INGENIERÍA DE SOFTWARE

Principios de Diseño

Tipos de Patrones

Hay muchas posibles taxonomías de los patrones de la ingeniería del software:

- Patrones de Análisis: proporcionan maneras probadas de representar conceptos generales del mundo real en un diagrama estático del análisis.
- Patrones de asignación de responsabilidad: documentan los principios de diseño en forma de patrones.
- Patrones de Diseño: se aplican para resolver problemas concretos de diseño.
- Patrones de Programación: describen cómo se pueden implementar ciertas tareas utilizando un lenguaje de programación concreto.

¿Qué conoce un programador experto que desconoce uno inexperto?

Reutilizar soluciones que funcionaron en el pasado:

EXPERIENCIA

"Diseñar software orientado a objetos es difícil pero diseñar software orientado a objetos reutilizable es más difícil todavía. Diseños generales y flexibles son muy difíciles de encontrar la primera vez"

PATRONES DE DISEÑO

- Un patrón es una plantilla que utilizamos para solucionar un problema durante el desarrollo del software con el fin de conseguir que el sistema desarrollado tenga las mismas buenas cualidades que tienen otros sistemas donde anteriormente se ha aplicado la misma solución con éxito.

- Descripciones de clases cuyas instancias colaboran entre sí, que deben ser adaptadas para resolver problemas de diseño generales en un contexto particular.

- Un patrón de diseño identifica: **Clases**, **Instancias**, **Roles**, **Colaboraciones** y la **Distribución de responsabilidades**.

PATRONES DE DISEÑO

- Describen un problema recurrente y una solución.
- Cada patrón nombra, explica, evalúa un diseño recurrente en OO.
- Elementos principales:
  - **Nombre:** permite hacer referencia al patrón cuando documentamos nuestro diseño.
  - **Problema:** indica qué resuelve el patrón.
  - **Solución:** Descripción abstracta.
  - **Consecuencias:** resultados y los compromisos derivados de la aplicación del patrón

Patrones GRASP

- GRASP (General Responsibility Asignment Software Patterns): Este tipo de patrones resuelve problemas durante una tarea muy concreta del diseño orientado a objetos: la asignación de responsabilidades a las clases software.

- Resuelven problemas comunes a cualquier asignación de responsabilidades. Por lo tanto, estos patrones, en realidad, consisten en pautas que nos ayuden a hacer una asignación de responsabilidades siguiendo los principios de diseño más básicos.
- La aplicación de patrones GRASP acostumbra a ser previa a la aplicación del resto de patrones de diseño.

Patrones Grasp

PATRON EXPERTO

- El **EXPERTO** es una clase que tiene toda la información necesaria para implementar una responsabilidad.

Patrones Grasp

PATRON EXPERTO

- Problema: ¿Cuál es el principio fundamental en virtud del cual se asignan las responsabilidades en el diseño orientado a objetos?

- Contexto: Se tiene que hacer un tratamiento o cálculo sobre una información.

- Solución: Asignar una responsabilidad al experto en información: la clase que cuenta con la información necesaria para cumplir la responsabilidad.

Patrones Grasp

PATRON EXPERTO

- En la aplicación del punto de venta, alguna clase necesita conocer el gran total de la venta. Se plantea la pregunta:

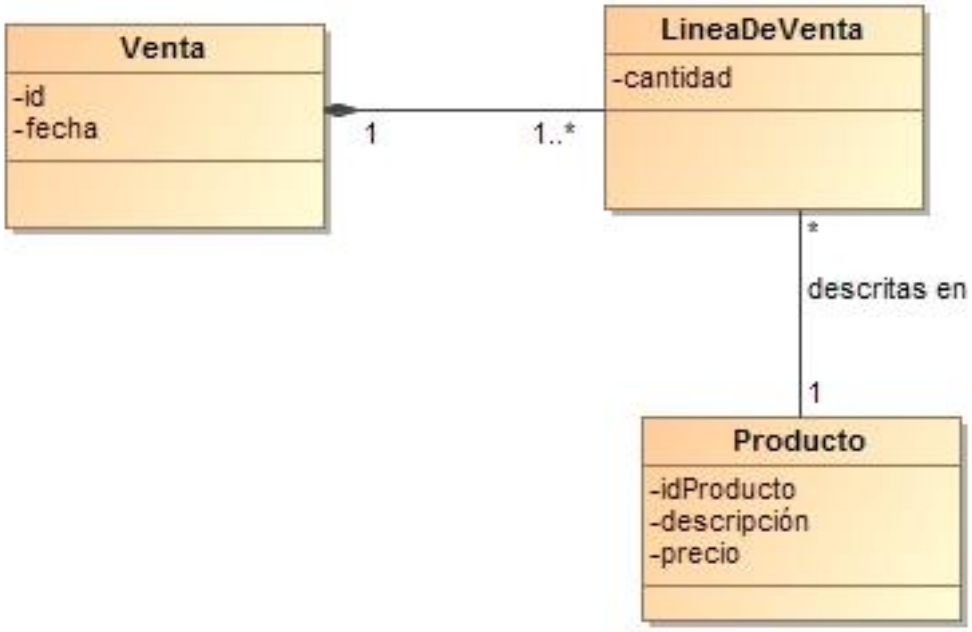
¿Quién es el responsable de conocer el gran total de la venta?

Desde el punto de vista del patrón Experto, deberíamos buscar la clase que posee la información necesaria para calcular el total.

Patrones Grasp

PATRON EXPERTO

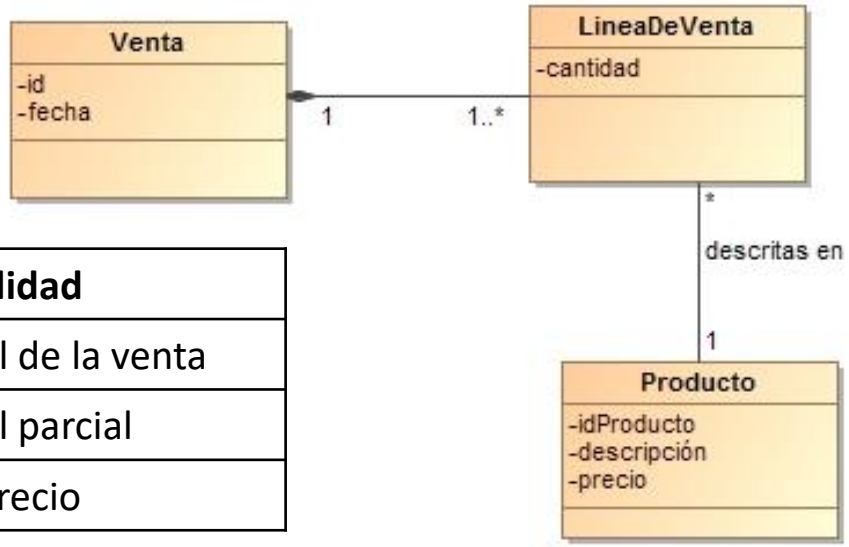
El diagrama de clases que representa la situación descrita es:



Patrones Grasp

PATRON EXPERTO

- Para cumplir con la responsabilidad de conocer y dar el total de la venta, se asignaron tres responsabilidades a las tres clases de objeto como se muestra en la tabla:

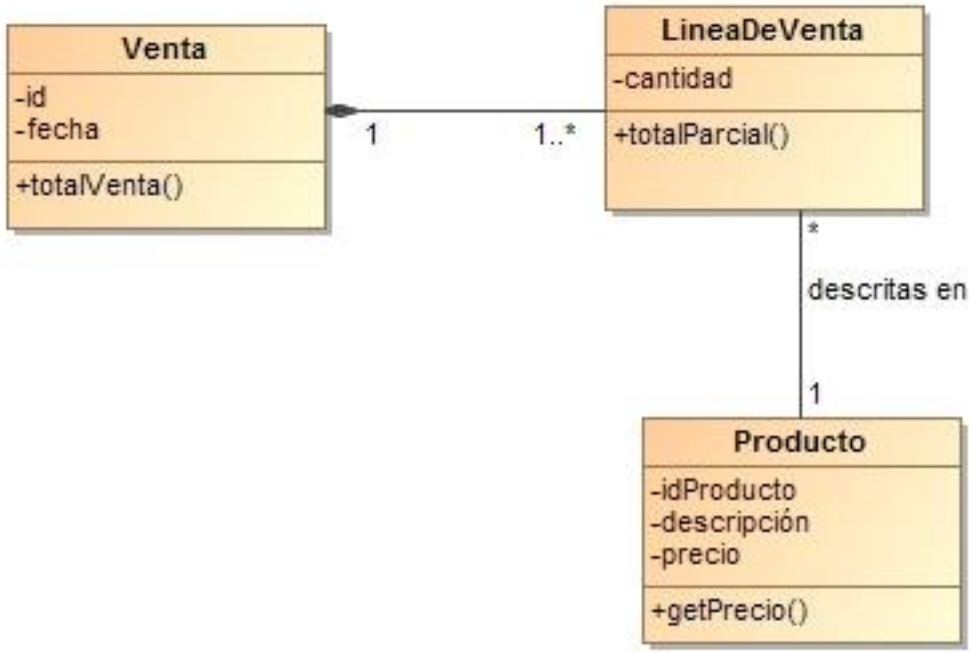


Clase	Responsabilidad
Venta	conoce total de la venta
LineaDeVenta	conoce total parcial
Producto	Conoce el precio

