Modelado de objetos – UML

Contenidos

Conceptos de modelado

Cuando se modela algo, se crea una simplificación de la realidad para comprender mejor el sistema que se está desarrollando. Con UML, se construyen modelos a partir de bloques de construcción básicos, tales como clases, interfaces, componentes, nodos, dependencias, generalizaciones y asociaciones.

Como ningún sistema complejo puede ser comprendido completamente desde una única perspectiva, UML define varios diagramas que permiten centrarse en diferentes aspectos del sistema independientemente.

Diagramas Estructurales

Cuando se modelan sistemas reales, sea cual sea el dominio del problema, muchas veces se dibujan los mismos tipos de diagramas, porque representan vistas frecuentes de modelos habituales. Normalmente, las *partes estáticas de un sistema* se representarán mediante uno de los diagramas siguientes:

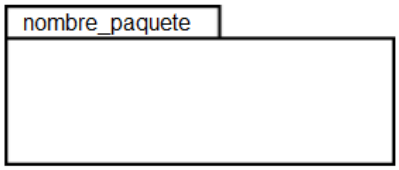
1. Diagramas de clases.
2. Diagramas de paquetes.
3. Diagramas de componentes.
4. Diagramas de perfiles.
5. Diagramas de estructura compuesta.
6. Diagramas de objetos.
7. Diagramas de despliegue.

Diagrama de clases. El diagrama UML más comúnmente usado, y la base principal de toda solución orientada a objetos. Las clases dentro de un sistema, atributos y operaciones, y la relación entre cada clase. Las clases se agrupan para crear diagramas de clases al crear diagramas de sistemas grandes.

Diagrama de paquetes. Es utilizado para definir los distintos paquetes a nivel lógico que forman parte de la aplicación y la dependencia entre ellos. Es principalmente utilizado por desarrolladores y analistas.

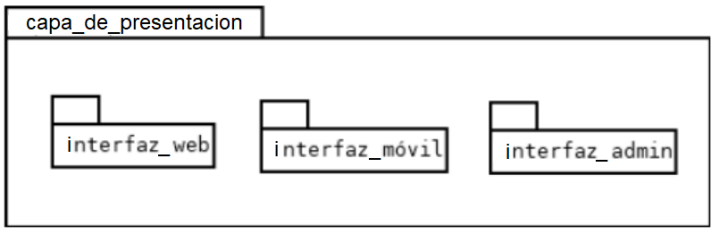
Un paquete es un conjunto de elementos. En concreto puede ser un conjunto de clases, casos de uso, componentes u otros paquetes.

Lo ideal es que este conjunto de elementos tenga una función diferenciada del resto de elementos. Es también importante identificar con nombres representativos de estas funciones a los distintos paquetes. Se representa con un símbolo simulando una carpeta, con el nombre en la parte superior como en la siguiente figura.



Representación de un paquete en UML

Como dijimos, un paquete puede contener otros paquetes. Como se ve en la siguiente figura.



Ejemplo de un sistema web dividido en paquetes

En este caso sería un paquete denominado *Capa de presentación* que contiene los paquetes «Interfaz web», «Interfaz móvil» e «Interfaz admin». En este caso el contenido son otros paquetes, pero podría ser, como ya hemos dicho, otro diagrama.

Por convención los nombres de paquetes en Java se escriben en minúsculas y pueden incluir puntos para separar niveles de jerarquía entre clases. Por ejemplo, en los ficheros de código Java se usa la palabra reservada *package* para especificar a qué paquete pertenece una clase o conjunto de clases. Suele indicarse como primera sentencia:

```
package java.awt.event;
```

Para usar un paquete dentro del código se usa la declaración *import*. Si sólo se indica el nombre del paquete, se importarán todas las clases que contiene:

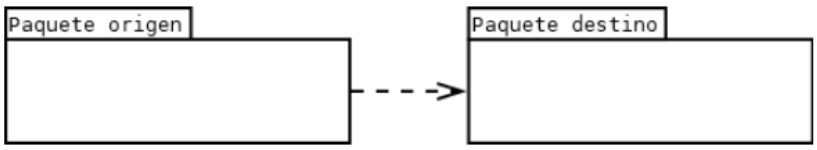
```
import java.awt.event.*;
```

Si además del nombre del paquete se especifica una clase, sólo se importa esa clase:

```
import java.awt.event.ActionEvent;
```

La creación de paquetes de programación se realiza mediante algún IDE como Apache NetBeans.

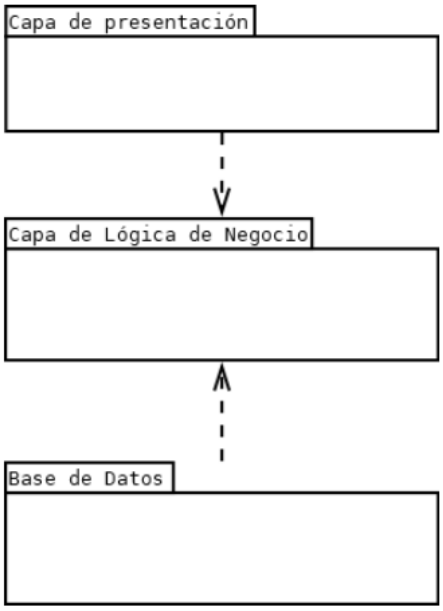
Una ***dependencia entre paquetes*** representa que un paquete necesita de los elementos de otro paquete para poder funcionar con normalidad. Se representa con una flecha discontinua que va desde el paquete que requiere la función hasta el paquete que ofrece esa función.



