https://www.seas.es/blog/?s=uml

https://www.seas.es/blog/informatica/tipos-de-relaciones-en-diagramas-de-casos-de-uso-uml/

Tipos de relaciones en diagramas de casos de uso. UML.

[18 MARZO, 2013](https://www.seas.es/blog/informatica/tipos-de-relaciones-en-diagramas-de-casos-de-uso-uml/)



Hace unas semanas hablamos en este blog de una duda que frecuentemente me plantean los alumnos a la hora de modelar diagramas de clases con [UML](https://www.seas.es/niveles/cursos/uml?piloto=TXW59): *“*[*Agregación Vs Composición en diagramas de clases. UML*](https://www.seas.es/blog/informatica/agregacion-vs-composicion-en-diagramas-de-clases-uml/)*”* , en esta ocasión vamos a intentar resolver otra consulta que me suelen plantear a la hora de modelar diagramas de casos de uso, se trata de los diferentes tipos de **relaciones** que podemos encontrar entre**casos de uso**.

Antes de nada diremos que **los casos de uso** fueron ideados por Jacobson a principios de los noventa y están inspirados en el concepto de escenario, el cual ya había sido utilizado para describir procesos. Los casos de uso especifican un comportamiento deseado del sistema, representan **requisitos funcionales** del mismo. Es importante resaltar que describen qué hace el sistema, no cómo lo hace. UML nos dice que: “Un caso de uso especifica un conjunto de secuencias de acciones, incluyendo variantes, que el sistema puede ejecutar y que produce un resultado observable de valor para un particular actor”. Los casos de uso nos sirven de base para elaborar los aspectos funcionales del pliego de condiciones y nos dan soporte en las etapas de modelado, desarrollo y validación de software.

Elementos de un caso de uso:

* **Conjunto de secuencias de acciones,** cada secuencia representa un posible comportamiento del sistema.
* **Actores,** se tratan de los roles que pueden jugar los agentes que interactúan con el sistema. Los roles son jugados por personas, dispositivos, u otros sistemas. Podríamos distinguir entre actores primarios, para los cuales el objetivo del caso de uso es esencial y actores secundarios, que interactúan con el caso de uso, pero cuyo objetivo no es esencial.
* **Variantes,**son versiones especializadas, un caso de uso que extiende a otro o un caso de uso que incluye a otro

Como veremos a continuación, en los diagramas de casos de uso se muestran: casos de uso (representados en forma de elipses), actores (en forma de personajes) y  relaciones (en forma de líneas y/o flechas). UML define cuatro tipos de **relaciones en los diagramas de casos de uso**:

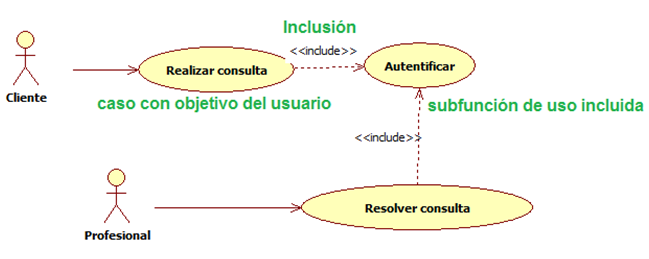
* **Comunicación**: Relación (asociación) entre un actor y un caso de uso. El estereotipo de la relación de comunicación es: <<communicate>> aunque generalmente no se estipula ningún nombre, como podemos apreciar en el siguiente *ejemplo de comunicación*:



* **Inclusión**: Un caso de uso base incorpora explícitamente el comportamiento de otro en algún lugar de su secuencia. La relación de inclusión sirve para enriquecer un caso de uso con otro y compartir una funcionalidad común entre varios casos de uso, también puede utilizarse para estructurar un caso de uso describiendo sus subfunciones. El caso de uso incluido existe únicamente con ese propósito, ya que no responde a un objetivo de un actor.

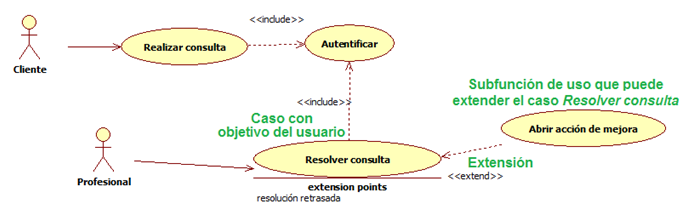
Estas relaciones se representan mediante una flecha discontinua con el estereotipo <<include>>. Algunos casos de uso típicos de inclusión son: comprobar, verificar, buscar, validar, autentificar o login… En principio, no deberíamos abusar de este tipo de relación, para no hacer una descomposición funcional del sistema. A partir de UML 1.3 la relación <<include>> reemplazó al denominado <<uses>>.

Veamos un *ejemplo de inclusión* entre casos de uso:



* **Extensión**: Un caso de uso base incorpora implícitamente el comportamiento de otro caso de uso en el lugar especificado indirectamente por este otro caso de uso. En el caso de uso base, la extensión se hace en una serie de puntos concretos y previstos en el momento del diseño, llamados **puntos de extensión**, los cuáles no son parte del flujo principal. La relación de extensión sirve para modelar: la parte opcional del sistema, un subflujo que sólo se ejecuta bajo ciertas condiciones o varios flujos que se pueden insertar en un punto determinado. Este tipo de relación produce confusión y no debería utilizarse en exceso. Conviene su uso sólo para insertar un nuevo comportamiento no previsto en un caso de uso existente. Estas relaciones se representan mediante una flecha discontinua con el estereotipo <<extend>>.

