Universidad Autónoma de Nuevo León
Facultad de Ciencias Físico Matemáticos

Licenciado en Seguridad en Tecnologías de Información

Diseño Orientado a Objetos

Patrones de diseño

Tarea

Lic. Miguel Ángel Salazar Santillán

Grupo: 007 Matrícula: 1732645

Diana Elizabeth Díaz Rodríguez

14/ 03/ 2017

¿Qué es un Patrón de Diseño?

Los patrones de diseño son el esqueleto de las soluciones a problemas comunes en el desarrollo de software. Brindan una solución ya probada y documentada a problemas de desarrollo de software que están sujetos a contextos similares. Los patrones de diseño son la base para la búsqueda de soluciones a problemas comunes en el desarrollo de software y otros ámbitos referentes al diseño de interacción o interfaces.

¿Para qué sirven?

Un patrón de diseño resulta ser una solución a un problema de diseño. Para que una solución sea considerada un patrón debe poseer ciertas características. Una de ellas es que debe haber comprobado su efectividad resolviendo problemas similares en ocasiones anteriores. Otra es que debe ser reutilizable, lo que significa que es aplicable a diferentes problemas de diseño en distintas circunstancias.

Un patrón de diseño debe cumplir al menos con los siguientes objetivos

- Estandarizar el lenguaje entre programadores
- Evitar perder tiempo en soluciones a problemas ya resueltos o conocidos
- Crear código reusable (excelente ventaja)
- Elementos y los principales componentes del sistema.
- Capas del sistema.
- Mecanismos arquitecturales necesarios para el sistema.
- Tecnologías elegidas para hacer frente a cada capa y el mecanismo y la motivación detrás de estas opciones.
- Componentes comprados o de código abierto que se utilizarán y la forma en que se tomó la decisión.
- Descripción de los principales patrones de diseño utilizados y por qué la elección.
- Definición de la forma en que el acceso a sistemas externos y / o legados.

Patrones Estructurales

- Adaptador (*Adapter*): Convierte una interfaz en otra.
- Puente (*Bridge*): Desacopla una abstracción de su implementación permitiendo modificarlas independientemente.
- Objeto Compuesto (*Composite*): Utilizado para construir objetos complejos a partir de otros más simples, utilizando para ello la composición recursiva y una estructura de árbol.
- Envoltorio (*Decorator*): Permite añadir dinámicamente funcionalidad a una clase existente, evitando heredar sucesivas clases para incorporar la nueva funcionalidad.
- Fachada (*Facade*): Permite simplificar la interfaz para un subsistema.
- Peso Ligero (*Flyweight*): Elimina la redundancia o la reduce cuando tenemos gran cantidad de objetos con información idéntica.

- Apoderado (*Proxy*): Un objeto se aproxima a otro.

Patrones de Comportamiento

- Cadena de responsabilidad (*Chain of responsibility*): La base es permitir que más de un objeto tenga la posibilidad de atender una petición.
- Orden (*Command*): Encapsula una petición como un objeto dando la posibilidad de "deshacer" la petición.
- Intérprete (*Interpreter*): Intérprete de lenguaje para una gramática simple y sencilla.
- Iterador (*Iterator*): Define una interfaz que declara los métodos necesarios para acceder secuencialmente a una colección de objetos sin exponer su estructura interna.
- Mediador (*Mediator*): Coordina las relaciones entre sus asociados. Permite la interacción de varios objetos, sin generar acoples fuertes en esas relaciones.
- Recuerdo (*Memento*): Almacena el estado de un objeto y lo restaura posteriormente.
- Observador (*Observer*): Notificaciones de cambios de estado de un objeto.
  End Class
- Estado (*Server*): Se utiliza cuando el comportamiento de un objeto cambia dependiendo del estado del mismo.
- Estrategia (*Strategy*): Utilizado para manejar la selección de un algoritmo.
- Método plantilla (*Template Method*): Algoritmo con varios pasos suministrados por una clase derivada.
- Visitante (*Visitor*): Operaciones aplicadas a elementos de una estructura de objetos heterogénea.

Patrones creacionales

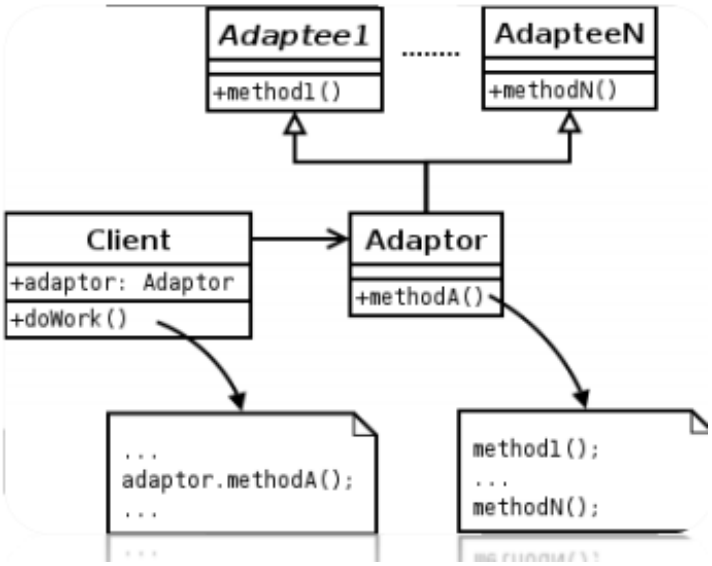
Corresponden a patrones de diseño software que solucionan problemas de creación de instancias. Nos ayudan a encapsular y abstraer dicha creación:

- Object Pool (no pertenece a los patrones especificados por GoF): se obtienen objetos nuevos a través de la clonación. Utilizado cuando el costo de crear una clase es mayor que el de clonarla. Especialmente con objetos muy complejos. Se especifica un tipo de objeto a crear y se utiliza una interfaz del prototipo para crear un nuevo objeto por clonación. El proceso de clonación se inicia instanciando un tipo de objeto de la clase que queremos clonar.
- Abstract Factory (fábrica abstracta): permite trabajar con objetos de distintas familias de manera que las familias no se mezclen entre sí y haciendo transparente el tipo de familia concreta que se esté usando. El problema a solucionar por este patrón es el de crear diferentes familias de objetos, como por ejemplo, la creación de interfaces gráficas de distintos tipos (ventana, menú, botón, etc.).
- Builder (constructor virtual): abstrae el proceso de creación de un objeto complejo, centralizando dicho proceso en un único punto.
- Factory Method (método de fabricación): centraliza en una clase constructora la creación de objetos de un subtipo de un tipo determinado, ocultando al usuario la casuística, es decir, la diversidad de casos particulares que se pueden prever, para elegir el subtipo que crear. Parte del principio de que las subclases determinan la clase a implementar. A continuación se muestra un ejemplo de este patrón:

Patrón De diseño Adapter

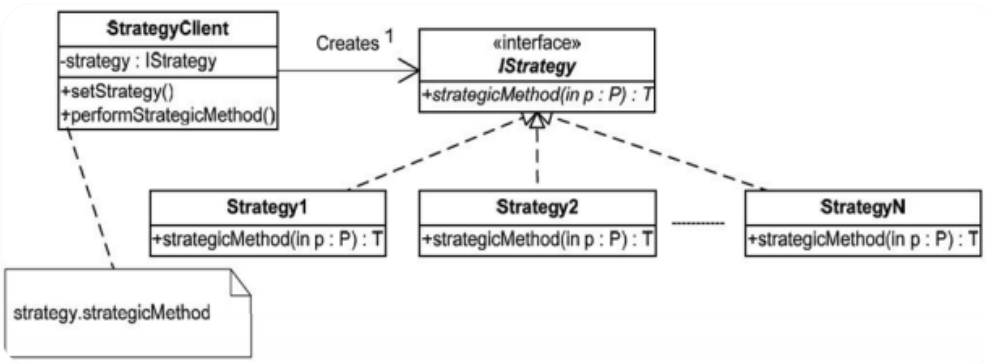
El patrón Adapter se utiliza para transformar una interfaz en otra, de tal modo que una clase que no pudiera utilizar la primera, haga uso de ella a través de la segunda.

- Propósito: Convierte la interfaz de una clase en otra interfaz que el cliente espera. Adapter permite a las clases trabajar juntas, lo que de otra manera no podría hacerlo debido a sus interfaces incompatibles.
- Estructura de Adapter: Una clase adaptadora usa múltiples herencias para adoptar una interfaz a otra



Patrón De diseño Strategy

Strategy es un patrón de diseño para el desarrollo de software. El patrón Strategy permite mantener un conjunto de algoritmos de los que el objeto cliente puede elegir aquel que le conviene e intercambiarlo según sus necesidades. Los distintos algoritmos se encapsulan y el cliente trabaja contra un objeto contexto o Context. Como hemos dicho, el cliente puede elegir el algoritmo que prefiera de entre los disponibles o puede ser el mismo objeto Context el que elija el más apropiado para cada situación. Cualquier programa que ofrezca un servicio o función determinada, que pueda ser realizada de varias maneras, es candidato a utilizar el patrón Strategy. Puede haber cualquier número de estrategias y cualquiera de ellas podrá ser intercambiada por otra en cualquier momento, incluso en tiempo de ejecución.


StrategyClient
-strategy : IStrategy
+setStrategy()
+performStrategicMethod()
Creates
1
«interface»
IStrategy
+strategicMethod(in p : P) : T
Strategy1
+strategicMethod(in p : P) : T
Strategy2
+strategicMethod(in p : P) : T
StrategyN
+strategicMethod(in p : P) : T
strategy.strategicMethod