

## 3 Fundamentos de la programación orientada a objetos

**Objetivo:** Explicar los diferentes paradigmas de programación, así como los conceptos y diseños de la programación orientada a objetos en solución de problemas.





*Aprendiendo UML En 24 Horas.* Joseph Schmuller,  
ISBN: 968444463X, Prentice-Hall, noviembre 2001

### 3.1 Paradigmas de programación: imperativa, funcional, lógica, declarativa y orientada a objetos.

El concepto de paradigma procede del griego **παραδειγμα** (paradeigma), que significa modelo o ejemplo.

Un paradigma es un modelo o patrón sostenido en una disciplina. Comprende un grupo de principios fundamentales que producirán una realidad con un propósito particular.

En la ciencia un paradigma es un conjunto de realizaciones científicas universalmente reconocidas que, durante cierto tiempo, proporcionan modelos experimentales que dan solución a otros modelos.

Por ejemplo, el paradigma de Ptolomeo implicaba que la tierra era plana y que el Sol giraba alrededor de la Tierra. Las observaciones de Copérnico probaron lo contrario y generaron el paradigma de que la Tierra es redonda y gira alrededor del Sol.

Por ejemplo, en 1905 Albert Einstein publicó su teoría de la relatividad especial, su publicación cambió el paradigma que había impuesto Isaac Newton.

## Paradigma de programación

- Es la forma, visión o manera que determinan los métodos y las herramientas que un programador usara en la construcción de un software.
- Es una colección de modelos conceptuales que modelan el proceso de diseño y determinan la estructura de un programa.

- Se refiere a una filosofía (o forma) de atacar y analizar problemas, así como diseñar e implementar una solución en una computadora.
- Representa un enfoque particular o filosofía para la construcción del software. No es mejor uno que otro sino que cada uno tiene ventajas y desventajas. También hay situaciones donde un paradigma resulta más apropiado que otro.

Los lenguajes de programación están basados en uno o más paradigmas de programación.

## Imperativo

Describe una programación como un flujo de instrucciones o comandos que van formando la estructura del software.

La máquina almacena una representación codificada de un cálculo y ejecuta una secuencia de comandos que modifican el contenido de ese almacenamiento.

Este paradigma expresa como debe solucionarse un problema especificando una secuencia de acciones a realizar a través de uno o más procedimientos denominados subrutinas o funciones.

El paradigma imperativo debe su nombre al papel dominante que desempeñan las sentencias imperativas. Su esencia es el cálculo iterativo, paso a paso, de valores de nivel inferior y su asignación a posiciones de memoria.

Algunos lenguajes imperativos son Algol, C, COBOL, FORTRAN, Pascal, Perl, PHP.

{Programa que calcula el factorial de un número}

uses crt; {bibliotecas}

var numero,limite,contador:integer; {variables}

begin

ClrScr;

Write('Numero: ');

Readln(numero);

limite:=numero;

numero:=1;

for contador:=1 to limite do

begin

numero:=numero\*contador;

end;

Write('Factorial: ',numero);

.ReadKey;

end.

## Funcional

El paradigma funcional se caracteriza por el uso de expresiones y funciones. Está basado en el concepto matemático de función:

*Una función  $f$  asigna a cada miembro de un conjunto  $X$ , exactamente un miembro de un conjunto  $Y$ .*

Este tipo de paradigma muestra un tipo de programación en forma de funciones matemáticas.

Por tanto, un programa es una función o un grupo de funciones compuestas por funciones más simples estableciendo que una función puede llamar a otra, o el resultado de una función puede ser usado como argumento de otra función.

Por ejemplo, si se desea obtener la calificación promedio de un alumno, se puede construir una función promedio que se obtendría a partir de otras funciones más simples: una (sumar) la cual obtiene la suma de las entradas de la lista, otra (contar) la cual cuenta el número de entradas de la lista y la tercera (dividir) que obtiene el cociente de los valores anteriores, su sintaxis sería:

(dividir (sumar notas) (contar notas))

Obsérvese que la estructura anidada refleja el hecho de que la función *dividir* actúa sobre los resultados de las funciones *suma* y *contar*.

Algunos ejemplos de lenguajes funcionales son Haskell, Miranda, Scala, Lisp, Scheme, Ocaml, SAL, Standard ML.