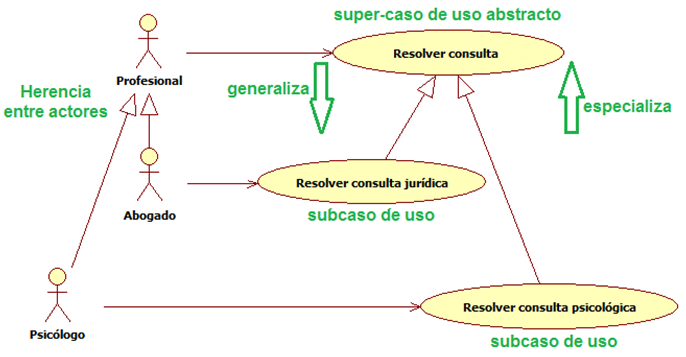
Veamos un *ejemplo de extensión:*



En este ejemplo usamos la relación de extensión entre los casos de uso *Abrir acción de mejora* y *Resolver consulta.*En este caso tendremos el punto de extensión “resolución retrasada” (en el caso de uso *Resolver consulta*) debido a que cuando haya pasado un tiempo estipulado por la organización (por ejemplo 3 días laborales) se abrirá una acción de mejora para dejar constancia del retraso y realizar posteriormente las acciones pertinentes, de ahí que digamos que el caso de uso *Abrir acción de mejora*es una subfunción de uso que puede extender al caso de uso*Resolver consulta.*

* **Especialización y generalización de los casos de uso**: Un caso de uso (subcaso) hereda el comportamiento y significado de otro, es decir las relaciones de comunicación, inclusión y extensión del super-caso de uso. En muchas ocasiones este super-caso de uso es abstracto y corresponde a un comportamiento parcial completado en el subcaso de uso. O dicho de otra manera, Los casos de uso “hijo” son una especialización del caso de uso “padre”. En la medida de lo posible debería evitarse puesto que produce cierta confusión en algunas ocasiones.

Veamos un *ejemplo de generalización*:



Como podemos ver en este último ejemplo también pueden existir vínculos de generalización o **herencia entre actores**. Los actores especializados (Abogado y Psicólogo) heredan los casos de uso del actor general (Profesional). La punta de flecha apunta al actor más general. Hay que reseñar que los actores especializados pueden tener otros casos de uso propios que no estarán disponibles para los demás actores. Este tipo de herencia entre actores si que se usa frecuentemente puesto que nos simplifica considerablemente el diagrama, nos ahorra un número importante de relaciones de comunicación entre actores y casos de uso y nos sirve para esclarecer visualmente la jerarquía entre actores del sistema.

Observación: la relación de comunicación que se da entre un actor(es) y un caso(s) de uso no se representa mediante una flecha sino mediante un linea simple.

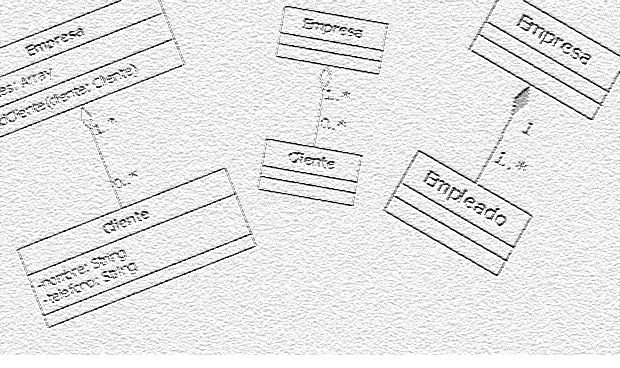
Generalización o especialización se da SOLO entre ACTORES o SOLO entre CASOS DE USO, lo cual es viable. El INCLUDE no me quedó muy claro.