Representación de dependencia entre paquetes

En esta imagen se dice que el Paquete Origen depende del Paquete Destino para dar su servicio.

Un ejemplo de relación de dependencia que aparece en la realidad un gran número de ocasiones es el siguiente.

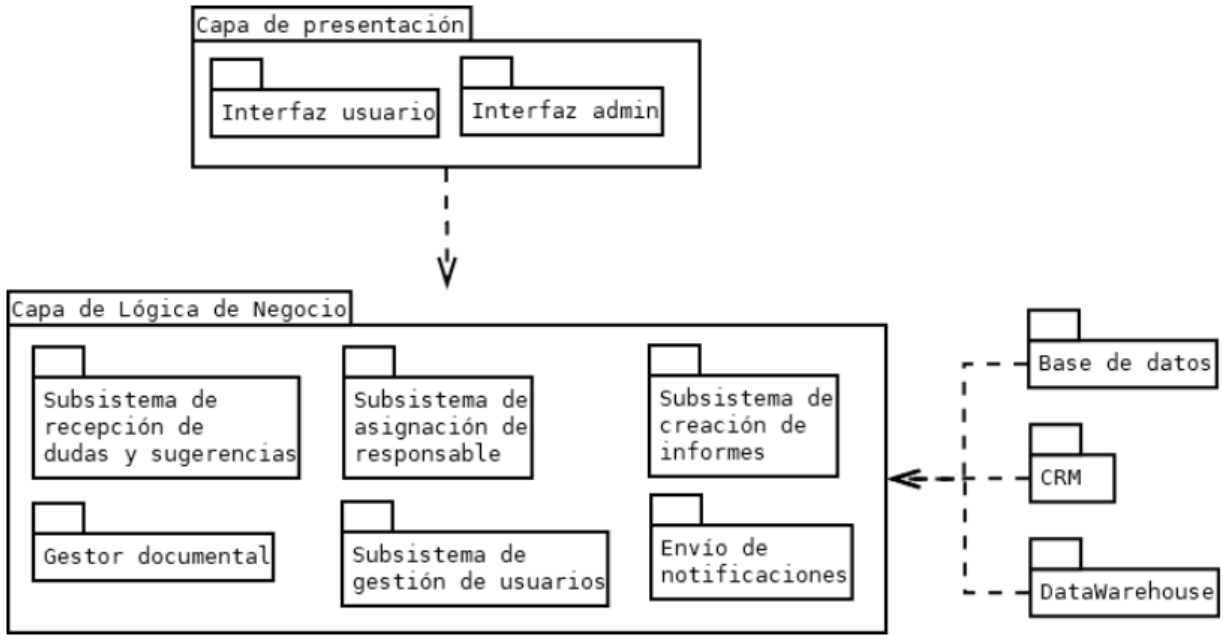


Ejemplo de relación de dependencia entre paquetes

Este diagrama, típico de arquitecturas de aplicaciones en tres capas tienen 3 paquetes: Capa de Lógica de Negocio, Capa de Presentación y la Base de datos. Estos dos últimos paquetes (presentación y bases de datos) dependen para su funcionamiento de la lógica de negocio.

Ejemplo. A continuación se muestra, un diagrama de paquetes de una aplicación que tiene como finalidad la recepción y gestión de quejas y sugerencias. Estaría compuesta por los siguientes paquetes: *Capa de presentación*. Incluye a su vez los paquetes Interfaz de Usuario e Interfaz Admin. Capa de *Lógica de Negocio*, con los siguientes paquetes: Subsistema de recepción de dudas y sugerencias. Subsistema de asignación de responsable. Subsistema de creación de informes. Gestor documental. Subsistema de gestión de usuarios. Envío de notificaciones. Base de datos. CRM (Customer

Relationship Management, lo que traducido al español es algo como Gestión de Relaciones con el Cliente). DataWarehouse.



Ejemplo de diagrama de paquetes

La explicación completa del ejemplo queda fuera del alcance del capítulo, pero el alumno puede investigar el significado de cada uno de los componentes del ejemplo.

Diagrama de componentes. Habitualmente se utiliza después de haber creado el diagrama de clases, pues necesita información de este diagrama como pueden ser las propias clases. Este diagrama proporciona una vista de alto nivel de los componentes dentro de un sistema. Los componentes pueden ser un componente de software, como una base de datos o una interfaz de usuario; o un componente de hardware como un circuito, microchip o dispositivo; o una unidad de negocio como un proveedor, nómina o envío.

Diagramas de perfiles. Un diagrama de perfiles permite extender UML para su uso con una plataforma de programación en particular (como el framework .NET de Microsoft o la plataforma Java Enterprise Edition), o modelar sistemas destinados a ser usados en un dominio en particular (por ejemplo, medicina, servicios financieros o ingeniería especializada).

Diagramas de estructura compuesta. Un diagrama de estructura compuesta desempeña una función similar a un diagrama de clases, pero le permite entrar en más detalles al describir la estructura interna de varias clases y mostrar las interacciones entre ellas. Puede representar gráficamente clases y partes internas y mostrar asociaciones tanto entre clases como dentro de ellas.