## Ejemplo en LISP

; Definición matemática

; Factorial(x) = 1 si x=0 caso base

; x\*factorial(x-1) si x>0 caso recursivo

; Función factorial hecha con recursividad  
(defun factorial (n)

(

if (= 0 n)

1

; caso base

(\* n (factorial (- n 1)))

; caso recursivo

)

)

(factorial 4)

;esto nos devolvería 24=4\*3\*2\*1

## Lógico

En este paradigma la lógica representa conocimiento el cuál es manipulado mediante inferencias. A diferencia de los otros paradigmas, en éste hay que especificar qué hacer y no cómo hacerlo, por ello son llamados lenguajes declarativos.

Se basa en el hecho que un programa implementa una relación antes que una correspondencia. Debido a que las relaciones son mas generales que las correspondencias (identificador - dirección de memoria), la programación lógica es potencialmente de más alto nivel que la programación funcional o la imperativa.

La programación lógica empieza su proceso a partir de un conjunto de reglas (axiomas) e inferencias para comprobar nuevas proposiciones que sean relevantes.

El proceso está basado en reglas lógicas de primer orden. El lenguaje Prolog es el mayor representante de este paradigma de programación.

## Ejemplo 3.3

## Ejemplo en Prolog

;Se definen las relaciones (reglas)

```
padrede('juan', 'maria').          % juan es padre de maria
padrede('pablo', 'juan').          % pablo es padre de juan
padrede('pablo', 'marcela').        % pablo es padre de marcela
padrede('carlos', 'debora').        % pablo es padre de juan
hijode(A,B) :- padrede(B,A).       % A hijo de B, B padre de A
abuelode(A,B) :- padrede(A,C), padrede(C,B).
hermanode(A,B) :- padrede(C,A) , padrede(C,B), A \== B.
familiarde(A,B) :- padrede(A,B).
familiarde(A,B) :- hijode(A,B).
familiarde(A,B) :- hermanode(A,B).
```

**Ejemplo 3.3**

; Se ejecutan preguntas al lenguaje para evaluar su  
; conocimiento adquirido

?- hermanode('juan', 'marcela').

yes

?- hermanode('carlos', 'juan').

no

?- abuelode('pablo', 'maria').

yes

?- abuelode('maria', 'pablo').

no

## Declarativo

Este paradigma está basado en el desarrollo de programas especificando, describiendo o declarando un conjunto de instrucciones (afirmaciones, condiciones, ecuaciones, proposiciones, restricciones o transformaciones) que describen el problema y detallan su solución.

En la programación declarativa se describe la lógica de computación necesaria para resolver un problema sin describir un flujo de control de ningún tipo, es decir, no es necesario definir algoritmos puesto que se detalla la solución del problema en lugar de como llegar a esa solución.

Es más complicado de implementar que el paradigma imperativo, tiene desventajas en la eficiencia, pero ventajas en la solución de determinados problemas.

No existen asignaciones destructivas y, por tanto, las variables son utilizadas con transparencia referencial, es decir, una expresión puede ser sustituida por el resultado de ser evaluada en el programa sin alterarlo semánticamente.

Algunos lenguajes declarativos son Haskell, Maude, Prolog, SQL.

## Ejemplo en SQL

```
SELECT  
    Customers.FirstName,  
    Customers.LastName,  
    SUM(Sales.SaleAmount) AS SalesPerCustomer  
FROM  
    Customers JOIN Sales  
ON  
    Customers.CustomerID = Sales.CustomerID  
GROUP BY  
    Customers.FirstName, Customers.LastName
```

## Estructurado

Este paradigma divide el código en bloques, estructuras que pueden o no comunicarse entre sí.

El teorema del programa estructurado, demostrado por Böhm-Jacopini, dicta que todo programa puede desarrollarse utilizando únicamente 3 instrucciones de control: secuencia, selección e iteración.

**Secuencial:** Una instrucción no se ejecuta hasta que termina la anterior.

**Selección:** Permite que el programa se bifurque en una u otra instrucción según se cumpla o no una condición lógica.

**Iteración:** Bucles o ciclos de control que ejecutan una secuencia de instrucciones mientras se cumple la condición establecida.