hola alguien tiene estos ejemplos resueltos?  
Complete el siguiente diagrama de estados UML de acuerdo con la especificación textual que  
aparece a continuación, proporcionando el texto correspondiente a la etiqueta de las  
transiciones de máquina de estados t1 a t7 y para las acciones a1 y a2. Se proporciona el  
texto de la transición inicial (indicando, a la vez, la sintaxis de la asignación que debe usar para  
el lenguaje de acciones).  
Su solución debe usar los siguientes eventos de disparo (o señales de activación):  
tirar\_anilla, tocar\_suelo  
las siguientes variables boleanas:  
fallado1, fallado2, abrir1, abrir2  
y las siguientes invocaciones de métodos:  
alucinar(), gritar(String grito), expirar().  
Un paracaidista se tira desde un avión, momento a partir del cual está en caída libre. Entonces  
puede tirar de la anilla para abrir el paracaídas principal (nota: pero no la que abre el  
paracaídas de emergencia). Al hacerlo, pueden pasar dos cosas: el paracaídas se abre, en cuyo  
caso grita “sí” y empieza a caer despacio, o no se abre, en cuyo caso sigue en caída libre. Si no  
se abre el paracaídas principal (nota: y no en otro caso), puede intentar abrir el paracaídas de  
emergencia con los dos mismos posibles resultados. Si se mantiene en caída libre un tiempo  
mayor que max\_fall\_time (nota: independientemente de que haya tratado de abrir cero,  
uno o dos paracaídas), la caída se convierte en una caída mortal y ve pasar delante de él toda  
su vida. En todos los casos, al final llega al suelo, gritando “¡aaah!” en caso de caída con  
paracaídas y expirando en caso de caída mortal.  
caída\_libre  
caída despacio  
entry / a1  
caída mortal  
do / a2  
/ fallado1 := false, fallado2:= false  
abrir1  
t1  
abrir2  
[true] [true]  
[false] [false]  
t2 t3  
t4 t5  
t6 t7Software de Comunicaciones. Ejemplos de UML Departamento de Ingeniería Telemática Universidad Carlos III de Madrid  
Problema del examen de septiembre de 2007  
Se quiere desarrollar un sistema de información para las universidades españolas según la  
descripción siguiente.  
Una Universidad se caracteriza mediante su nombre y la ciudad donde se sitúa. A una  
Universidad están vinculados dos tipos de Persona: Trabajadores, que la Universidad  
emplea, y Estudiantes, que estudian en la Universidad. Cada Persona tiene un DNI y un  
nombre.  
Los Trabajadores pertenecen a dos grupos: PDI y PAS. Cada Trabajador tiene asociada una  
fecha de inicio de su contrato. Cada miembro del PDI también tiene una categoría, mientras  
que cada miembro del PAS tiene un puesto. Los miembros del PDI pueden o no ser Doctores.  
Las actividades que desarrolla el PDI son investigar y enseñar, mientras que la actividad que  
desarrolla el PAS es administrar.  
Cada Universidad se compone de un conjunto de Departamentos, cada uno de los cuales  
tiene un nombre y un conjunto de Trabajadores adscrito. Un Trabajador no puede estar  
adscrito a más de un Departamento. Un PDI está adscrito obligatoriamente a un  
Departamento, mientras que un PAS, no. Cada Departamento está dirigido por un Doctor.  
Un Estudiante puede ser bien Estudiante de grado, de una determinada titulación, bien  
Estudiante de doctorado, de un determinado programa de doctorado. Un Estudiante de  
grado puede también colaborar con un Departamento como becario y puede realizar un PFC  
dirigido por un miembro del PDI. Un Estudiante de doctorado realiza una tesis dirigida por  
un Doctor.  
Puede suponer que un Estudiante no puede estudiar en más de una Universidad y que un  
Trabajador no puede ser empleado por más de una Universidad.  
Proporcione un modelo de esta descripción en forma de un diagrama de clases UML utilizando  
para nombres de clases únicamente las palabras que aparecen en negrita en la descripción  
anterior. Las palabras que aparecen en cursiva proporcionan pistas para la definición de los  
otros elementos del modelo. No hace falta proporcionar información de tipado para las  
propiedades que pueda definir.  
Para más puntuación, añada a su modelo los elementos necesarios para tomar en cuenta lo  
siguiente:  
• una Persona puede ser a la vez Trabajador y Estudiante,  
• un Estudiante no puede ser a la vez Estudiante de grado y Estudiante de doctorado,  
• los únicos tipos de Trabajador que existen son PDI y PAS,  
• un Trabajador no puede ser a la vez PDI y PAS.Software de Comunicaciones. Ejemplos de UML Departamento de Ingeniería Telemática Universidad Carlos III de Madrid  
Problema del examen de septiembre de 2008  
(a) Los diagramas UML de la Figura 1 constituyen un modelo simplificado de un cajero  
conectado a un banco. Constan de un diagrama de clases con dos clases, y dos diagramas  
de estado cada uno de los cuales describe el comportamiento de una de estas clases.  
Estudie los diagramas y responda a continuación a las siguientes preguntas:  
(i) (0,3 puntos) ¿Cuál es el significado de los rectángulos verticales negros del diagrama  
de estados de la clase Bank? ¿Qué comportamiento del banco describe la transición  
que va desde el estado CardValid hacia el rectángulo negro situado más abajo a la  
derecha del diagrama?.  
(ii) (0,2 puntos) ¿Cuál es el significado de los diamantes situados dentro del estado  
Verifying de la clase Bank? ¿Cómo es que no aparece guarda alguna en ninguna de  
las transiciones de salida del diamante situado más abajo?  
(iii) (0,2 puntos) ¿Cuál es el significado de la línea discontinua situada dentro del estado  
Verifying del diagrama de estados de la clase Bank, y cuál es la diferencia entre  
este tipo de estado y un estado como Giving Money del diagrama de estados de la  
clase ATM?  
(iv) (0,6 puntos) Describa brevemente cada una de las partes que pueden aparecer en la  
etiqueta de una transición de un diagrama de estados UML. ¿Cuál es el significado del  
texto que empieza por el carácter “^” situado en las etiquetas de algunas de las  
transiciones de los dos diagramas de estado de la Figura 1? Explique qué representan  
cada una de las dos transiciones del diagrama de estados de la clase ATM cuya etiqueta  
contiene el texto ^bank, y cada una de las tres transiciones del diagrama de estados  
de la clase Bank cuya etiqueta contiene el texto ^atm [Pista: mire el diagrama de  
clases].  
(b) (0,7 puntos) Ahora proporcione un diagrama de secuencias UML que muestre la  
comunicación entre un objeto que desempeñe el rol atm y un objeto que desempeñe el rol  
bank correspondiente. [Pistas: el diagrama debería reflejar si la comunicación es síncrona  
o asíncrona; se sugiere usar operadores de interacción].  
Problema del examen de enero de 2009  
Estudie el diagrama de clases UML que aparece en la Figura 1.  
(i) Describa en lenguaje natural el dominio modelado por esta especificación UML; si no  
tiene tiempo para proporcionar una descripción exhaustiva, al menos ilustre cada uno  
de los distintos elementos sintácticos que en ella aparece.  
(ii) ¿Qué otra información podría haberse proporcionado dentro de las cajas?  
(iii) ¿Qué tiene de particular la caja que contiene el texto “IntervaloTiempo”?«signal» done  
verifyPIN()  
«signal» PINVerified  
«signal» abort  
«signal» reenterPIN  
1  
atm  
1  
bank Bank  
int maxNumIncorrect = 2  
int numIncorrect = 0  
boolean cardValid = true  
ATM  
(a) Class diagram  
ReturningCard  
Verification AmountEntry  
CardEntry  
Counting  
Dispensing  
PINEntry  
Giving Money  
PINVerified  
abort  
/ ^bank.done  
/ ^bank.verifyPIN()  
reenterPIN  
(d) State machine diagram for class ATM  
VerifyingCard CardValid  
Idle  
PINCorrect  
entry / numIncorrect = 0  
PINIncorrect  
VerifyingPIN  
[else] / ^atm.abort  
[cardValid]  
[else] / cardValid = false; ^atm.abort  
/ ^atm.PINVerified  
[numIncorrect < maxNumIncorrect]  
/ numIncorrect++; ^atm.reenterPIN  
done  
Verifying  
verifyPIN()  
(e) State machine diagram for class Bank  
Fig. 1. UML model of an ATM

https://www.seas.es/blog/informatica/agregacion-vs-composicion-en-diagramas-de-clases-uml/

# Agregación Vs Composición en diagramas de clases. UML.

[25 ENERO, 2013](https://www.seas.es/blog/informatica/agregacion-vs-composicion-en-diagramas-de-clases-uml/)

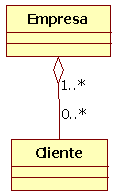


Una duda que frecuentemente me plantean los alumnos a la hora de modelar diagramas de clases con UML (Unified Modeling Language), es el uso de las relaciones estructurales de agregación y composición. Se trata de dos tipos de especialización de la relación de asociación entre clases.  
Vamos a intentar mediante algunos ejemplos muy simples y esclarecedores, ver las diferencias que existen entre la composición fuerte y la composición débil, conocida habitualmente como agregación.

