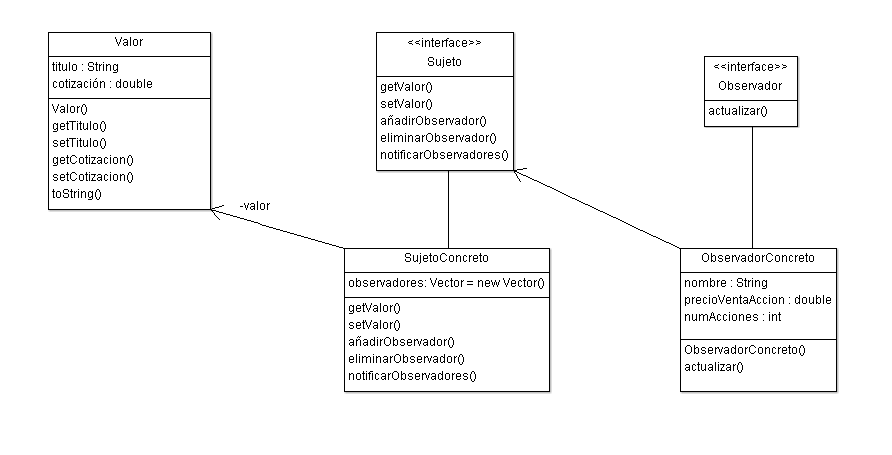
La interfaz Sujeto tendrá:

* Métodos públicos.
  + actualizar().Se encarga de actualizar el último valor de la acción.

La Clase Sujeto tendrá:

* Atributos privados.
  + nombre del tipo String.
  + precioVentaAcción del tipo doublé.
  + numAcciones del tipo entero.
* Métodos públicos.
  + Constructor. ObservadorConcreto().
  + actualizar().Se encarga de actualizar el último valor de la acción.

El diseño de clases de la implementación del Patrón Observador para el ejemplo indicado quedaría de la siguiente forma.



## Repaso Teórico. Patrón Comando. Base teórica

Patrón Comando (Command Pattern).

También conocido como Patrón Orden (Action or Transaction Pattern).

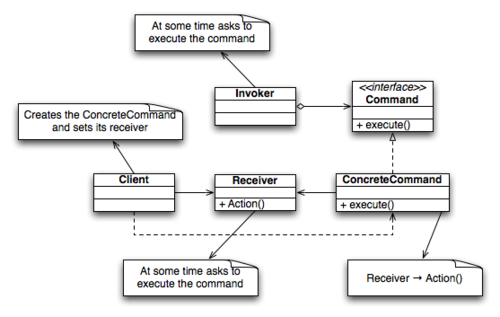
Es un patrón de diseño de comportamiento y se aplica a nivel de objeto o de componente.

Este patrón permite encapsular un comando en un objeto. Este objeto contiene el comportamiento y los datos necesarios para una acción específica. Permite parametrizar a los clientes con diferentes peticiones, hacer cola o llevar un registro de las peticiones. Además permite deshacer operaciones.

El Patrón Comando se suele utilizar cuando:

* Parametrizar objetos con una acción a realizar.
* Se quiere desacoplar la fuente de una petición del objeto que la cumple.
* Haya que implementar un mecanismo de rehacer y deshacer acciones.
* Permitir registrar los cambios de manera que se puedan volver a aplicar en caso de caída del sistema.
* Estructurar un sistema mediante operaciones de alto nivel basadas en operaciones más sencillas (primitivas). Sistemas de Transacciones.

Su estructura en diagrama de clases es:



A continuación se pasa a detallar el rol que juega cada una de las clases.

* Comando (Command). Interfaz en la que se definen los métodos que se usaran por el invocador (Invoker). Además estos métodos serán implementados por cada uno de los comandos concretos (ConcreteCommand). En el diagrama de clases únicamente se muestra el método ejecutar, pero se pueden añadir más métodos como deshacer y rehacer.
* ComandoConcreto (ConcreteCommand). Implementación de la interfaz Comando. Mantiene una referencia con el receptor. Implementa el método ejecutar para que realice la acción propia de cada comando concreto.
* Invocador (Invoker). El que llama al método ejecutar del objeto Comando.
* Receptor (Receiver). Para identificar que Comando cumple con la petición solicitada.
* Cliente (Client). Crea un objeto ComandoConcreto y establece su receptor.

## EJERCICIO TUTORIZADO. Patrón Comando

Se requiere desarrollar una aplicación informática que permita gestionar las reuniones de la organización a la que pertenecemos. La aplicación de gestión de reuniones, entre otras funciones, debe poder realizar un cambio de localización de la reunión. Por tanto, para resolver este problema se debe aplicar el Patrón Comando.

Para ello lo primero que se tiene que realizar es crear una clase Reunión, que será aquella que contendrá la información relacionada con la misma.