**Ejemplo 3.5**

## Ejemplo en C

```
#include <stdio.h>

int main()
{
    printf("Hola mundo");
    return 0;
}
```

## Orientado a objetos

Este paradigma de programación facilita la creación de software de calidad debido a que potencializa el mantenimiento, la extensión y la reutilización del software generado.

La programación orientada a objetos trata de amoldarse al modo de pensar del ser humano y no al de la máquina. Esto es posible gracias a la forma racional con la que se manejan las abstracciones que representan las entidades del dominio del problema y a propiedades como la jerarquía o el encapsulamiento.

El elemento básico de este paradigma no es la función (como en la programación estructurada), sino un ente denominado objeto.

Un objeto es la representación de un concepto para un programa, y contiene toda la información necesaria para abstraer dicho concepto: los datos que describen su estado así como las operaciones que pueden modificar dicho estado y determinan las capacidades del objeto.

Algunos lenguajes que manejan este paradigma son Simula, Smalltalk, C++, Java, Visual Basic .NET.

**Ejemplo 3.6**

## Ejemplo en C#

```
using System;
```

```
namespace HelloNameSpace {  
    public class HelloWorld {  
        static void Main(string[] args) {  
            Console.WriteLine("Hola Mundo!");  
        }  
    }  
}
```

### 3.2 Conceptos manejados en la programación orientada a objetos.

La programación orientada a objetos (POO) es un modelo o paradigma de programación que utiliza objetos, que se comunican a través de mensajes, para resolver algún problema específico.

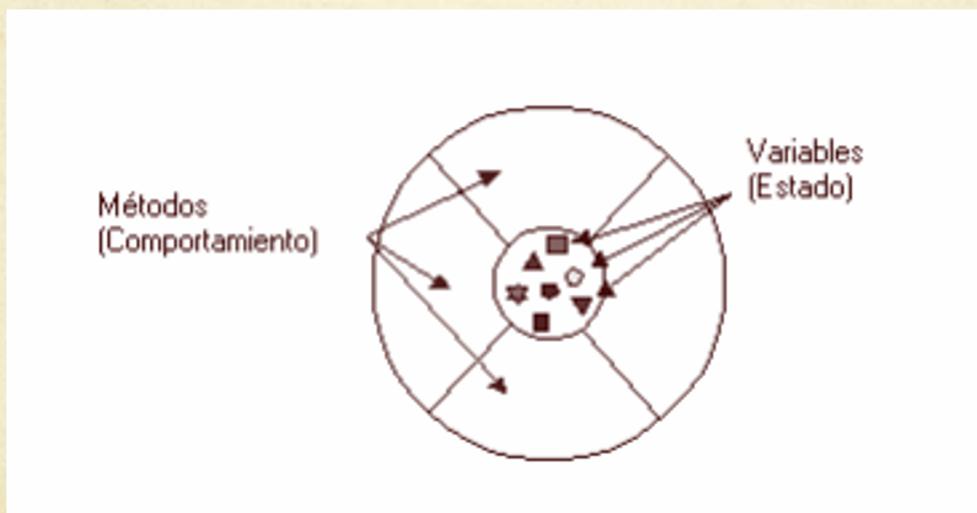
La idea general de este tipo de programación es abordar el problema como se haría en el mundo real, es decir, cada objeto tiene una tarea determinada y los otros objetos pueden utilizarlas.

## Objetos

Un objeto, también conocido como instancia, es el ente básico en la programación orientada a objetos, es decir, un programa orientado a objetos se compone, exclusivamente, de objetos.

Un objeto posee ciertas características (propiedades) y es capaz de llevar a cabo ciertas operaciones o funcionalidades (métodos).

Un objeto es la representación en un programa de un concepto (calculadora, cuenta bancaria, farmacéutica, etc.) y contienen toda la información necesaria para abstraer ese concepto: datos que describen sus atributos así como las operaciones que pueden realizarse sobre estos atributos.



## Un objeto consta de:

- **Tiempo de vida:** La duración de un objeto en un programa siempre está limitada en el tiempo. La mayoría de los objetos sólo existen durante una parte de la ejecución del programa. Los objetos son creados mediante un mecanismo denominado **instanciación**, y cuando dejan de existir se dice que son **destruidos**.
- **Estado:** Todo objeto posee un estado, definido por sus atributos. Con él se definen las propiedades del objeto, y el estado en que se encuentra en un momento determinado de su existencia.