### **Agregación**

La agregación es un tipo de asociación que indica que una clase es parte de otra clase (composición débil). Los componentes pueden ser compartidos por varios compuestos (de la misma asociación de agregación o de varias asociaciones de agregación distintas). La destrucción del compuesto no conlleva la destrucción de los componentes. Habitualmente se da con mayor frecuencia que la composición.  
La agregación se representa en UML mediante un diamante de color blanco colocado en el extremo en el que está la clase que representa el “todo”.

Veamos un ejemplo de agregación:

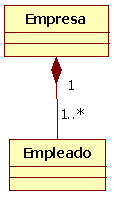


• Tenemos una clase Empresa.  
• Tenemos una clase Cliente.  
• Una empresa agrupa a varios clientes.

### **Composición**

Composición es una forma fuerte de composición donde la vida de la clase contenida debe coincidir con la vida de la clase contenedor. Los componentes constituyen una parte del objeto compuesto. De esta forma, los componentes no pueden ser compartidos por varios objetos compuestos. La supresión del objeto compuesto conlleva la supresión de los componentes.  
El símbolo de composición es un diamante de color negro colocado en el extremo en el que está la clase que representa el “todo” (Compuesto).

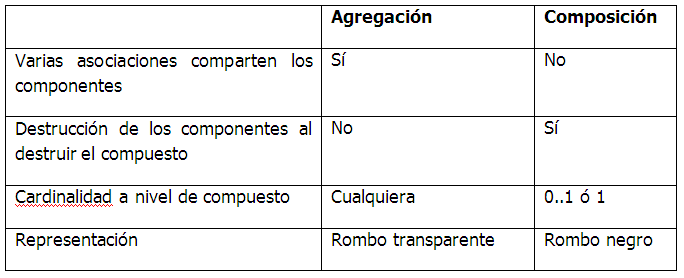
Veamos un ejemplo de composición:



• Tenemos una clase Empresa.  
• Un objeto Empresa está a su vez compuesto por uno o varios objetos del tipo empleado.  
• El tiempo de vida de los objetos Empleado depende del tiempo de vida de Empresa, ya que si no existe una Empresa no pueden existir sus empleados.

### **Diferencias entre Composición y Agregación**

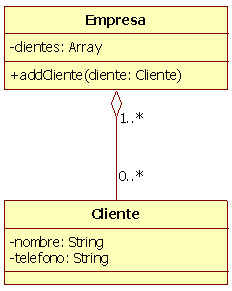
La siguiente tabla intenta resumir algunas diferencias entre agregación y composición.



### **Y en código…**

Para traducir ambas relaciones a código, podemos utilizar un atributo en la clase contenedora o compuesta donde almacenaremos una colección de los objetos que la componen, y por otro lado declararemos un método para agregar elementos a la colección. Dependiendo del lenguaje de programación empleado, podemos utilizar diferentes estructuras de datos que nos permitan almacenar esa colección de objetos, aunque generalmente se utilizan arrays (arreglos) para este fin.

Veamos un ejemplo:



Como podemos apreciar, es tan simple como crear en la clase Empresa un atributo clientes (colección de clientes) que sea un array, luego creamos un método addCliente donde recibiremos objetos de tipo Cliente y los agregaremos dentro del array.

### **Concluyendo…**

En líneas generales, como hemos visto, se podría decir que **la diferencia entre agregación y composición es conceptual**, no se diferencia por código, o al menos, en el mayor de los casos y en la mayoría de los lenguajes de programación (como [Java](https://www.seas.es/niveles/cursos/java?piloto=W61)o [PHP](https://www.seas.es/niveles/cursos/php-mysql?piloto=W61)). De todas maneras, en el caso de la composición, si quisiéramos ser más estrictos con los diagramas de clases modelados con UML, deberíamos destruir de alguna manera el objeto componente (empleado) una vez que se desasociaran del objeto compuesto (empresa).

En definitiva, UML nos permite la posibilidad de diferenciar este tipo de asociaciones con el fin de que, aquella persona que le interese, pueda estipular de una u otra manera que se trata de una composición o una agregación, aunque en términos de implementación no se diferencie tan apenas su uso ni tenga tanta relevancia. Pero una vez más, y como vimos en un post anterior de este blog: “[UML en su justa medida…](https://www.seas.es/blog/informatica/uml-en-su-justa-medida/)” , UML propone muchas posibilidades y debe ser el analista y/o desarrollador quien decida y haga un uso correcto del mismo, con el fin de visualizar, especificar, construir y documentar adecuadamente los artefactos (modelos) de un sistema software.

Todo esto y mucho más, se estudia en el curso de [Experto en gestión y desarrollo de aplicaciones informáticas orientadas a objetos](https://www.seas.es/niveles/expertos/aplicaciones-orientadas-objetos?piloto=W61).

https://www.seas.es/blog/informatica/patrones-de-diseno-en-java-patron-strategy/

Patrones de diseño en Java: Patrón Strategy

[21 ABRIL, 2014](https://www.seas.es/blog/informatica/patrones-de-diseno-en-java-patron-strategy/)



Tras los [patrones Composite](https://www.seas.es/blog/informatica/patrones-de-diseno-en-java-patron-composite/?piloto=TXW59) y[Observer](https://www.seas.es/blog/informatica/patrones-de-diseno-en-java-patron-observer/?piloto=TXW59),  en esta ocasión veremos el patrón Strategy. Este patrón está orientado a establecer unas reglas de comportamiento en las clases u objetos que lo implementen. Por tanto, en la JDK será más habitual encontrarnos con interfaces que lo plantean como es el caso de Comparable.

**Patrón Strategy**

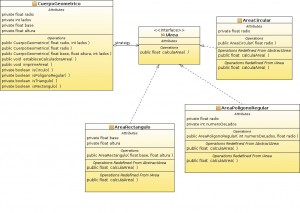
Este patrón facilita la implementación de distintas estrategias o comportamientos específicos en clases hijas a través de una clase común. Así, en tiempo de ejecución y en función de algún parámetro como el tipo de instancia, se ejecutará la estrategia concreta para esa situación.

Se recomendará usar este patrón cuando en un mismo programa debamos proporcionar distintas alternativas de comportamiento, permitiendo a través de clases independientes, encapsular las distintas estrategias.