Patrones Grasp

PATRON EXPERTO

- Representación de la solución:



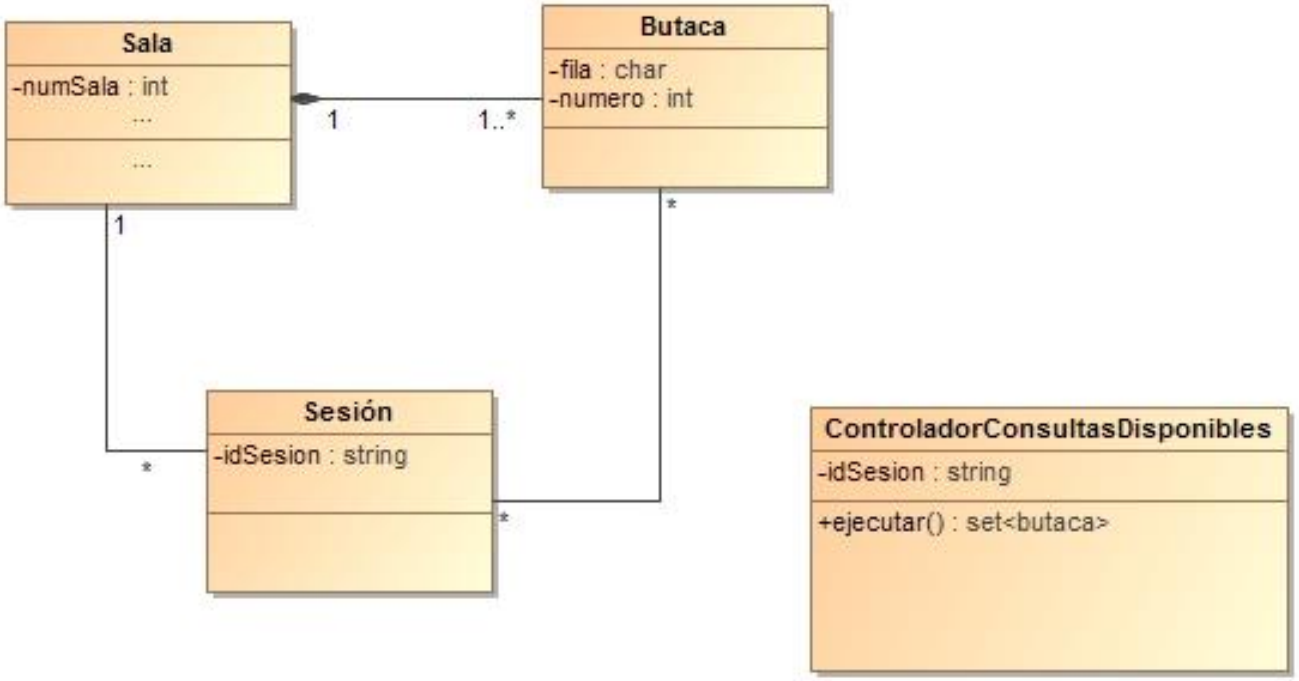
Patrones Grasp

PATRON EXPERTO

- Consecuencias:
  - Mantiene el encapsulamiento de información.
  - Favorece el bajo acoplamiento.
  - Puede originar clases excesivamente complejas.

Patrones Grasp

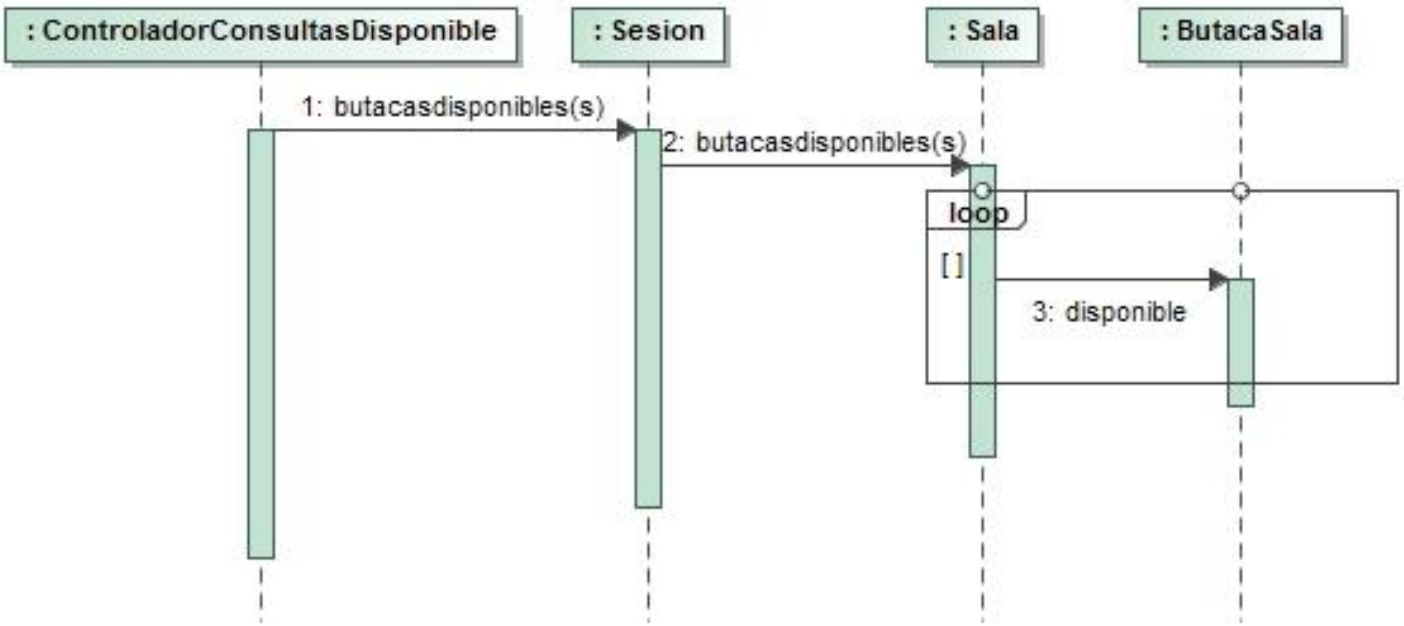
- **Ejercicio:** Supongamos que estamos desarrollando un sistema de reserva de butacas para un cine. Queremos diseñar el tratamiento relativo a la consulta de la lista de butacas disponibles por sala para una sesión:



Patrones Grasp

PATRON EXPERTO

- Representación de la solución:



Patrones Grasp

PATRON CREADOR

- Problema: ¿Quién debería ser responsable de crear una nueva instancia de alguna clase?

- Contexto: Se tienen que crear instancias de una clase.

Patrones Grasp

PATRON CREADOR

- **Solución**: Asigne a la clase B la responsabilidad de crear una instancia de la clase A si se da alguna de los casos siguientes:

    1) **B** agrega a **A**

    2) **B** tiene información necesaria para realizar la creación de **A**

    3) **B** registra a **A**

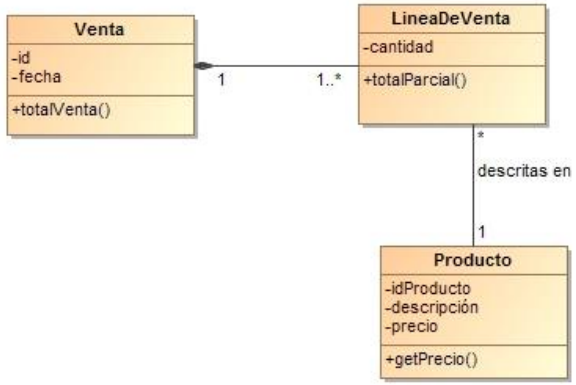
    4) **B** utiliza 'estrechamente' las instancias creadas de **A**

- Si existe más de una opción, prefiera la clase B que agregue o contenga la clase A.