Diagramas de objetos. El diagrama de objetos fue definido en la ahora obsoleta especificación UML 1.4.2 como "*Un gráfico de instancias, incluyendo objetos y valores de datos. Cada diagrama de objetos representa una instancia de un diagrama de clase; muestra una fotografía del estado detallado de un sistema en un punto específico del tiempo*". Esta misma especificación también afirma que el diagrama de objetos es «*un diagrama de clase con objetos y no clases*»

El actual UML 2.5, en su jerarquía de diagramas, muestra este diagrama de objetos como un diagrama completamente independiente del mencionado diagrama de clases. Otros profesionales mantienen que los diagramas de componentes y los diagramas de despliegue contienen son, en el fondo, diagramas de objetos.

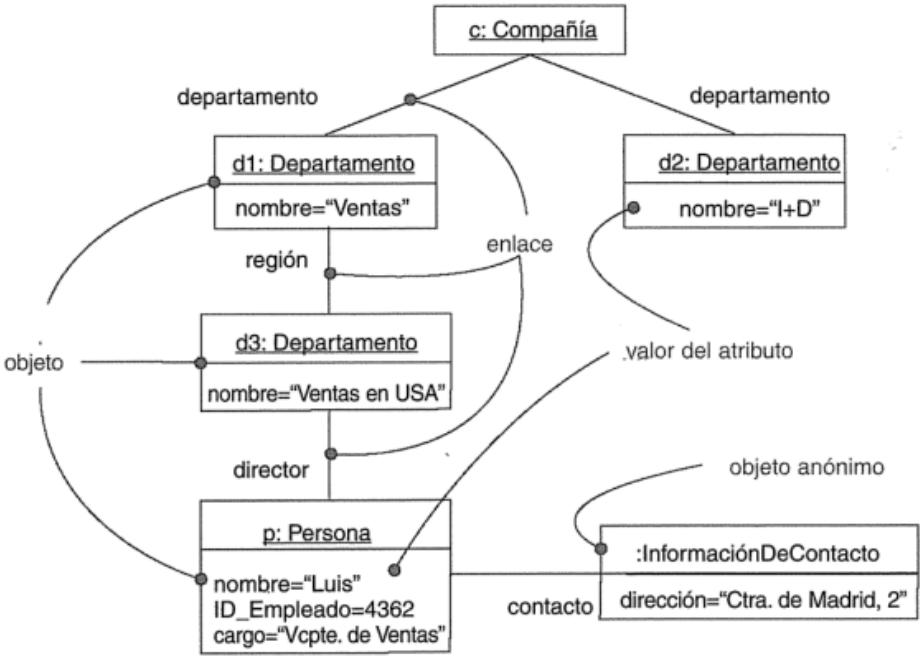
En UML, los diagramas de objetos muestran un instante en el sistema y las relaciones entre distintas instancias. Algunas líneas generales en comparación con el diagrama de clases son las siguientes:

- El diagrama de objetos utiliza notaciones similares a los usados en el diagrama de clases.

- Los diagramas de objetos se utilizan para modelar los elementos que están presentes en un diagrama de clases.
- El diagrama de objetos muestra los clasificadores reales del sistema y las relaciones entre ellos en un punto específico del tiempo.
- Los diagramas de objetos se pueden instanciar como diagrama de clases, despliegue, componentes e, incluso, casos de uso.
- En ninguno de los dos diagramas (de clases y objetos) se muestran los mensajes entre los elementos que colaboran, ya que se trata de diagramas estructurales.

Elementos del diagrama de objetos

Objetos: Cada objeto se representa con un rectángulo con su nombre (comienza con minúsculas) y el de su clase (comienza con mayúsculas) en la parte superior subrayados y separados por dos puntos. En caso de ser un objeto anónimo no se escribe su nombre, dejando solo el de la clase.



Un diagrama de objetos

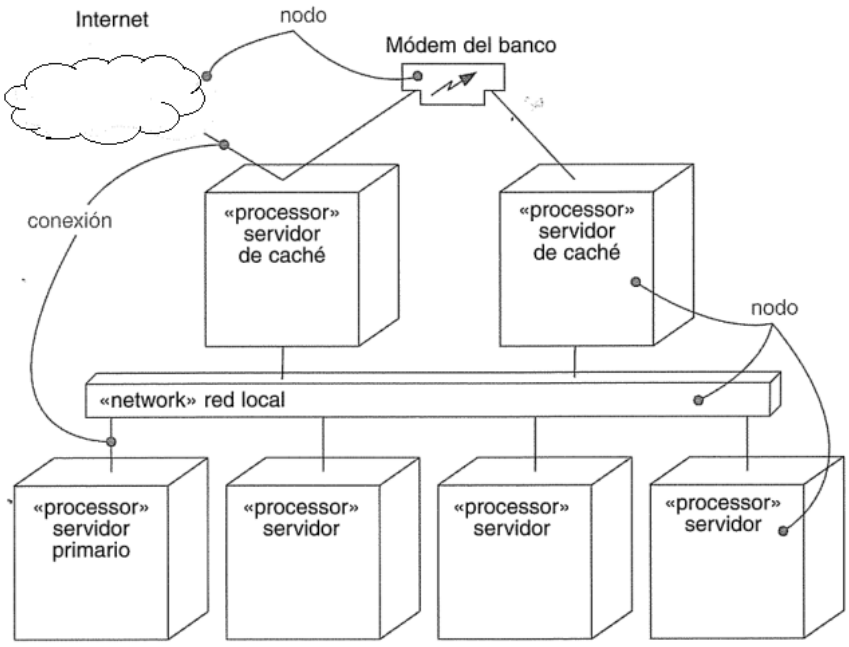
Atributos: De igual forma que el diagrama de clases, se muestra en un compartimento en la parte inferior del nombre del objeto. A diferencia de las clases, los atributos pueden tener valores asignados a ellos.