Los distintos componentes de este patrón son:

* Interfaz Strategy: será aquella interfaz que define el nombre del método o métodos que conformarán la estrategia.
* Clases Strategy concretas: todas aquellas clases que implementen la interfaz Strategy dando forma al algoritmo.
* Contexto: elemento donde se desarrollará la estrategia.

El diagrama UML de la implementación de este patrón será:

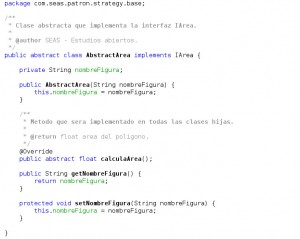
[](https://www.seas.es/blog/wp-content/uploads/2014/04/strategy-diagram.jpg)

Como podemos observar, la interfaz iArea será la estrategia genérica. En este caso, además, se ha añadido una clase abstracta AbstractArea que implementará dicha interfaz y definirá algunas propiedades que serán comunes o necesarias en las implementaciones concretas (en este caso, el nombre de la figura).

* **iArea**: Interfaz en la que definimos un método que nos ayudará a calcular el área de cualquier cuerpo geométrico.

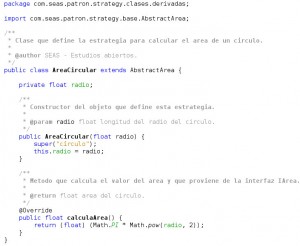
[](https://www.seas.es/blog/wp-content/uploads/2014/04/iarea.jpg)

* **AbstractArea**: Clase abstracta que implementará la interfaz IArea y donde definimos un constructor y alguna propiedad básica o común al resto de estrategias.

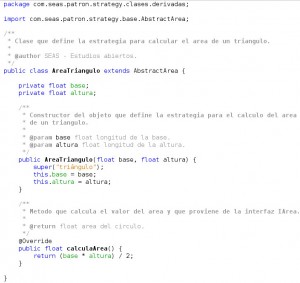
[](https://www.seas.es/blog/wp-content/uploads/2014/04/abstract-area.jpg)

A continuación desarrollaremos 3 clases para implementar la forma de calcular el área de un polígono regular, un círculo y un triángulo. Estas serán las estrategias existentes en nuestra aplicación de ejemplo.

* **AreaCircular**: Estrategia que define que el cálculo del área de un circulo se lleva a cabo multiplicando el valor del radio al cuadrado por el número PI.

[](https://www.seas.es/blog/wp-content/uploads/2014/04/area-circular.jpg)

* **AreaTriangulo**: Estrategia que define que el cálculo del área de un triángulo se lleva a cabo multiplicando el valor de la base por la altura dividiéndolo entre 2.

[](https://www.seas.es/blog/wp-content/uploads/2014/04/area-triangulo.jpg)

* **AreaPoligonoRegular**: Estrategia que define que el cálculo del área de un polígono regular de n lados conocido el radio de la circunferencia que lo contiene se lleva a cabo multiplicando el número de lados, por el radio al cuadrado, por el seno de 2PI dividido entre n y todo ello dividido entre 2.

[](https://www.seas.es/blog/wp-content/uploads/2014/04/area-poligono-regular.jpg)

Para completar el ejemplo, se han creado un par de excepciones específicas de forma que si se pasa un valor incorrecto al constructor de un objeto nuestra aplicación nos comunique cual es el error:

[](https://www.seas.es/blog/wp-content/uploads/2014/04/poligono-no-existe-exception.jpg)[](https://www.seas.es/blog/wp-content/uploads/2014/04/poligono-no-soportado-exception.jpg)

Finalmente, desarrollamos la clase CuerpoGeometrico donde podremos ver las distintas estrategias en acción:

[](https://www.seas.es/blog/wp-content/uploads/2014/04/cuerpo-geometrico.jpg)Finalmente, incluimos la clase con el método main y las pruebas del patrón:

[](https://www.seas.es/blog/wp-content/uploads/2014/04/strategy-principal.jpg)

Como siempre, el proyecto con el código está [disponible aquí](http://documentacion.eligetucurso.com/varios/SEAS_Patron_Strategy.rar).

No funciona el enlace pero consegui descárgalo ¡!!!!!!!!!!!!!!!!!!

https://www.seas.es/blog/informatica/patrones-de-diseno-en-java-patron-composite/

Patrones de diseño en Java: Patrón Composite

[4 OCTUBRE, 2013](https://www.seas.es/blog/informatica/patrones-de-diseno-en-java-patron-composite/)



Siguiendo con los patrones de diseño y su implementación en lenguaje Java, vamos a ver el patrón Composite. Como muchos otros patrones, está presente dentro de la JDK en la estructura de componentes y contenedores de la interfaz gráfica de usuario.

**Patrón Composite.**

El patrón Composite nos permite construir objetos complejos partiendo de otros más sencillos utilizando una estrategia de composición recursiva. Podemos equiparar este patrón a un panal y sus celdas. Cada objeto simple (celda) puede relacionarse con otros formando una estructura más compleja (panal).

También se puede usar este patrón para tratar a todos los componentes de la estructura en árbol para, mediante una interfaz o superclase, establecer unas reglas de comportamiento que nos permitan tratar a todos los elementos de la misma manera.

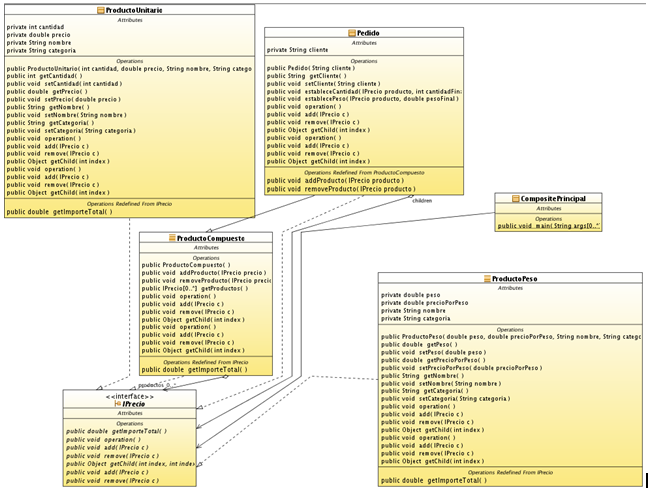
Identificaremos los siguientes componentes en la implementación del patrón:

* Interfaz o superclase de comportamiento: En ella estableceremos las pautas de comportamiento que tanto los componentes como el objeto compuesto deberán cumplir.
* Hojas: Serán las clases que implementan la interfaz que define el comportamiento, es decir, cada una de las celdas de nuestro panal.
* Composite: Será la clase que define el objeto compuesto de hojas. En este caso, el panal.
* Cliente: Será la clase que utilice la implementación de este patrón.