Patrones Grasp

PATRON CREADOR

- En la aplicación del punto de venta:

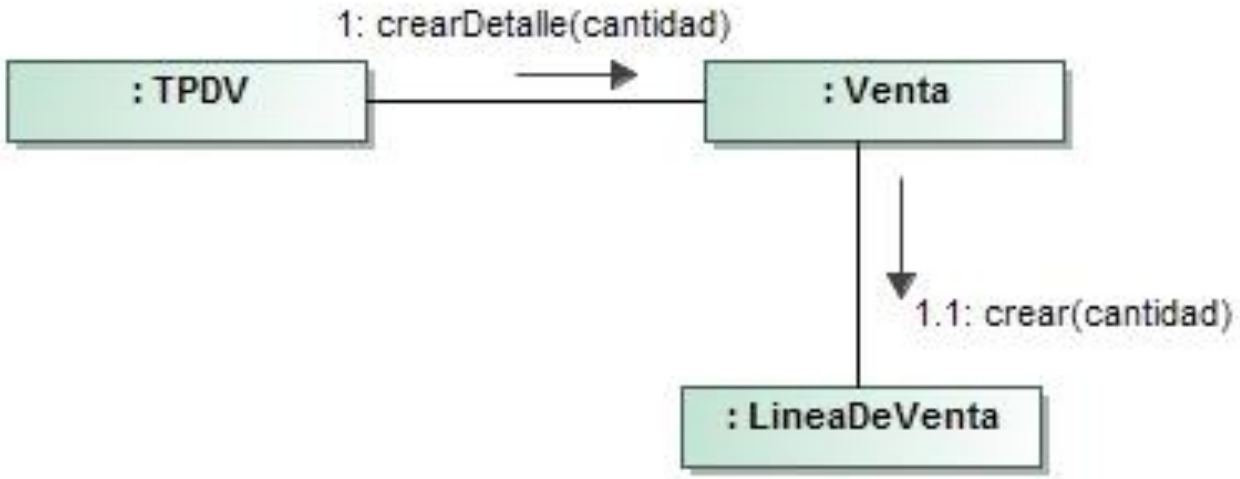


¿Quién debería ser responsable de la creación de una instancia de LineadeVenta?

Patrones Grasp

PATRON CREADOR

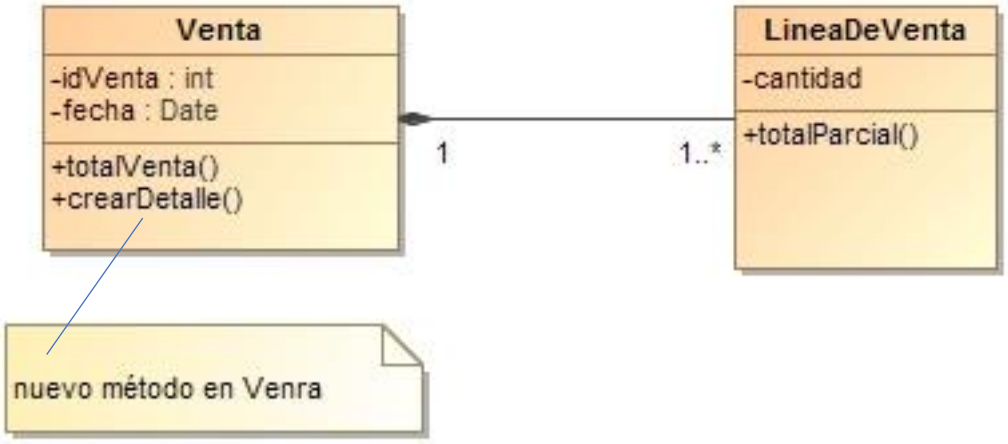
- Representación de la solución:



Patrones Grasp

PATRON CREADOR

- Representación de la solución:



Patrones Grasp

PATRON CREADOR

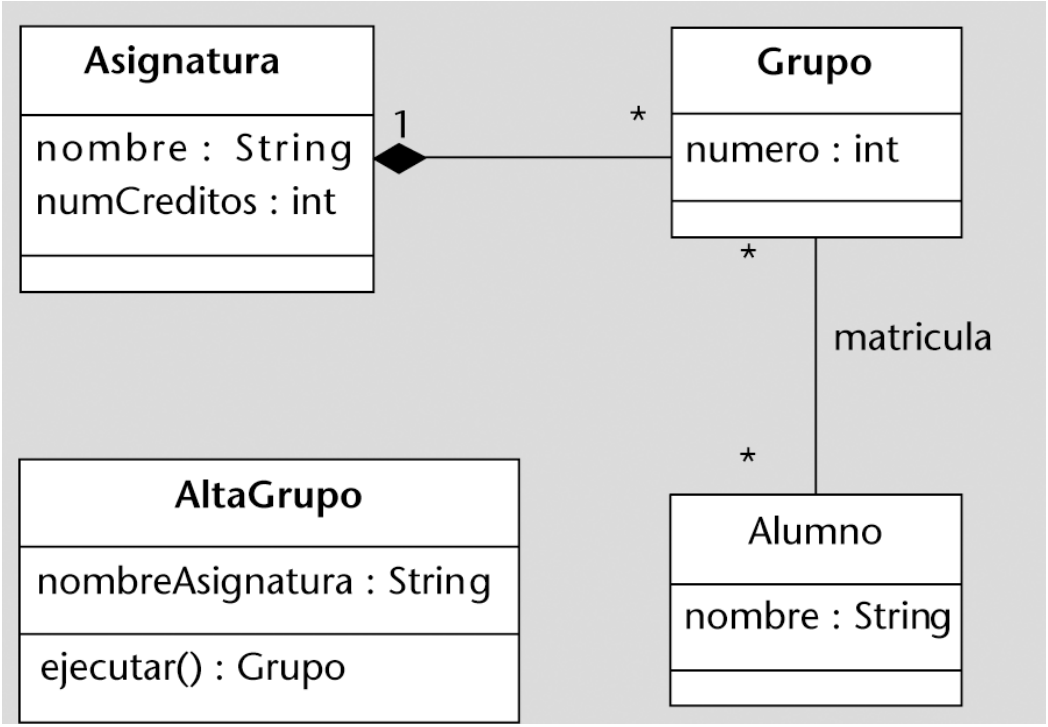
Discusión:

- La intención básica es encontrar un creador que necesite dialogar con el objeto creado en algún momento.

- Se puede ver como un caso particular del Experto (cuando B tiene los datos de inicialización de A).

Patrones Grasp

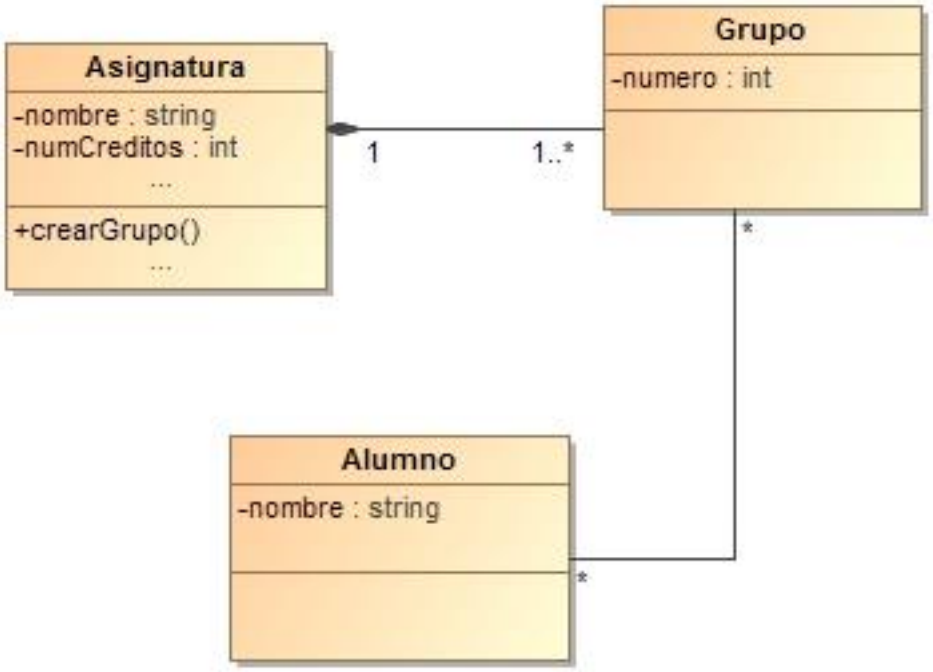
- **Ejercicio:** ¿Quién será el responsable de la creación de un nuevo grupo de una asignatura? :