- **Comportamiento:** Todo objeto presenta una funcionalidad, definida por sus métodos, para que el resto de objetos que componen los programas puedan interactuar con él.

Es posible que un objeto desee exponer alguna de sus variables miembro o proteger otras de sus propios métodos o funciones. En la programación orientada a objetos existen diferentes niveles de protección de las variables y de las funciones miembro para casos como éste. Los niveles de protección determinan qué objetos y clases pueden acceder a qué variables o a qué métodos.

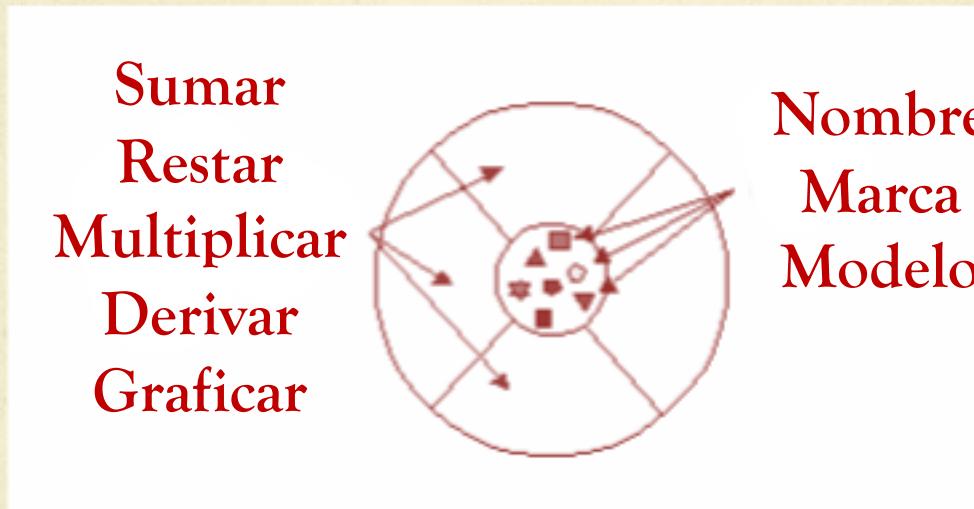
De cualquier forma, el hecho de encapsular las variables y las funciones miembro relacionadas proporciona dos importantes beneficios a los programadores de aplicaciones:

1) Capacidad de crear módulos: El código fuente de un objeto puede escribirse y mantenerse independiente del código fuente del resto de los objetos. De esta forma, un objeto puede pasarse fácilmente de una parte a otra del programa.

**2) Protección de información:** Un objeto tiene una interfaz pública perfectamente definida que otros objetos podrán usar para comunicarse con él.

De esta manera, los objetos pueden mantener información privada y pueden cambiar el modo de operar de sus funciones miembros sin que esto afecte a otros objetos que usen estas funciones miembro.

Por ejemplo, una calculadora posee ciertos atributos (nombre, marca, modelo, teclas, etc.), así como algunas funcionalidades (sumar, restar, multiplicar, derivar, graficar, etc.).



## Mensajes

En general, un objeto aparece como un componente más de un programa o aplicación que contiene otros objetos.

Los objetos de un programa interactúan y se comunican entre ellos por medio de mensajes. Por ejemplo, cuando un objeto A quiere que otro objeto B ejecute una de sus funciones miembro (métodos de B), el objeto A manda un mensaje al objeto B.

**Un mensaje consta de tres partes: Objeto destinatario, método a ejecutar y parámetros.**

**Los mensajes representan todas las posibles interacciones que pueden realizarse entre objetos.**

**Los objetos no necesitan formar parte del mismo proceso, ni siquiera estar en el mismo equipo, para poder interactuar (enviarse mensajes) entre ellos.**