Una vez definidas las partes que intervendrán en este patrón, pasamos al ejemplo práctico.

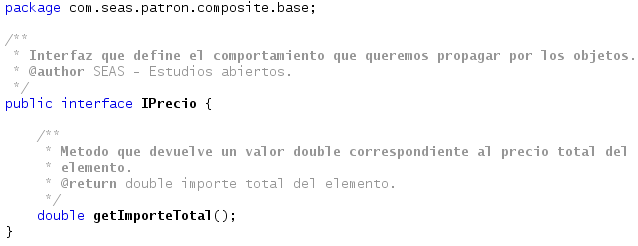
**Ejemplo práctico.**

Resumen gráfico de la implementación a través de un diagrama UML:



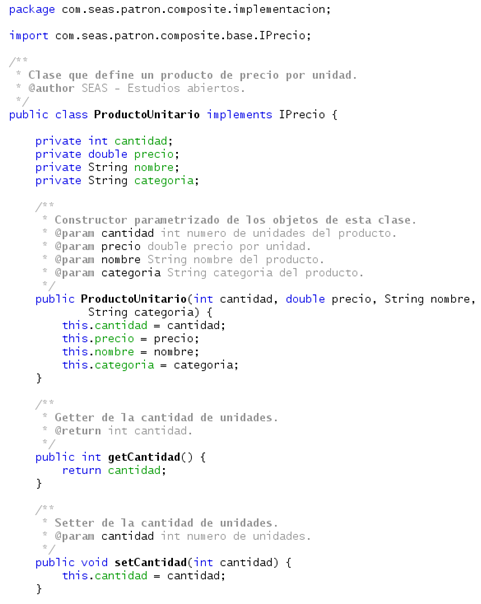
Como podemos observar en la implementación se ha diseñado un sistema de clases que representan distintos productos y pedidos y una interfaz IPrecio.

En la interfaz, definimos el comportamiento como se indicaba con anterioridad, estableciendo un método getImporteTotal() que como aparece en la documentación devuelve el precio del elemento/s.

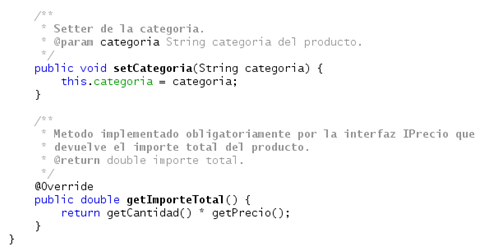


Cada una de las clases que implementan esta interfaz son hojas del patrón. En este caso hemos distinguido 3 clases:

* ProductoUnitario: representará productos cuya venta se realiza por unidades indivisibles (e.g.: una botella de vino). Estos productos tendrán una serie de características y una forma específica de calcular el precio (precio unitario por número de unidades).

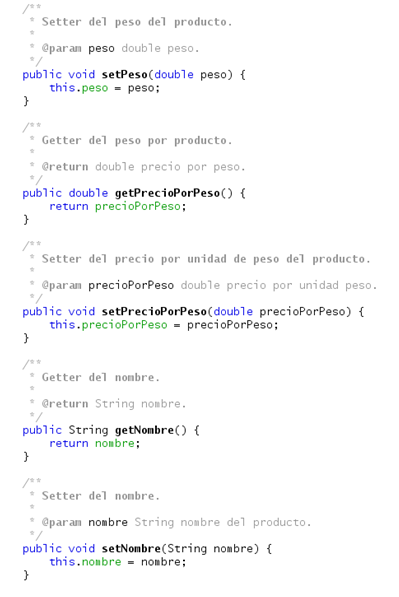


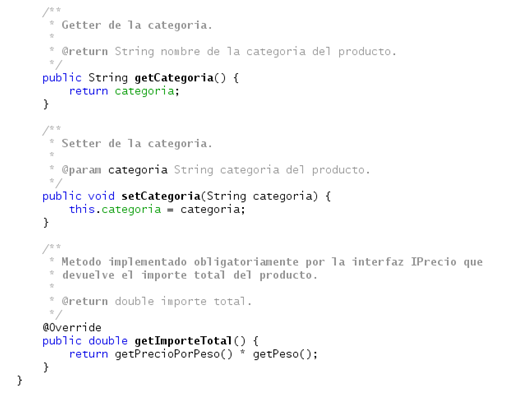




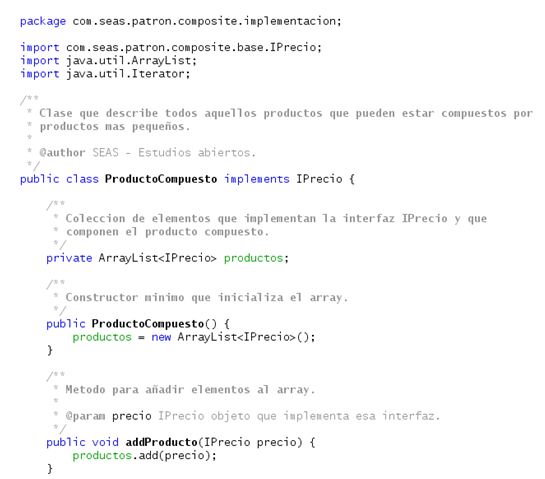
* ProductoPeso: representará productos cuya venta depende del peso del mismo (e.g.: un jamón). Como el anterior, tendrá una serie de propiedades y otra manera distinta de calcular el precio (peso del elemento/s por precio por peso).