Patrones Grasp

PATRON CREADOR

- Representación de la solución:



Patrones Grasp

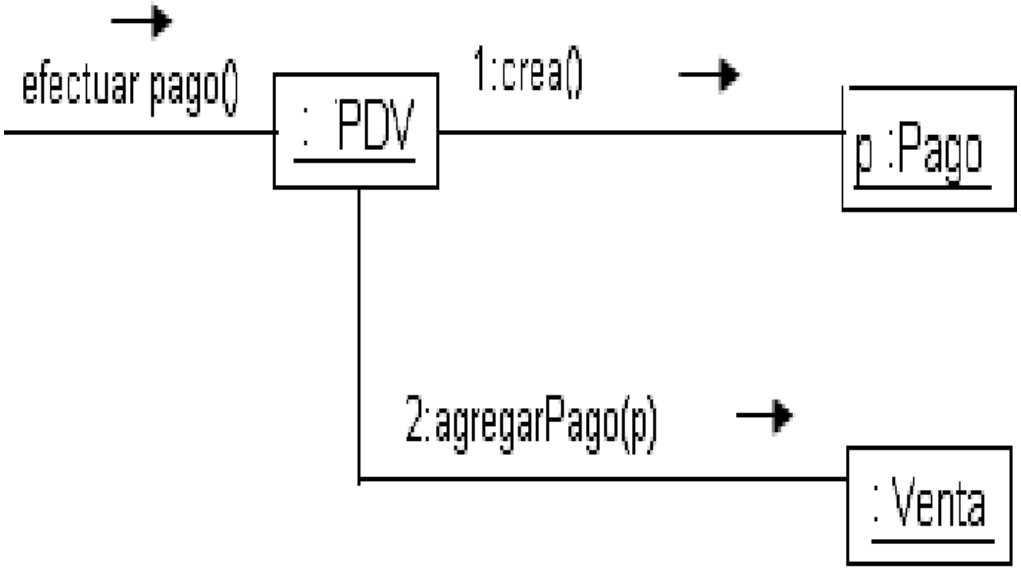
PATRON BAJO ACOPLAMIENTO

- Problema: ¿Cómo dar soporte a una dependencia escasa y a un aumento de la reutilización?
- Contexto: Mantener el bajo acoplamiento en las soluciones de diseño.
- Solución: Asigne responsabilidades de manera que el acoplamiento permanezca bajo.

Patrones Grasp

PATRON BAJO ACOPLAMIENTO

- Ejemplo: En el caso del punto de ventas se tienen tres clases Pago, TPDV y Venta y se quiere crear una instancia de Pago y asociarla a Venta. ¿Que clase es la responsable de realizarlo?

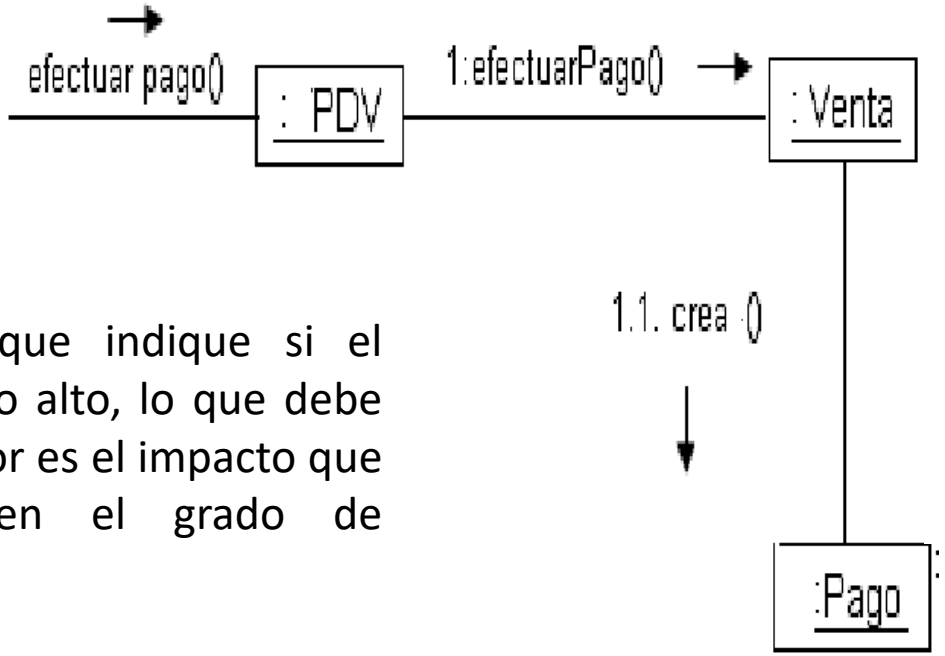


De acuerdo a los patrones anteriores

Patrones Grasp

PATRON BAJO ACOPLAMIENTO

- Solución bajo acoplamiento



Obs: No existe medida que indique si el acoplamiento es demasiado alto, lo que debe tener en cuenta el diseñador es el impacto que provoca una decisión en el grado de acoplamiento.

Patrones Grasp

PATRON ALTA COHESION

- Problema: ¿Cómo mantener manejable la complejidad?
- Contexto: Mantener alta cohesión en las soluciones de diseño.
- Solución: Asigne responsabilidades de manera que sea alta la cohesión.

Patrones Grasp

PATRON ALTA COHESION

- Alta Cohesión. Una clase tiene responsabilidades moderadas en un área funcional y colabora con otras para realizar las tareas.

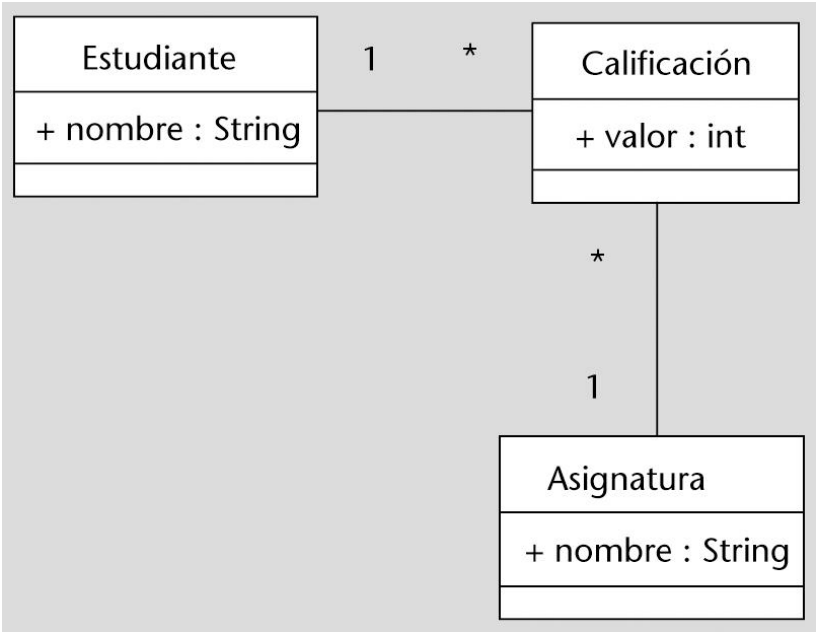
- Cohesión moderada. Una clase tiene responsabilidades exclusivas que están relacionadas lógicamente con el concepto de clase, pero no entre ellas.

- Baja Cohesión. Una clase con baja cohesión hace muchas cosas, hace demasiado trabajo: Clases difíciles de entender; Difíciles de reutilizar; Difíciles de mantener; Delicadas, afectadas por los cambios.

Ejercicios

- supongamos que el sistema que estamos desarrollando tiene que calcular la media y la desviación estándar de las calificaciones de un estudiante, y también las de una asignatura a partir del modelo conceptual siguiente:

¿qué clase es la responsable de realizar estos cálculos?