Vínculos: Son asociaciones entre dos objetos y se representan con los mismos elementos que en el diagrama de clases. Por ejemplo, una asociación.

Diagramas de despliegue. Los diagramas de despliegue se utilizan para modelar la vista de despliegue estática de un sistema. La mayoría de las veces, esto implica modelar la topología del hardware sobre la que se ejecuta el sistema.

Los diagramas de despliegue son importantes para visualizar, especificar y documentar por ejemplo sistemas embebidos, sistemas cliente/servidor y sistemas distribuidos.

UML ha sido diseñado para modelar muchos de los aspectos hardware de un sistema a un nivel suficiente para que un ingeniero de software pueda especificar la plataforma sobre la que se ejecutará el software del sistema, y para que un ingeniero de sistemas pueda manejar la frontera entre el hardware y el software del sistema.



Un diagrama de despliegue

Normalmente, los diagramas de despliegue contienen:

- Nodos.
- Relaciones de dependencia y asociación.

Al igual que los demás diagramas, los diagramas de despliegue pueden contener notas, restricciones y paquetes.

Diagramas de comportamiento

Los diagramas de comportamiento se emplean para visualizar, especificar, construir y documentar los aspectos dinámicos de un sistema. Los aspectos dinámicos de un sistema involucran cosas tales como el flujo de mensajes a lo largo del tiempo y el movimiento físico de componentes en una red. Tenemos:

- Diagramas de casos de uso
- Diagramas de máquina de estados

- Diagramas de actividades
- Diagramas de interacción: diagramas de secuencia, diagramas de comunicación, diagramas de tiempos y diagramas global de interacciones.

Diagramas de casos de uso. Son importantes para modelar el comportamiento de un sistema, un subsistema o una clase. Cada uno muestra un conjunto de *casos de uso*, *actores* y sus *relaciones*.

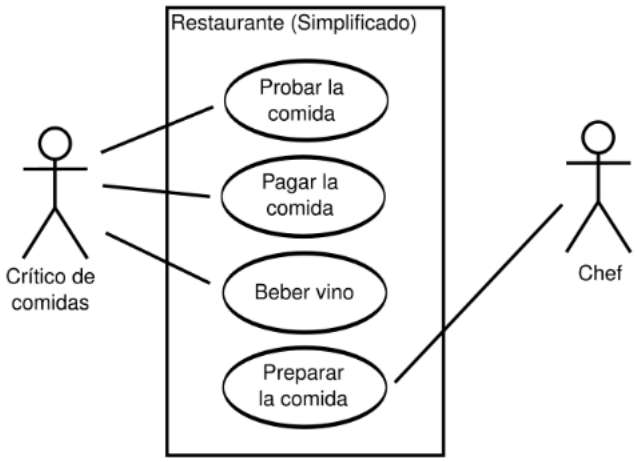


Diagrama de casos de uso para un restaurante

El diagrama anterior describe la funcionalidad de un Sistema Restaurante muy simple. Los casos de uso están representados por elipses y se muestran como parte del sistema que está siendo modelado, los actores no.

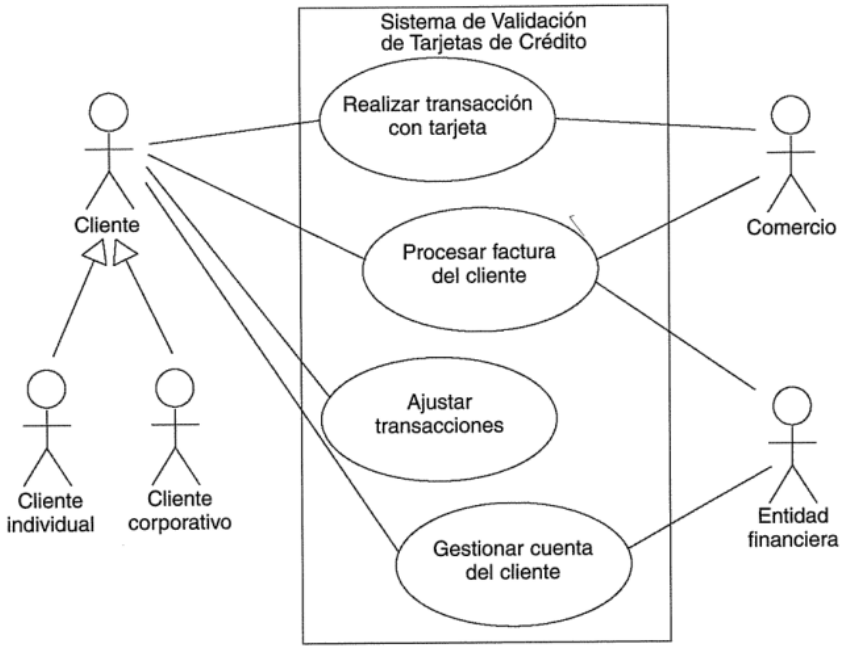
En este caso, podemos apreciar tanto declaraciones correctas como incorrectas. El probar la comida y pagarla es un requerimiento funcional del sistema, pero **beber vino no lo es, por lo tanto este caso de uso está incorrecto**.