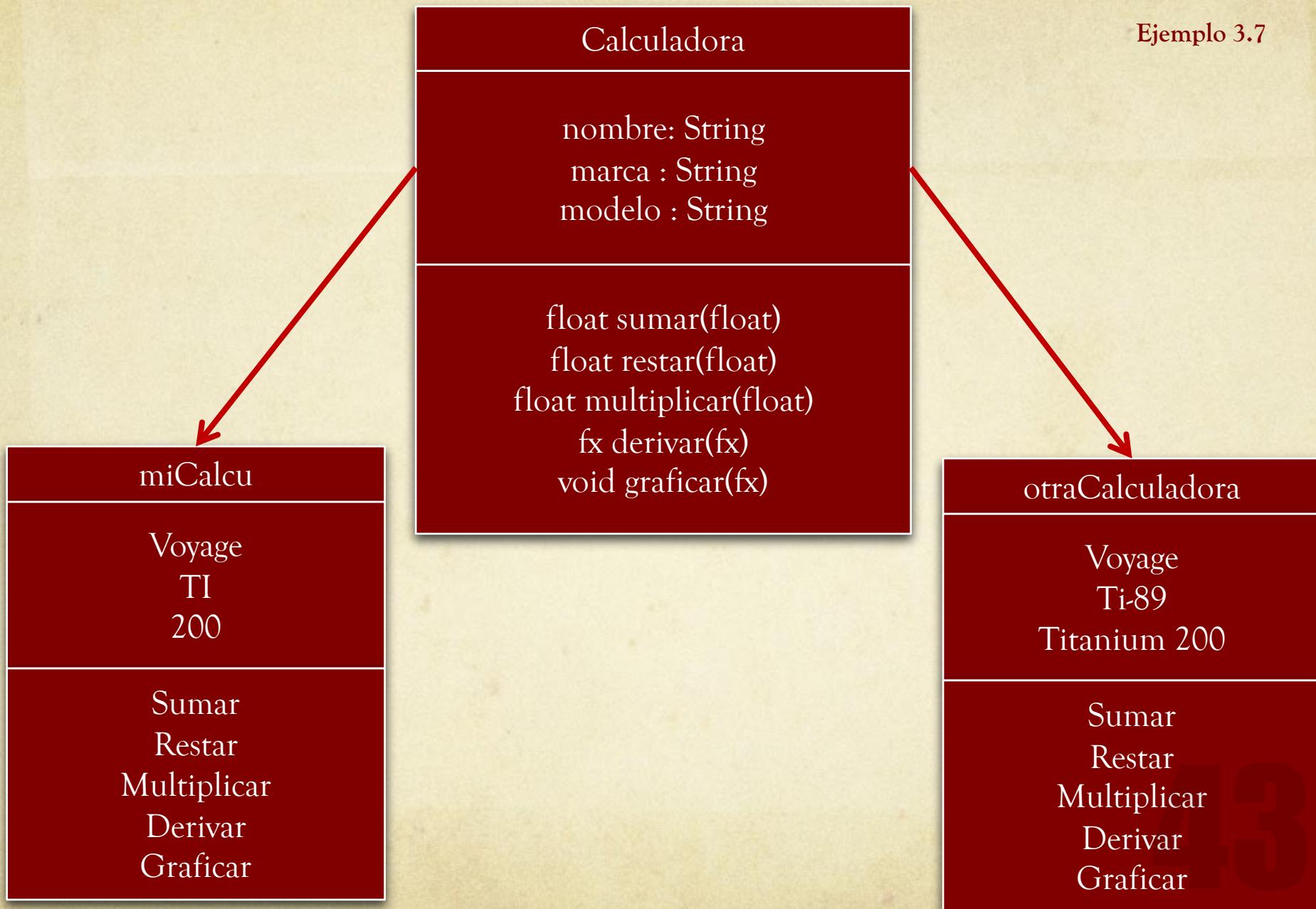
## Clase

Normalmente, en el mundo real existen varios objetos de un mismo tipo, por ejemplo una cámara fotográfica. En la programación orientada a objetos se dice que los objetos del mismo tipo pertenecen a la misma clase.

Las clases son abstracciones que representan a un conjunto de objetos con un comportamiento e interfaz común, es decir, una clase es una plantilla que define las variables y los métodos que son comunes para todos los objetos de un cierto tipo.

Las clases presentan el estado de los objetos a los que representan mediante variables denominadas atributos. Cuando se instancia un objeto el compilador crea en la memoria dinámica un espacio para tantas variables como atributos tenga la clase a la que pertenece el objeto.