Patrones de Arquitectura

- son aquellos que se aplican en la definición de la arquitectura de software y que, por lo tanto, resuelven problemas que afectarán al conjunto del diseño del sistema.
- Tres patrones representativos de esta categoría son:

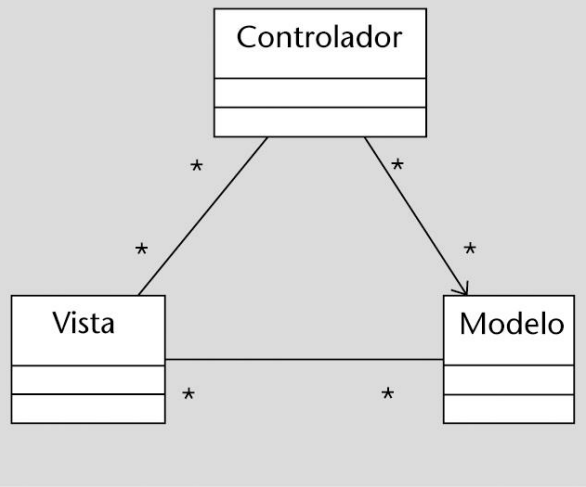
  - Arquitectura en Capas
  - Inyección de dependencias
  - Modelo Vista Controlador

  Estudiaremos el Modelo Vista Controlador

Patrones de Arquitectura

MVC (MODEL VIEW CONTROLLER)

Utilizado para construir interfaces de usuario en Smalltalk-80.



- Contexto : sistema con interfaz gráfica de usuario.
- Problema: Se desea desacoplar la interfaz gráfica del resto del sistema. En una arquitectura en capas, queremos definir la arquitectura de la capa de presentación.
- Solución: Dividir el sistema en tres tipos de componentes: modelos, vistas y controladores.

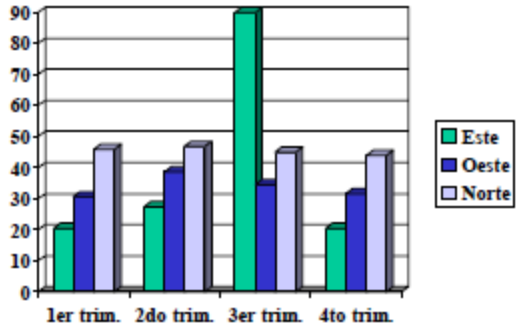
Patrones de Arquitectura

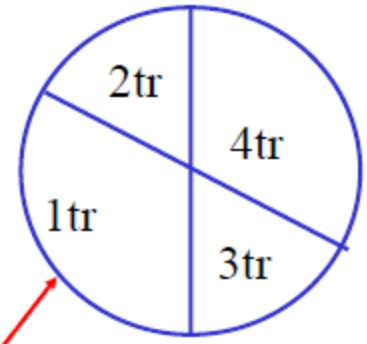
MVC (MODEL VIEW CONTROLLER)

- MODELOS : objetos del dominio
- VISTAS : objetos presentación en pantalla (interfaz de usuario)
- CONTROLADORES : define la forma en que la interfaz reacciona a la entrada del usuario. Notifican a las vistas de los cambios de estado del sistema.

Patrones de Arquitectura

MVC (MODEL VIEW CONTROLLER)





Vistas

Modelo

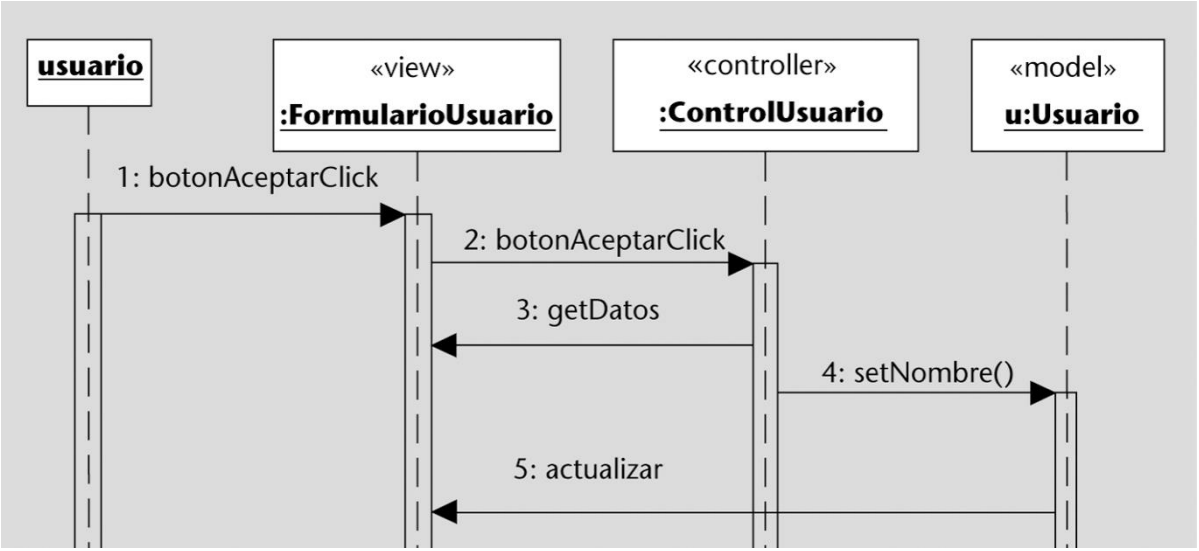
t1=20
t2=25
t3=90
t4=20

	1tr.	2tr.	3tr.	4tr.
Este	20	25	90	20
Oeste	30	38	32	32
Norte	47	47	45	45

Patrones de Arquitectura

MVC (MODEL VIEW CONTROLLER)

Ejemplo, cuando el usuario pulse el botón Aceptar, la vista llamará al controlador; éste traducirá el acontecimiento a una operación de sistema sobre el modelo. Como resultado del cambio en su estado, el usuario notificará a las vistas que se tengan que actualizar.



Patrones GOF

GOF (Gang Of Four) : Son una familia de patrones de diseño que se aplican para resolver problemas concretos de diseño que no afectarán al conjunto de la arquitectura del sistema. Se dividen en tres subcategorías:

- Patrones creacionales : Iniciación y configuración de objetos.
- Patrones estructurales : Separan la interfaz de la implementación. Se ocupan de cómo las clases y objetos se agrupan, para formar estructuras más grandes.
- Patrones de comportamiento : Describen la comunicación entre objetos y clases.

Patrones GOF

	Propósito		
Ámbito	Creación	Estructural	Comportamiento
Clase	Factory Method	Adapter	Interpreter Template Method
Objeto	Abstract Factory Builder Prototype singleton	Adapter Bridge Composite Decorator Fachade Flyweight Proxy	Chain of responsability Command Iterator Mediador Memento Observer State Strategy Visitor

Patrones de Creación

- Abstract Factory: proporciona una interfaz para crear familias de objetos o que dependen entre sí, sin especificar sus clases concretas.

- Builder: separa la construcción de un objeto complejo de su representación, de forma que el mismo proceso de construcción pueda crear diferentes representaciones.

- Factory Method: define una interfaz para crear un objeto, pero deja que sean las subclases quienes decidan qué clase instanciar. Permite que una clase delegue en sus subclases la creación de objetos.

Patrones de Creación

- Prototype: especifica los tipos de objetos a crear por medio de una instancia prototípica, y origina nuevos objetos copiando este prototipo.

- Singleton: garantiza que una clase solo tenga una instancia, y proporciona un punto de acceso global a ella.

Patrones estructurales

- Adapter: convierte la interfaz de una clase en una distinta, que es la esperada por los clientes. Permitiendo que cooperen clases que de otra manera no podrían por tener interfaces incompatibles.

- Bridge : desvincula una abstracción de su implementación, de manera que ambas puedan variar de forma independiente.

- Composite: combina objetos en estructuras de árbol para representar jerarquías de parte-todo. Permite que los clientes traten de manera uniforme a los objetos individuales y a los compuestos.

Patrones estructurales

- Decorator: añade dinámicamente nuevas responsabilidades a un objeto, proporcionando una alternativa flexible a la herencia para extender la funcionalidad.

- Flyweight: usa el compartimiento para permitir un gran número de objetos de grano fino de forma eficiente.

- Facade: proporciona una interfaz unificada para un conjunto de interfaces de un subsistema. Define una interfaz de alto nivel que hace que el subsistema sea más fácil de usar.