Normalmente, un diagrama de casos de uso contiene:

- Sujetos.
- Casos de uso.
- Actores.
- Relaciones de dependencia, generalización y asociación.

Al igual que los demás diagramas, pueden contener notas y restricciones.

Los diagramas de casos de uso también pueden contener paquetes, que se emplean para agrupar elementos del modelo en partes mayores.



Modelado del contexto de un sistema

En el ejemplo anterior se muestra el contexto de un sistema de validación de tarjetas de crédito, destacando los actores en torno al sistema. Se puede ver que existen Clientes, de los cuales hay dos categorías (Cliente Individual y Cliente Corporativo). Estos actores representan los roles que juegan las personas que interactúan con el sistema. En este contexto, también hay actores que representan a otras instituciones, tales como Comercio (con el cual un Cliente realiza una transacción con tarjeta para comprar un artículo o un servicio) y Entidad Financiera (que presta servicio como sucursal bancaria para la cuenta de la tarjeta de crédito. En el mundo real, estos dos últimos actores probablemente serán a su vez sistemas con gran cantidad de software.

Diagramas de máquina de estados. El *diagrama de máquina de estados* o, más comúnmente llamado, el *diagrama de estados* es un diagrama de comportamiento usado para especificar el comportamiento de una parte del sistema diseñado a través de transiciones de estados finitos. Es utilizado para mostrar los estados por los que pasa un componente de un sistema de información.

El comportamiento se modela utilizando una serie de nodos que representan estados y que están conectados a través de las llamadas transiciones. Estas transiciones se activan a través de eventos.

El diagrama de estados está formado por tres elementos: **estados, pseudoestados y transiciones**.

Diagramas de actividades. Un *diagrama de actividades* es fundamentalmente un *diagrama de flujo* que destaca la actividad que tiene lugar a lo largo del tiempo. Un *diagrama de interacción* muestra objetos que pasan mensajes; un *diagrama de actividades* muestra las operaciones que se pasan entre los objetos. La diferencia semántica es sutil, pero tiene como resultado una forma muy diferente de mirar el mundo. Al contrario de un *diagrama de flujo* clásico, un diagrama de actividad muestra tanto la concurrencia como las bifurcaciones del control, como se ve en la siguiente figura.

Normalmente, los diagramas de actividades contienen:

- Estados de actividad y estados de acción.
- Transiciones.
- Objetos.

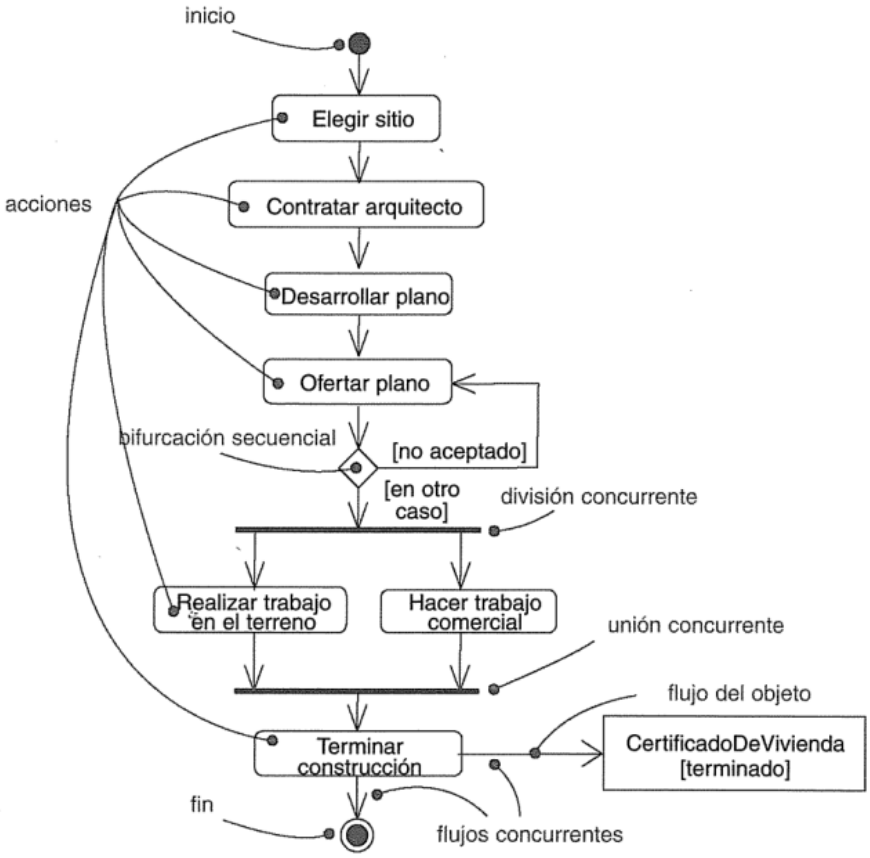


Diagrama de actividades

Diagramas de interacción. Dentro de los diagramas de interacción tenemos: *diagramas de secuencia*, *diagramas de comunicación*, *diagramas de tiempos* y *diagramas global de interacciones*.

