# Facade (patrón de diseño Fachada)

# Alumno: Elmer Condori Ballesteros

# Motivación

# El patrón fachada viene motivado por la necesidad de estructurar un entorno de programación y reducir su complejidad con la división en subsistemas, minimizando las comunicaciones y dependencias entre éstos.

# Consideraciones para su aplicación

# Se aplicará el patrón fachada cuando se necesite proporcionar una interfaz simple para un subsistema complejo, o cuando se quiera estructurar varios subsistemas en capas, ya que las fachadas serían el punto de entrada a cada nivel. Otro escenario proclive para su aplicación surge de la necesidad de desacoplar un sistema de sus clientes y de otros subsistemas, haciéndolo más independiente, portable y reutilizable (esto es, reduciendo dependencias entre los subsistemas y los clientes).

# Estructura

# Se puede ver en la siguiente figura:

# Facade UML class diagram.svg

# A continuación, se muestra un ejemplo:

# FacadeDesignPattern.png

# Participantes

# Fachada (*Facade*): conoce qué clases del subsistema son responsables de una determinada petición, y delega esas peticiones de los clientes a los objetos apropiados del subsistema.

# Subclases (*ModuleA*, *ModuleB*, *ModuleC*...): implementan la funcionalidad del subsistema. Realizan el trabajo solicitado por la fachada. No conocen la existencia de la fachada.

# Colaboraciones

# Los clientes que se comunican con el subsistema enviando peticiones al objeto Fachada, el cual las reenvía a los objetos apropiados del subsistema.

# Los objetos del subsistema realizan el trabajo final y la fachada hace algo de trabajo para pasar de su interfaz a las del subsistema.

# Los clientes que usan la fachada no tienen que acceder directamente a los objetos del subsistema.

# Ventajas e inconvenientes

# La principal ventaja del patrón fachada consiste en que para modificar las clases de los subsistemas, sólo hay que realizar cambios en la interfaz/fachada, y los clientes pueden permanecer ajenos a ello. Además, y como se mencionó anteriormente, los clientes no necesitan conocer las clases que hay tras dicha interfaz.

# Como inconveniente, si se considera el caso de que varios clientes necesiten acceder a subconjuntos diferentes de la funcionalidad que provee el sistema, podrían acabar usando sólo una pequeña parte de la fachada, por lo que sería conveniente utilizar varias fachadas más específicas en lugar de una única global.

# Patrones relacionados

# Uno de los patrones relacionados más directamente es el [singleton](http://es.wikipedia.org/wiki/Singleton_(patr%C3%B3n_de_dise%C3%B1o)" \o "Singleton (patrón de diseño)), dado que en determinadas ocasiones las fachadas pueden ser instancias únicas.

# Otros patrones que guardan una cierta relación con el patrón fachada son los [GRASP](http://es.wikipedia.org/wiki/GRASP) (*General Responsibility Assignment Software Patterns*), los cuales no son patrones de diseño, sino buenas prácticas que guían al desarrollador para encontrar los patrones de diseño, que son más concretos. Uno de los patrones GRASP es un controlador que actúa como punto de entrada en la capa lógica, lo que se puede comparar perfectamente con el uso del patrón fachada.

# Usos conocidos (Problemas/Soluciones)

# Problema: Un [cliente](http://es.wikipedia.org/wiki/Cliente_(inform%C3%A1tica)) necesita acceder a parte de la funcionalidad de un sistema más complejo.

# Definir una interfaz que permita acceder solamente a esa funcionalidad.

# Problema: Existen grupos de tareas muy frecuentes para las que se puede crear código más sencillo y legible.

# Definir funcionalidad que agrupe estas tareas en funciones o métodos sencillos y claros.

# Problema: Una biblioteca es difícilmente legible.

# Crear un intermediario más legible.

# Problema: Dependencia entre el código del cliente y la parte interna de una biblioteca.

# Crear un intermediario y realizar llamadas a la biblioteca sólo o, sobre todo, a través de él.

# Problema: Necesidad de acceder a un conjunto de APIs que pueden además tener un diseño no muy bueno.

# Crear una API intermedia, bien diseñada, que permita acceder a la funcionalidad de las demás.

# Problema: Muchas clases cliente quieren usar varias clases servidoras, y deben saber cuál es exactamente la que le proporciona cada servicio. El sistema se volvería muy complejo, porque habría que relacionar todas las clases cliente con todas y cada una de las clases servidoras.

# Crear una o varias clases Facade, que implementen todos los servicios, de modo que o todos los clientes utilicen esa única clase, o que cada grupo de clientes use la fachada que mejor se ajuste a sus necesidades.

# 

# 

# 