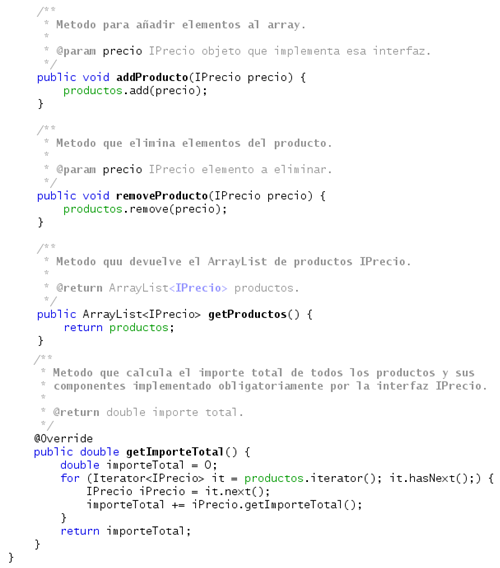




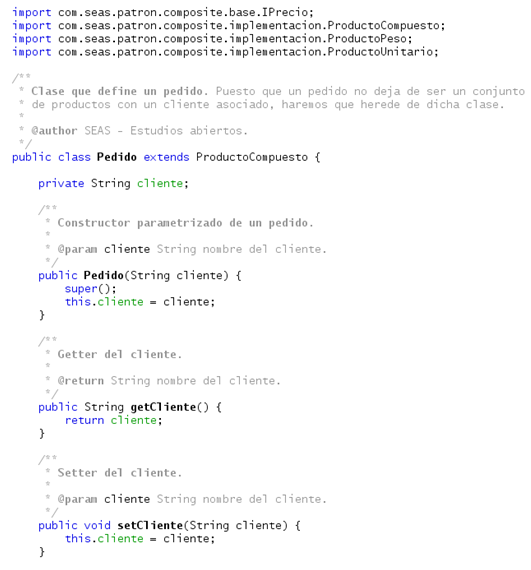


* ProductoCompuesto: representará un grupo de productos cuya venta se realiza de forma conjunta (e.g.: una cesta de navidad que contiene varios productos). Esta clase podría desarrollar a la vez los papeles de panal y celda. Contiene una colección de elementos IPrecio que lo conformarán y la implementación del método getImporteTotal() hará llamadas a las implementaciones de cada uno de los elementos de dicha colección.





Como último componente de este patrón vemos la clase Pedido, que es una especialización de la clase ProductoCompuesto que añadirá una serie de propiedades (e.g.: nombre del cliente) y unas funcionalidades añadidas.



Finalmente, crearemos la clase cliente que utilizará esta implementación:



El código fuente de este ejemplo está disponible para [descarga aquí](https://www.seas.es/blog/wp-content/uploads/2013/10/seas_patron_composite_proyecto.zip). . (ya descargado)

Éste enlace sí funciona!!!

Descargado

https://www.seas.es/blog/informatica/patrones-de-diseno-en-java-patron-observer/

Patrones de diseño en Java: Patrón Observer

[28 AGOSTO, 2013](https://www.seas.es/blog/informatica/patrones-de-diseno-en-java-patron-observer/)



Con este, doy comienzo a una serie de artículos sobre distintos patrones de diseño y su implementación en [lenguaje Java](https://www.seas.es/niveles/cursos/java?piloto=W59). El objetivo no es formar al programador experto, sino facilitar la comprensión de los patrones a todo aquel que se inicie en el mundo de la programación.

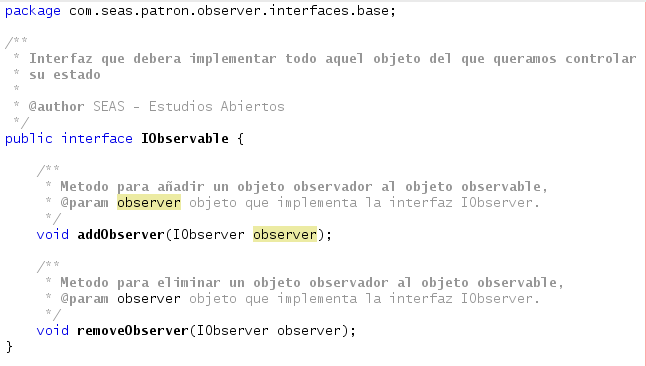
En este caso, comenzaremos con el patrón Observer que, por ejemplo, está presente en la JDK de Java en los denominados Listeners y Adapters que son utilizados para atender a las acciones realizadas por un usuario sobre un componente de la interfaz gráfica.

**Patrón Observer.**

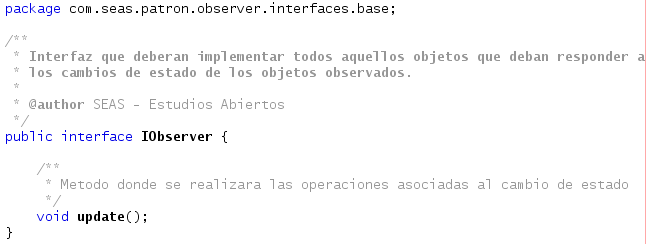
El patrón Observer nos permite implementar una estrategia que reaccione a los cambios de estado en  el objeto observado.

Debemos identificar los siguientes actores en la implementación del patrón:

* Objetos observables: Estos objetos deberán implementar un mecanismo para poder añadir y eliminar objetos observadores. Estableceremos las condiciones que deben cumplir a través de una interfaz IObservable. Además de los método que se ven a continuación, la interfaz, podría declarar un método “void notificar();” que se encargaría, en su implementación, de avisar a todos los observadores del cambio de estado.

​

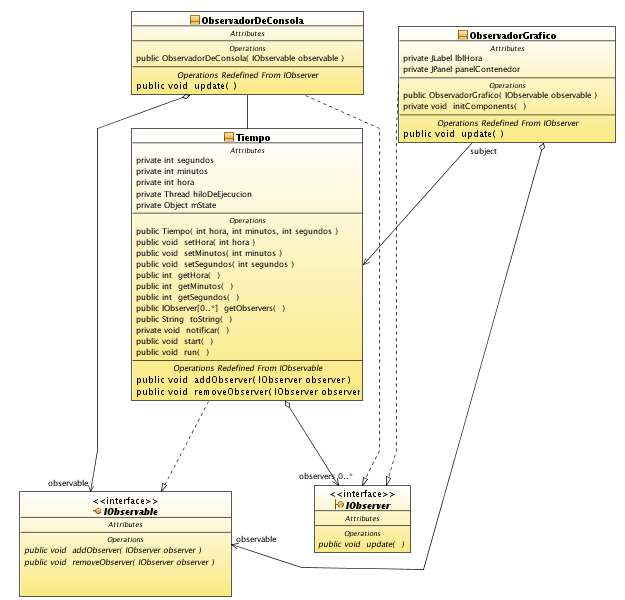
* Objetos observadores: serán aquellos objetos que implementen la interfaz IObserver donde vendrá definido, al menos, un método que defina la reacción correspondiente al cambio de estado.