Normalmente, los diagramas de interacción contienen:

- Roles u objetos.
- Comunicaciones o enlaces.
- Mensajes.

Al igual que los demás diagramas, pueden contener notas y restricciones.

Diagrama de secuencia. El diagrama de secuencia es un tipo de diagrama de interacción contenido en UML 2.5. Su objetivo es representar el intercambio de mensajes entre los distintos objetos del sistema para cumplir con una funcionalidad. Define, por tanto, el comportamiento dinámico del sistema de información.

Normalmente es utilizado para definir cómo se realiza un caso de uso por lo que es comúnmente utilizado junto al diagrama de casos de uso. También se suele construir para comprender mejor el diagrama de clases, ya que el diagrama de secuencia muestra como objetos de esas clases interactúan haciendo intercambio de mensajes.

Un *diagrama de secuencia* es un diagrama de interacción que destaca la ordenación temporal de los mensajes. Gráficamente, un diagrama de secuencia es una tabla que representa objetos, dispuestos a lo largo del eje X, y mensajes, ordenados según se suceden en el tiempo, a lo largo del eje Y. Vea la siguiente figura.

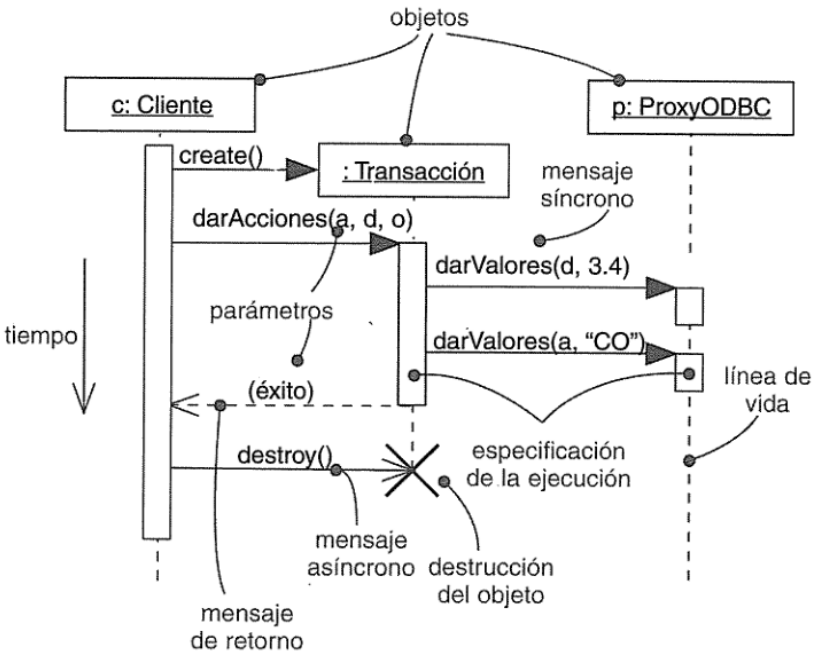


Diagrama de secuencia

Un diagrama de secuencia se forma colocando en primer lugar los objetos o roles que participan en la interacción en la parte superior del diagrama, a lo largo del eje horizontal. Normalmente, se coloca a la izquierda el objeto o rol que inicia la interacción, y los objetos o roles subordinados a la derecha. A continuación, se colocan los mensajes que estos objetos envían y reciben a lo largo del eje vertical, en orden de sucesión en el tiempo, de arriba abajo. Esto ofrece al lector una señal visual clara del flujo de control a lo largo del tiempo.

Los diagramas de secuencia tienen dos características que los distinguen de los diagramas de comunicación.

En primer lugar, está la línea de vida. La línea de vida de un objeto es la línea discontinua vertical que representa la existencia de un objeto a lo largo de un periodo de tiempo.

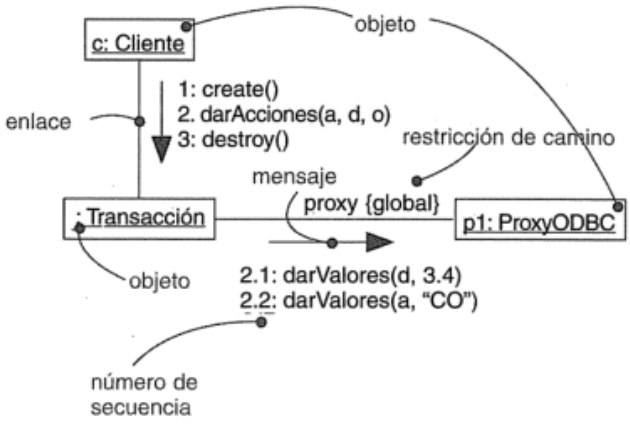
Pueden crearse objetos durante la interacción. Sus líneas de vida comienzan con la recepción del mensaje estereotipado como *create*. Los objetos pueden destruirse durante la interacción. Sus líneas de vida acaban con la recepción del mensaje estereotipado como *destroy* (además se muestra la señal visual de una gran X que marca el final de sus vidas).