La clase Reunion tendrá:

* Atributos privados.
  + motivo del tipo String.
  + participantes del tipo String.
  + Localizacion del tipo String.
* Métodos públicos.
  + Constructor. Reunion().
  + getMotivo() y setMotivo(). Para coger y devolver el motivo de la reunión.
  + getParticipantes() y setParticipantes(). Para coger y devolver los participantes a la reunión.
  + getLocalizacion() y setLocalizacion(). Para coger y devolver la localización de la reunión.
  + getFecha() y setFecha(). Para coger y devolver la fecha y hora de la reunión.
  + Además se deberá implementar un método que te pase la fecha al tipo String.

Para poder gestionar la fecha y la hora es recomendable crear también una clase FechaHora que tendrá:

* Atributos privados.
  + año del tipo int.
  + mes del tipo int.
  + dia del tipo int.
  + hora del tipo int.
  + min del tipo int.

Para aplicar el Patrón Comando se deberá adaptar la solución genérica del patrón a este problema en concreto. Para esto se deberán construir dos interfaces (Comando y Comando Deshacer) y tres clases (ComandoCambiarLocalizacion, Invocador y Cliente).

La interfaz Comando tendrá:

* Métodos públicos.
  + getReunion() y setReunion(). Para coger y devolver el objeto Reunion.
  + ejecutar().Para ejecutar el comando en cuestión.

La interfaz ComandoDeshacer que heredará de la interfaz Comando tendrá:

* Métodos públicos.
  + Deshacer() y Rehacer().

La Clase Invocador tendrá:

* Métodos públicos.
  + setComando(). Coger el comando.
  + ejecutaComando(). Ejecutar el comando.
  + deshacerComando() y rehacerComando().

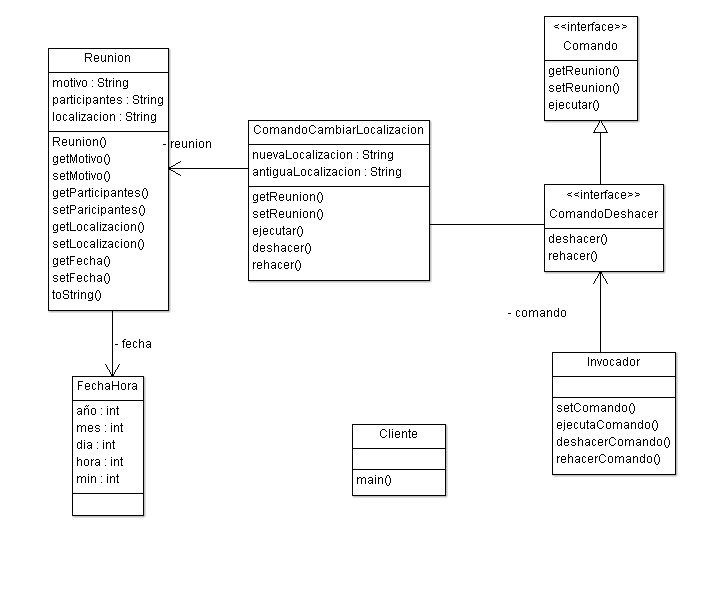
La Clase Cliente tendrá:

* Métodos públicos.
  + main().

La Clase ComandoCambiarLocalizacion tendrá:

* Atributos privados.
  + nuevaLocalizacion del tipo String.
  + antiguaLocalizacion del tipo String.
* Métodos públicos.
  + getReunion() y set Reunion. Para coger y devolver el objeto Reunion.
  + ejecutar().Se encarga de ejecutar el comando de cambiar localización.
  + deshacer() y rehacer().

El diseño de clases de la implementación del Patrón Observador para el ejemplo indicado quedaría de la siguiente forma.



## ENUNCIADO PARA EL ESTUDIO. Patrones de Diseño

Realizar el diagrama de clases aplicando patrones de diseño que sirva para la implementación una aplicación de subastas de obras de arte.

La aplicación mostrará una obra de arte (Nombre, Año, Autor). La subasta empezará con un precio base. Los participantes de la subasta podrán pujar por la obra de arte durante un periodo de tiempo determinado. Durante ese periodo en todo momento los participantes conocerán el precio más alto de la puja. Una vez finalizado ese tiempo, la obra de arte será adjudicada al participante que pujara más alto.

## ENUNCIADO PARA EL ESTUDIO. Patrones de Diseño

Realizar el diagrama de clases aplicando patrones de diseño que sirva para la implementación de un juego de simulación on-line de paddle.

Concretamente se deberá diseñar la forma de gestionar los resúmenes de los partidos jugados en el torneo de paddle que realizan los medios de comunicación. Estos resúmenes se podrán realizar a través de un medio escrito o por internet. Además el resumen del partido podrá incluir contenidos específicos o generales.