**-- PATRONES GRASP--**

En sistemas orientados a objetos son objetos que se relacionan entre ellos para realizar las instrucciones que invocan los usuarios.

Sirven para ver de qué manera podemos asignar responsabilidades a las clases de nuestro diseño, son guías para asignar una responsabilidad.

Los tipos de patrones GRASP son:

* Experto
* Creador
* Alta cohesión
* Bajo acoplamiento
* Controlador
* Polimorfismo
* Fabricación pura
* Indirección
* No hables con extraños

Para saber cuál usar en cada momento es necesario asociarlos a directrices.

* Cuando las clases deben tener responsabilidades bien definidas:

1. Patrón experto
2. Patrón creador
3. Patrón controlador

* Cuando las clases deben delegar en otras clases:
  1. Patrón indireccion
* Cuando los objetos ligados a herencia usen lógicas CASE:
  1. Patrón polimorfismo
* Si aparecen clases no relacionadas con el dominio del problema pero si con el dominio de la solución
  1. Fabricación pura
* Cuando las clases deben tener alta cohesión
  1. Patrón alta cohesión
* Cuando las clases tienen bajo acoplamiento
  1. Bajo acoplamiento
  2. No hables con extraños

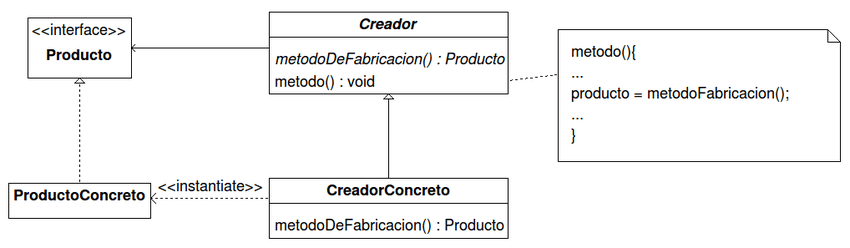
**-- PATRON CREADOR –**

El problema que afronta es asignarle responsabilidades de crear un objeto de una determinada clase.

La solución para que la clase B sea responsable de crear objetos de clase A podría ser una de estas:

* Clase B agrega objetos de clase A
* Clase B contiene objetos de clase A
* Clase B registra objetos de clase A
* Clase B utiliza a clase A para su propia realización
* Clase B contiene datos para inicializar clase A

En estos casos diremos que la clase B es experta de creación de clase A.



Este patrón sirve de guía para decir a quien se le asigna la responsabilidad de crear objetos. Su propósito es encontrar un creador que debemos conectar al objeto producido.

El beneficio principal es que proporciona clases con bajo acoplamiento.

**-- PATRON EXPERTO --**

Este patrón nos transmite el criterio fundamental para asignar responsabilidades en un diseño orientado a objetos. Nos dice la clase que contiene la información que hace posible llevar a cabo esa responsabilidad. Este patrón es de los más usados de los GRASP.

Con este patrón las responsabilidades están claras y si cada uno las cumple se realizaran las tareas de manera más eficiente. A veces, para cumplir una responsabilidad se necesitan la colaboración de todos los usuarios y se crean los expertos parciales (cada persona controla sobre un determinado tema y luego trabajan juntos para obtener un resultado favorable).

Los beneficios que se consiguen con este patrón son:

* Se conserva el encapsulamiento y disminuye el acoplamiento
* Promueve clases sencillas y cohesivas

**-- PATRON BAJO ACOPLAMIENTO --**

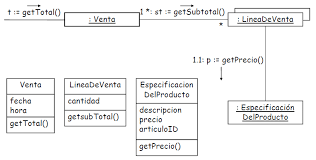
Con este patrón las clases dependen poco unas de otras. A la hora de asignar una nueva responsabilidad se mantiene el número de relaciones evitando una alta dependencia.

En los Lenguajes Orientados a Objetos las formas más comunes entre tipo A y tipo B son:

* Tipo A tiene un atributo de tipo B
* Tipo A tiene un método que utiliza un objeto de tipo B
* Tipo A es subclase de tipo B
* Tipo B es una interfaz y tipo B la implementa

Este patrón sirve para diseñar clases con relativa dependencia. Así al hacer un cambio no hay un impacto grande.

Los beneficios de este patrón son:

* Los cambios en otros componentes no afectan
* Las clases son fáciles de entender por separado
* Las clases son fáciles de reutilizar

Este es el patrón bajo acoplamiento aplicado a un ejemplo de ventas.

**-- PATRON ALTA COHESION--**

Lo que sugiere es que todas las responsabilidades que tiene una sola clase estén muy relacionadas. Para ello, añadimos nuevas responsabilidades a clases que ya tienen responsabilidades similares.

Una clase con alta cohesión posee pocas responsabilidades pero luego comparten el esfuerzo cuando la tarea es grande.



En este ejemplo, a lo que se refiere el patrón es que desde registro se podrían abir dos “caminos”: uno a venta y otro pago. Con el patrón de alta cohesión, no hace falta ya que siguen un camino único.

Este patrón es fácil de entender, mantener y reutilizar. Mejora la claridad y la facilidad de entender diseños. Gracias a este patrón se genera bajo acoplamiento y se consigue simplificar el mantenimiento.

**-- PATRON CONTROLADOR--**

Un evento del sistema es un evento de alto nivel generado por un actor externo. El patrón controlador asigna quien se encarga de un nuevo evento.

La solución que propone este patrón es asignar la responsabilidad a una clase que represente el sistema, la empresa o algo que pueda participar en la tarea como puede ser un actor o un manejador artificial de todos los eventos del sistema.

Beneficios:

* Nos dice quien toma las decisiones sobre los eventos de entrada
* Es más un servicio de aplicación que un controlador

Nota: no confundir con el patrón modelo vista controlador de los patrones GOF.

**-- PATRON FABRICACION PURA--**

Nos dice a quién asignar una responsabilidad cuando en nuestro diseño no hay nadie que se pueda encargar de esa tarea sin violar los patrones de alta cohesión y bajo acoplamiento.

Para ello, nos inventamos un nuevo objeto una nueva clase y le asignamos una nueva clase y le asignamos responsabilidades que estén altamente cohesionadas.

Con este patrón creamos clases de granularidad muy fina, es decir, las responsabilidades son muy concretas. Los patrones adaptador, observador o visitante son ejemplos de este.

Beneficios:

* Alta cohesión
* Aumenta el potencial de reutilización

**-- PATRON INDIRECCION--**

Nos dice a quién debemos asignarle una nueva responsabilidad intentando que la clase a la que añado esa responsabilidad no se tenga que acoplar con una clase que, en principio, no estaba acoplada con ella.

Se suele usar un objeto intermedio. Su principal beneficio es el bajo acoplamiento que se consigue.

**-- PATRON NO HABLES CON EXTRAÑOS--**

Con este patrón asignamos una responsabilidad para evitar conocer a objetos o clases que todavía no conocemos.

El objetivo es asignar responsabilidades a objetos que sí conocen a otros objetos que no queremos conocer y así evitamos una nueva relación entre dos clases.

Es también llamado Ley de Deméter; que impone restricciones respecto a quien debemos enviarle mensajes:

* Al propio objeto
* A objetos que nos vengan como parámetro del método que se esta ejecutando
* A nuestros propios atributos
* A elementos que están en una colección que sea parte de nuestros atributos
* A objetos que hayamos creado en el método de nuestra ejecución

Con esto evitamos una visibilidad innecesaria de objetos indirectos y evitamos acoplamientos innecesarios.

Su beneficio es el bajo acoplamiento.