En segundo lugar, está el foco de control. El foco de control es un rectángulo delgado y estrecho que representa el período de tiempo durante el cual un objeto ejecuta una acción, bien sea directamente o a través de una procedimiento subordinado. La parte superior del rectángulo se alinea con el comienzo de la acción; la inferior se alinea con su terminación.

El contenido principal de los diagramas de secuencia son los mensajes. Un mensaje se representa con una flecha que va de una línea de vida hasta otra. La flecha apunta al receptor. Si el mensaje es asíncrono, la flecha es abierta. Si el mensaje es síncrono

(una llamada), la flecha es un triángulo relleno. Una respuesta a un mensaje síncrono (el retorno de una llamada) se representa con una flecha abierta discontinua. El mensaje de retorno puede omitirse, ya que de manera implícita hay un retorno después de cada llamada, pero a menudo es útil para reflejar valores de retorno.

Diagramas de comunicación. El diagrama de comunicación (denominado previamente diagrama de colaboración en las primeras versiones de UML) es un tipo de diagrama de interacción UML que muestra las interacciones entre objetos y/o partes (representadas como líneas de vida) utilizando mensajes secuenciados en una disposición de forma libre.



Ejemplo de diagramas de comunicación

Diagramas de tiempo. El diagrama de tiempos es un diagrama UML incluido en la categoría de diagramas de interacción (perteneciente a los diagramas de comportamiento). Es utilizado para modelar el comportamiento del sistema dando especial importancia al tiempo. Los diagramas de tiempo se centran en las condiciones que cambian dentro y entre las líneas de vida a lo largo de un eje de tiempo lineal. Los diagramas de tiempo describen el comportamiento de los clasificadores individuales y las interacciones de los clasificadores, enfocando la atención en el tiempo de los eventos que causan cambios en las condiciones de las líneas de vida.

El diagrama de tiempos incluye los siguientes elementos: Líneas de vida, Estados, Restricciones de duración y Restricciones de tiempo.

Diagrama global de interacciones. El diagrama global de interacciones es uno de los diagramas incluidos dentro de UML en la clasificación de diagramas de comportamiento y, en concreto, diagramas de interacción.

Se trata de una variante del *Diagrama de actividades* donde los nodos son las interacciones o apariciones de interacción. Este diagrama se enfoca en la visión general del flujo de control de las interacciones, que también puede mostrar el flujo de actividad entre los diagramas. Dicho de otra manera, se utiliza para vincular los diagramas y lograr un alto grado de navegabilidad entre los diagramas.

Al tratarse de una variante del diagrama de actividades, utiliza la notación de este diagrama.

Patrones de diseño

Los patrones de diseño son unas técnicas para resolver problemas comunes en el desarrollo de software y otros ámbitos referentes al diseño de interacción o interfaces. Cada patrón es como un plano que se puede personalizar para resolver un problema de diseño particular de tu código. Podemos decir, que un patrón de software es una:

- Solución probada a un problema recurrente.
- Solución reutilizable a un problema común en un contexto dado.

Además cada patrón tiene sus ventajas y desventajas ante determinadas situaciones. No se puede elegir un patrón y copiarlo en el programa como si se tratara de funciones o bibliotecas ya preparadas. El patrón no es una porción específica de código, sino un concepto general para resolver un problema particular. Puedes seguir los detalles del patrón e implementar una solución que encaje con las realidades de tu propio programa.

El concepto de patrón de diseño existe desde finales de los 70's, pero cobra fuerza en los 90's cuando "la banda de los cuatro"[1], publicó el libro "*Design Patterns - Elements of Reusable Object-Oriented Software*" donde se explican 23 patrones de diseño que luego se convirtieron en referentes en el desarrollo de software.

Los patrones de diseño pretenden:

- Proporcionar catálogos de elementos reusables en el diseño de sistemas de software.
- Evitar la reiteración en la búsqueda de soluciones a problemas ya conocidos y solucionados anteriormente.
- Formalizar un vocabulario común entre diseñadores.
- Estandarizar el modo en que se realiza el diseño.
- Facilitar el aprendizaje a las nuevas generaciones de diseñadores condensando conocimiento ya existente.

Asimismo, no pretenden:

- Imponer ciertas alternativas de diseño frente a otras.
- Eliminar la creatividad inherente al proceso de diseño.

No es obligatorio utilizar los patrones, solo es aconsejable en el caso de tener el mismo problema o similar que soluciona el patrón, siempre teniendo en cuenta que en un caso particular puede no ser aplicable. "Abusar o forzar el uso de los patrones puede ser un error".

"Cada patrón describe un problema que ocurre una y otra vez en nuestro medio ambiente y, a continuación describe el núcleo de la solución a ese problema, de tal manera que se puede utilizar esta solución un millón de veces, sin tener que hacerlo de la misma manera dos veces".

[1] Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides.

1977, A Pattern Language, Christopher Alexander

Según el autor, puede haber distintas clasificaciones de patrones. A continuación se da una clasificación

Según la escala o nivel de abstracción tenemos patrones:

Patrones de arquitectura: Aquellos que expresan un esquema organizativo estructural fundamental para sistemas de software. Dentro de estos patrones tenemos por ejemplo al patrón MVC (Modelo Vista Controlador).

Patrones de diseño: Aquellos que expresan esquemas para definir estructuras de diseño (o sus relaciones) con las que construir sistemas de software. Se especifican y clasifican en la siguiente sección.