- Proxy: proporciona un sustituto o representante de otro objeto para controlar el acceso a este.

Patrones de comportamiento

- Command: encapsula una petición en un objeto, permitiendo así parametrizar a los clientes con distintas peticiones, encolar o llevar un registro de las peticiones y poder deshacer las operaciones.

- Chain of Responsibility: evita acoplar el emisor de una petición a su receptor, al dar a más de un objeto la posibilidad de responder a la petición. Crea una cadena con los objetos receptores y pasa la petición a través de la cadena hasta que esta sea tratada por algún objeto.

- Interpreter: dado un lenguaje, define una representación de su gramática junto con un intérprete que usa dicha representación para interpretar las sentencias del lenguaje.

Patrones de comportamiento

- Iterator: proporciona un modo de acceder secuencialmente a los elementos de un objeto agregado sin exponer su representación interna.

- Mediator: define un objeto que encapsula cómo interactúan un conjunto de objetos. Promueve un bajo acoplamiento al evitar que los objetos se refieran unos a otros explícitamente, y permite variar la interacción entre ellos de forma independiente.

- Memento: representa y externaliza el estado interno de un objeto sin violar la encapsulación, de forma que éste puede volver a dicho estado más tarde.

Patrones de comportamiento

- Observer: define una dependencia entre objetos, de uno-a-muchos, de forma que cuando un objeto cambia de estado se notifica y actualizan automáticamente todos los objetos.

- State: permite que un objeto modifique su comportamiento cada vez que cambia su estado interno. Parecerá que cambia la clase del objeto.

- Strategy: define una familia de algoritmos, encapsula uno de ellos y los hace intercambiables. Permite que un algoritmo varíe independientemente de los clientes que lo usan.

Patrones de comportamiento

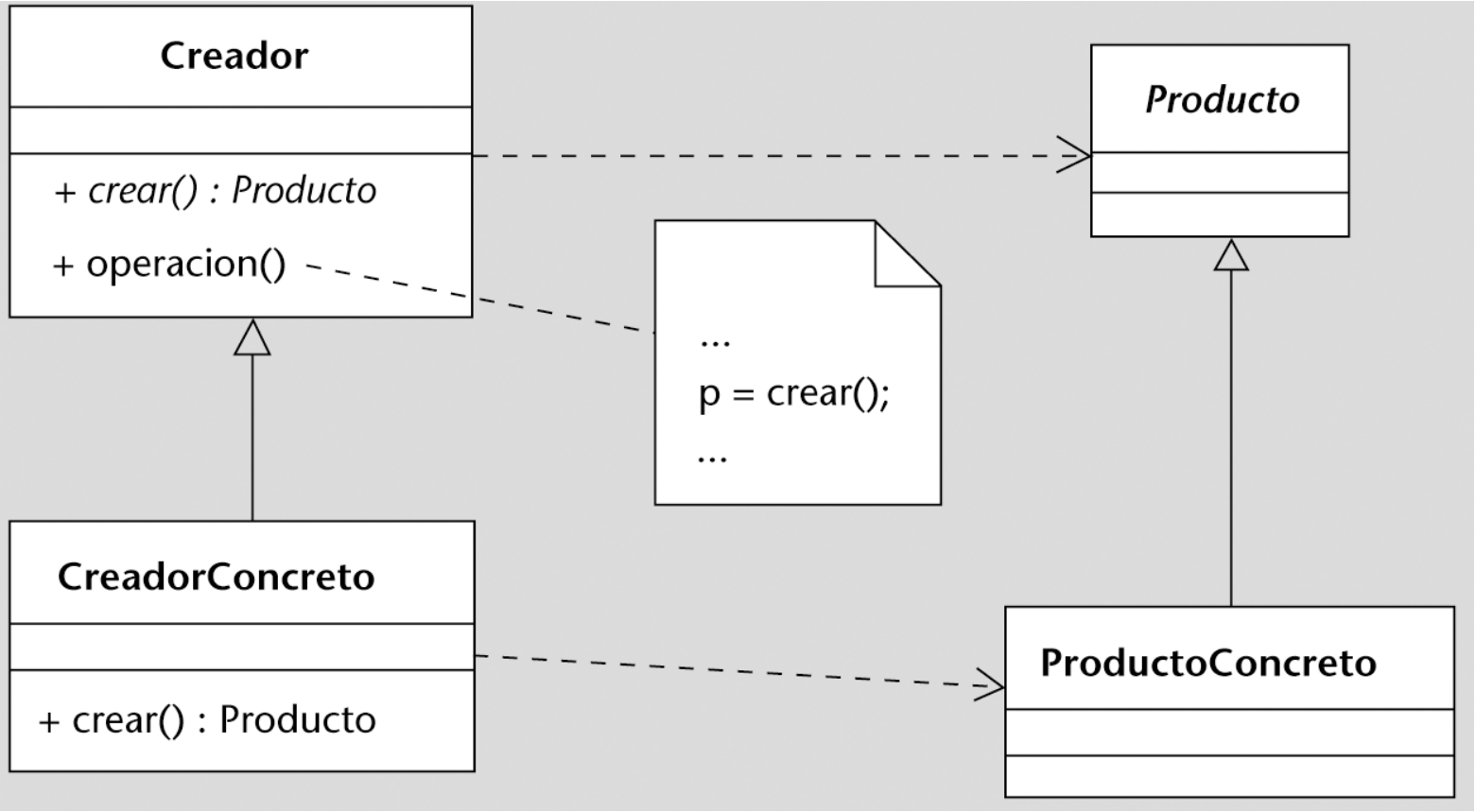
- Template Method: define en una operación el esqueleto de un algoritmo, delegando en las subclases algunos de sus pasos. Permite que las subclases redefinan ciertos pasos del algoritmo sin cambiar su estructura.

- Visitor: representa una operación sobre los elementos de una estructura de objetos. Permite definir una nueva operación sin cambiar las clases de los elementos sobre los que opera.

Método Factoría (Factory method)

- Contexto Un objeto tiene que crear objetos de otra clase.
- Problema Crear diferentes familias de objetos, como por ejemplo la creación de interfaces gráficas de distintos tipos (ventana, menú, botón, etc.). Se quiere delegar la responsabilidad de especificar la clase de los objetos que se tienen que crear en las subclases.
- Solución Definir, en la clase abstracta Creador (la que crea los objetos), un método "crear" encargado de la creación y que denominaremos método factoría. Proporcionar, a cada subclase de Creador, una implementación del método factoría que cree un tipo concreto de objeto.

Método Factoría (Factory method)