Con estos contratos, nos aseguramos que los elementos observadores y a observar cumplen los requisitos y pueden ser tratados de una forma genérica que nos permita cierto grado de reutilización.

**Ejemplo práctico.**

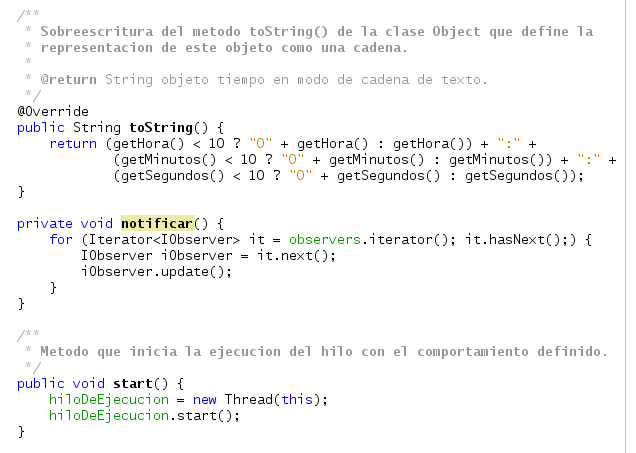
Resumen gráfico de la implementación a través de un diagrama UML:



A continuación se lleva a cabo la implementación de dicho patrón definiendo una clase Tiempo (horas, minutos y segundos), que tendrá codificados los métodos de la interfaz IObservable para añadir y eliminar observadores:



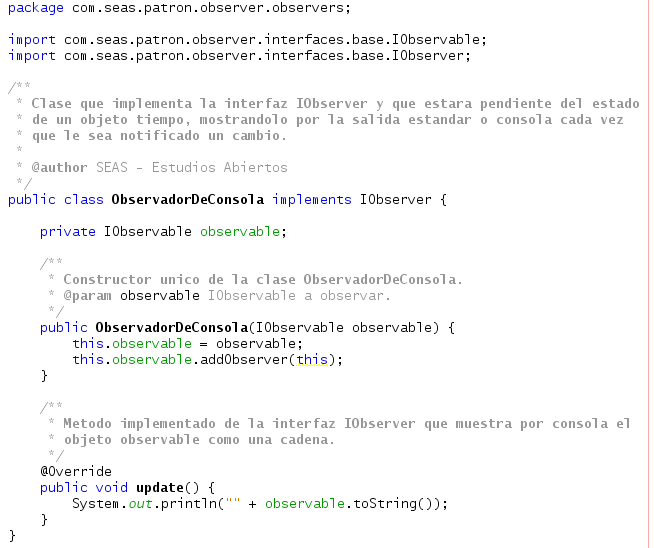
Incorporará también un método privado para notificar a los distintos observadores de los cambios en el estado, la sobreescritura del método toString() para convertir un objeto tiempo a cadena y un método que inicie la ejecución del hilo con el bucle infinito.



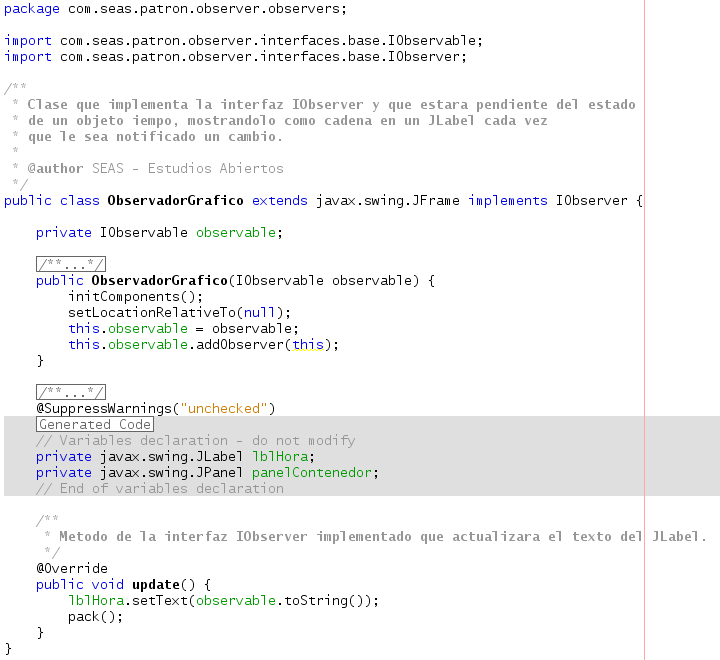
Dicho hilo utilizará la implementación de la interfaz Runnable que se encargará, cada segundo, de notificar el cambio del estado a los IObserver.



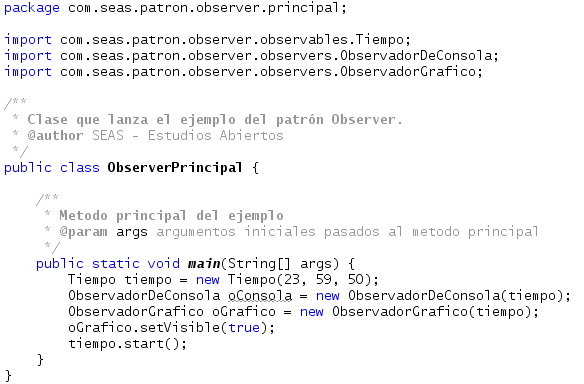
Finalmente, codificaremos los Observadores, uno para mostrar la hora cada segundo por consola:



Y otro para hacerlo a través de un JLabel en una ventana de la aplicación:



Finalmente, desarrollamos el método main que lanzará el ejemplo y mostrará como los observadores son notificados cada segundo de los cambios de estado en el objeto Tiempo observado:



El código fuente de este ejemplo está disponible para [descarga aquí](https://www.seas.es/blog/wp-content/uploads/seas_patron_observer_proyecto.zip). (ya descargado)

En el próximo articulo veremos el patrón Composite.