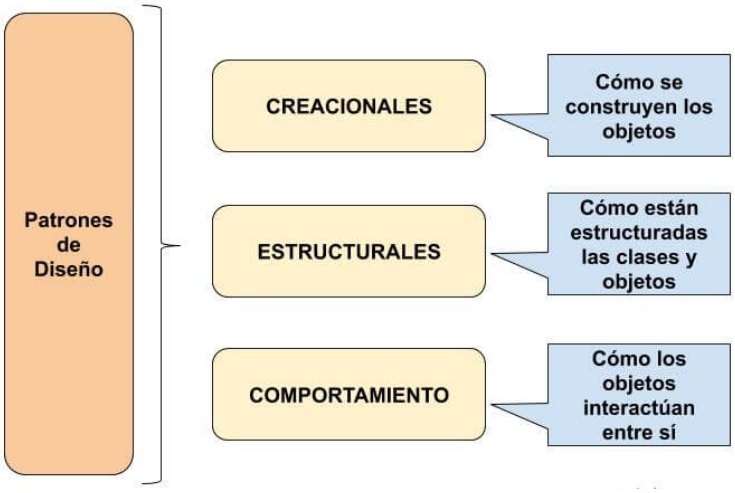
Dialectos: Patrones de bajo nivel específicos para un lenguaje de programación o entorno concreto.

Clasificación de los patrones de diseño

A su vez los patrones de diseño se clasifican en:

- Creacionales: simplifican la creación de objetos y están basados en los conceptos de encapsulamiento y ocultamiento. Con estos patrones de diseño normalmente se trabaja con *interfaces*. Ejemplos: Abstract-Factory, Factory method, Builder, Singleton, Prototype.
- Estructurales: Son los patrones de diseño software que solucionan problemas de composición (agregación) de clases y objetos. Es decir, cómo se relacionan las clases del sistema. Ejemplos: Adapter, Bridge, Composite, Decorator, Facade, Flyweight, Proxy.
- De comportamiento: Se definen como patrones de diseño software que ofrecen soluciones respecto a la interacción y responsabilidades entre clases y objetos,

así como los algoritmos que encapsulan. Ejemplos: Command, Interpreter, Iterator, Mediator, Observer, Strategy, Template method, Visitor.



Clasificación de patrones de diseño

En la siguiente figura se deja un listado clasificado de patrones. Este listado no es definitivo pero lista a la mayoría de los patrones de diseño

		Propósito		
		De Creación	**Estructurales**	**De Comportamiento**
Ámbito	**Clase**	• Factory Method	• Adapter (de clases)	• Interpreter • Template Method
	Objeto	• Abstract Factory • Builder • Prototype • Singleton	• Adapter (de objetos) • Bridge • Composite • Decorator • Façade • Flyweight • Proxy	• Chain of Responsibility • Command • Iterator • Mediator • Memento • Observer • State • Strategy • Visitor

Patrones de diseño

Estos patrones, tal como la tecnología, van mutando y evolucionando.

Patrones de diseño Singleton

Singleton es un patrón de diseño creacional que nos permite asegurarnos de que una clase tenga una única instancia, a la vez que proporciona un punto de acceso global a dicha instancia.

Problema. El patrón Singleton resuelve dos problemas al mismo tiempo, vulnerando el Principio de responsabilidad única:

1. Garantizar que una clase tenga una única instancia. ¿Por qué querría alguien controlar cuántas instancias tiene una clase? El motivo más habitual es controlar el acceso a algún recurso compartido, por ejemplo, una base de datos o un archivo.

   Funciona así: imagina que has creado un objeto y al cabo de un tiempo decides crear otro nuevo. En lugar de recibir un objeto nuevo, obtendrás el que ya habías creado.

   Ten en cuenta que este comportamiento es imposible de implementar con un constructor normal, ya que una llamada al constructor siempre debe devolver un nuevo objeto por diseño.

2. Proporcionar un punto de acceso global a dicha instancia. ¿Recuerdas esas variables globales que utilizaste (bueno, sí, fui yo) para almacenar objetos esenciales? Aunque son muy útiles, también son poco seguras, ya que cualquier código podría sobrescribir el contenido de esas variables y descomponer la aplicación.

   Al igual que una variable global, el patrón Singleton nos permite acceder a un objeto desde cualquier parte del programa. No obstante, también evita que otro código sobreescriba esa instancia.

Este problema tiene otra cara: no queremos que el código que resuelve el primer problema se encuentre disperso por todo el programa. Es mucho más conveniente tenerlo dentro de una clase, sobre todo si el resto del código ya depende de ella.

Hoy en día el patrón Singleton se ha popularizado tanto que la gente suele llamar singleton a cualquier patrón, incluso si sólo resuelve uno de los problemas antes mencionados.

Solución. Todas las implementaciones del patrón Singleton tienen estos dos pasos en común:

1. Hacer privado el constructor por defecto para evitar que otros objetos utilicen el operador new con la clase Singleton.
2. Crear un método de creación estático que actúe como constructor. Tras bambalinas, este método invoca al constructor privado para crear un objeto y lo guarda en un campo estático. Las siguientes llamadas a este método devuelven el objeto almacenado en caché.

Si tu código tiene acceso a la clase Singleton, podrá invocar su método estático. De esta manera, cada vez que se invoque este método, siempre se devolverá el mismo objeto.