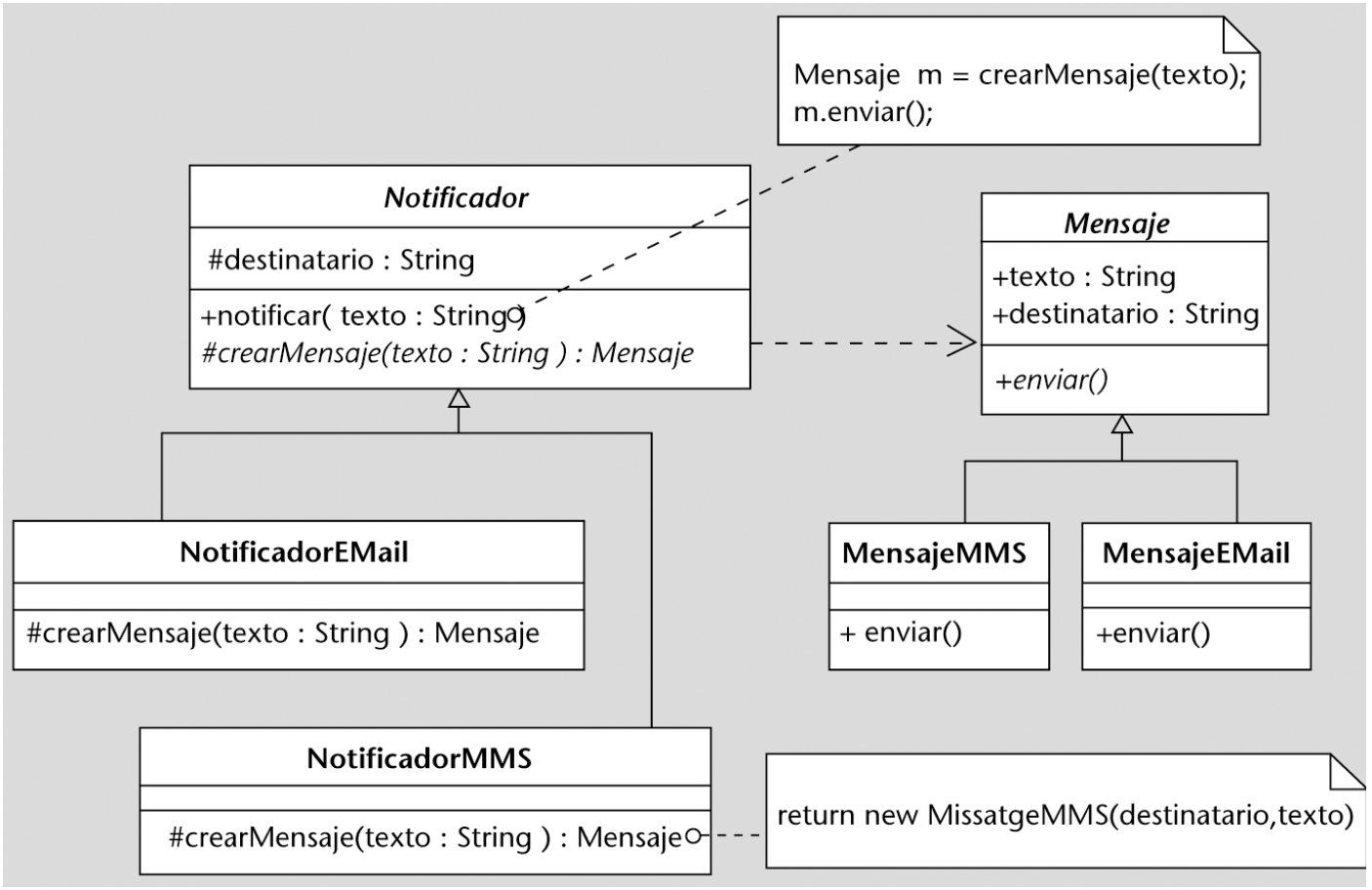


Método Factoría (Factory method)

Ejemplo de Uso:

- Suponga se está desarrollando un sistema que tiene que notificar ciertos acontecimientos a los usuarios. La clase encargada de gestionar las notificaciones tiene que crear un mensaje y enviarlo, pero se quiere delegar en las subclases la selección del tipo de mensaje a enviar (por ejemplo, de correo electrónico o un mensaje MMS a un teléfono móvil).
- Se aplica el método factoría haciendo que la decisión del tipo de mensaje que se creará sea de la subclase. De esta forma, los usuarios que trabajen con un notificador de tipo NotificadorEMail recibirán los mensajes por correo electrónico y los que utilicen un NotificadorMMS los recibirán en su teléfono móvil.eto.

Método Factoría (Factory method)



Decorador (DECORATOR)

También conocido como wrapper

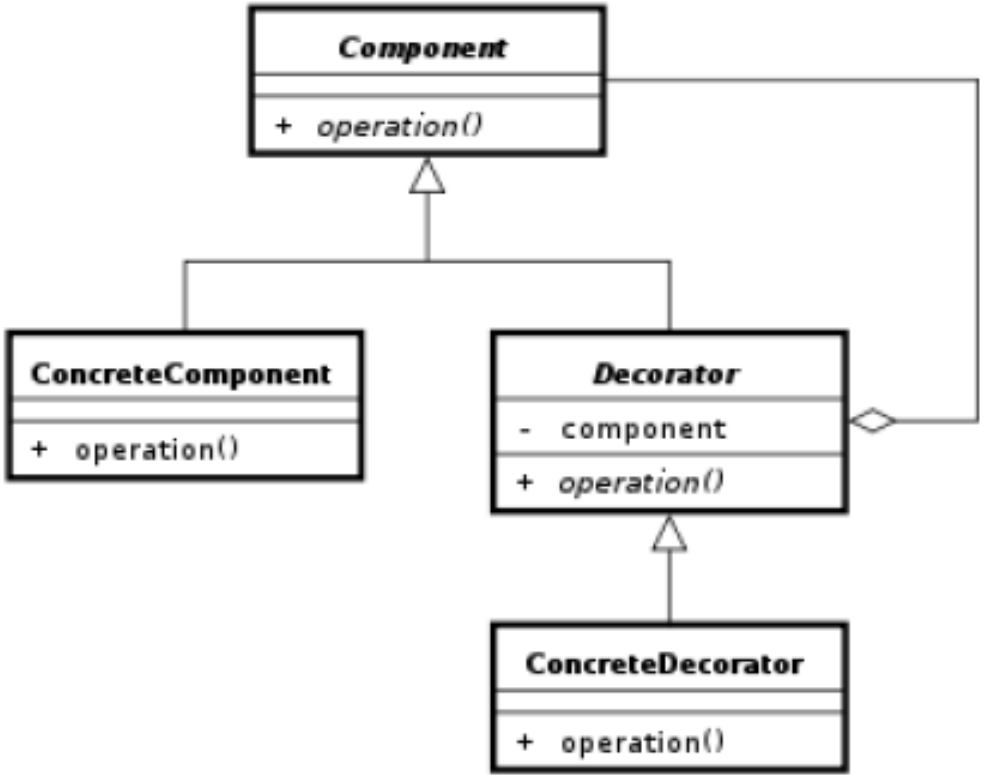
- Contexto Para añadir responsabilidades a objetos concretos de manera dinámica y transparente, esto es, sin afectar a otros objetos. Para responsabilidades que se pueden añadir y quitar o cuando la herencia sea impracticable, porque implique crear múltiples subclases para todas las combinaciones posibles.

Decorador (DECORATOR)

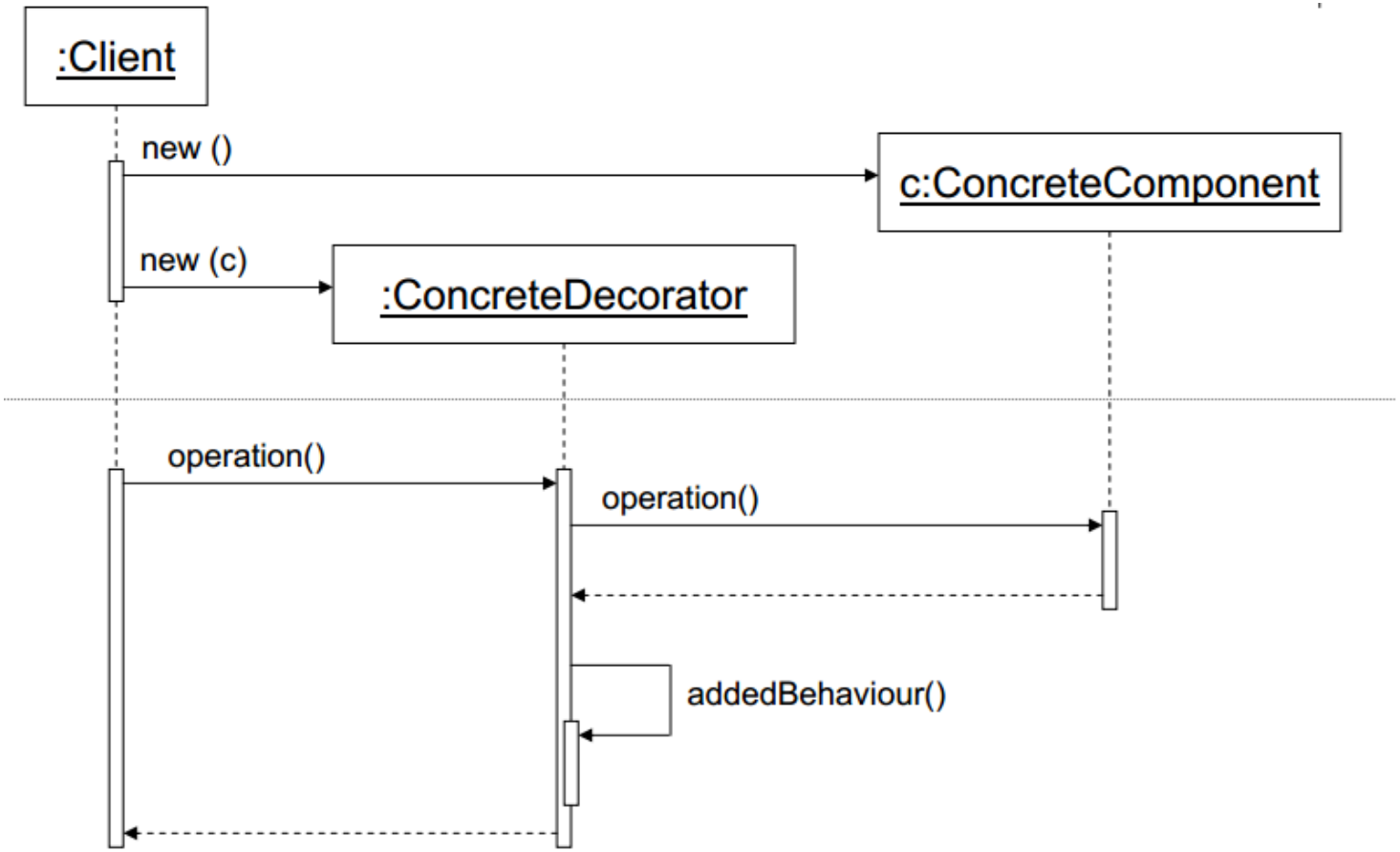
- Problema A veces se quiere añadir funcionalidad a un objeto concreto, no a una clase entera. Permite añadir responsabilidades extras a objetos concretos de manera dinámica.

- Solución Definir una clase “decoradora” que envuelve al componente, y le proporciona la funcionalidad adicional requerida: más flexible, transparente al cliente.

Decorador (DECORATOR)



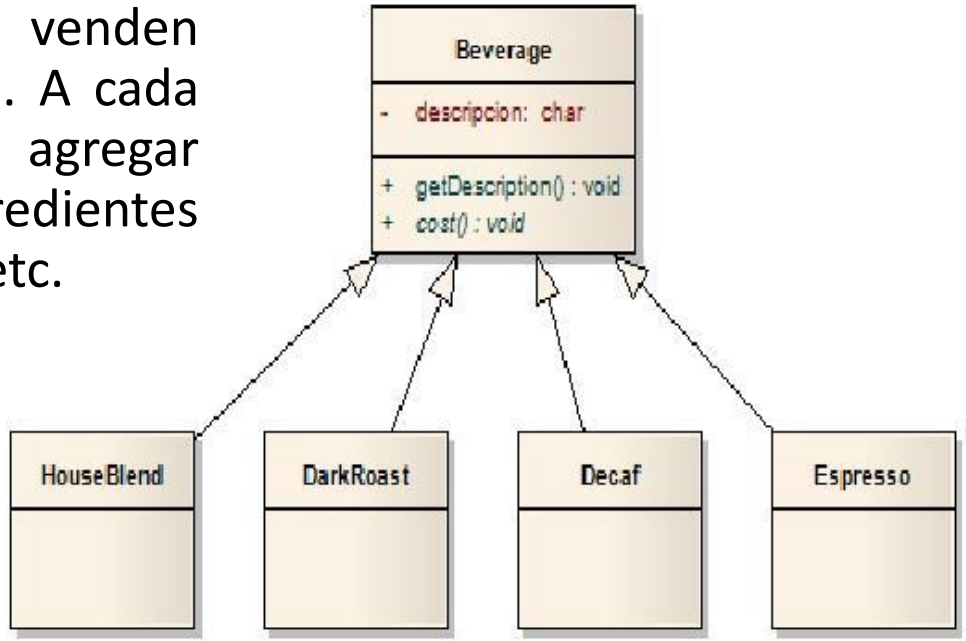
Decorador (DECORATOR)



Decorador (DECORATOR)

ejemplo de uso

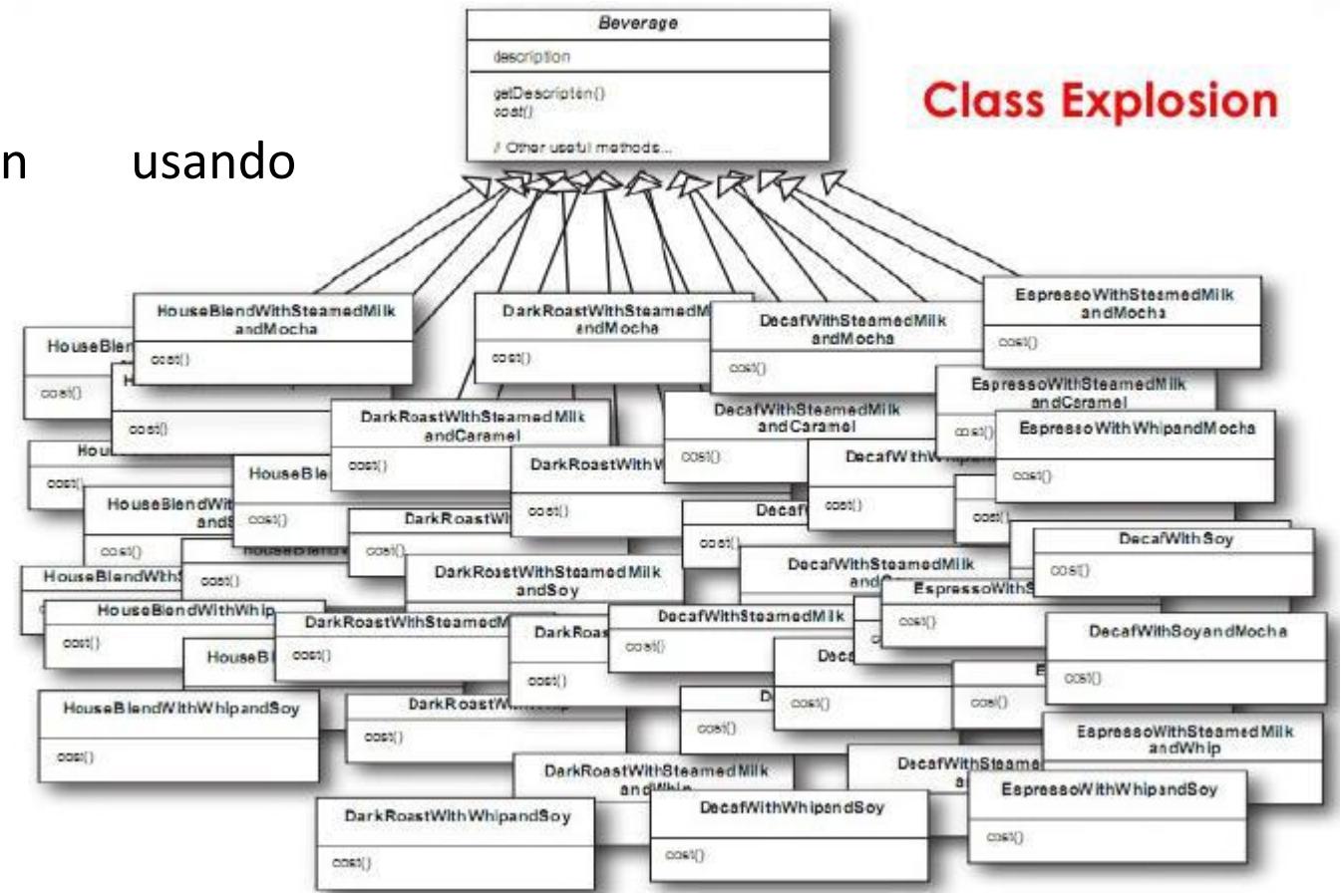
- En un café tipo starbucks, se venden diferentes tipos de bebestibles. A cada uno de ellos se le puede agregar diferentes salsas u otros ingredientes como leche, crema, chocolate, etc.



Decorador (DECORATOR)

ejemplo de uso

- Una solución usando herencia



Decorador (DECORATOR)

```
Public interfaces Vendible {
   public String getDescripcion();
   public int getPrecio(); }

Public abstract class beberage implements Vendible { }

Public class extends beberage {
      public String getDescripcion () {
            return “Café espresso”; }
      public int getPrecio () {
            return 2500; }
}
```

Decorador (DECORATOR)

Usando patrón Decorator:

```
Public abstract class BeberageDecorator implements Vendible {
  public beberageDecorator (Vendible vendible) {
            set Vendible(vendible); }
  public Vendible getVendible {
            return vendible; }
}
Public class SalsaDeChocolate extends BeberageDecorator {
   public SalsaDeChocolate (Vendible vendible) {
          super (vendible) ;
          public String getDescripcion()
          return getVendible().getDescripción() + "+ salsa de chocolate";
}
public int getPrecio()
     return getVendible().getPrecio() + 600 } }
```