Los métodos son las funciones mediante las cuales las clases representan el comportamiento de los objetos. En dichos métodos se modifican los valores de los atributos del objeto y representan las capacidades del objeto.

**Ejemplo 3.7**

## Métodos

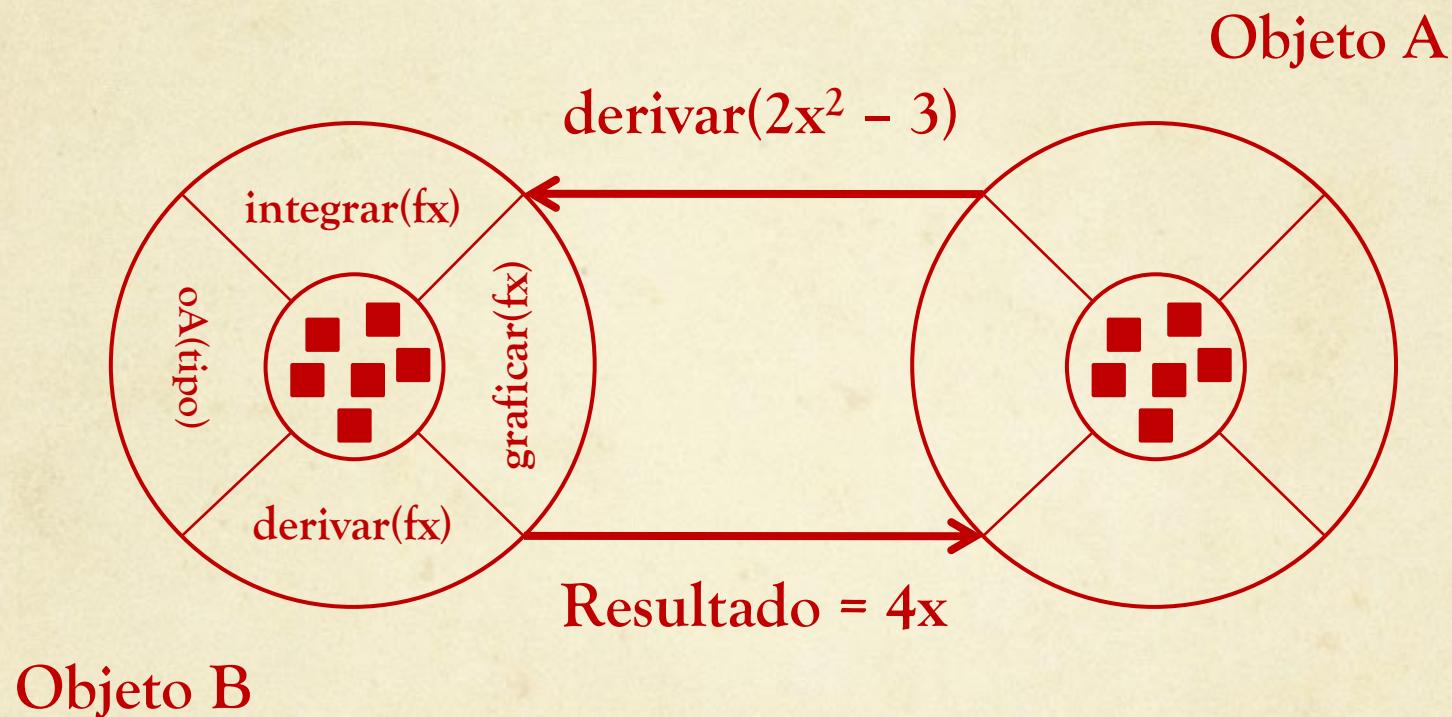
Conjunto de instrucciones a las que se les asocia un nombre de modo que si se desea ejecutarlas, sólo basta con referenciarlas a través de dicho nombre en vez de tener que escribirlas cada vez que se desean ejecutar esas instrucciones.

Estas porciones de código pueden estar asociadas con una clase (métodos de clase o métodos estáticos) o con un objeto (métodos de instancia).

Un método consta de una serie de sentencias para llevar a cabo una acción, un número de parámetros de entrada que regularán dicha acción y, posiblemente, un valor de salida (o valor de retorno) de algún tipo.

El propósito de los métodos es el de proveer un mecanismo para acceder (leer o modificar) los datos privados que se encuentran almacenados en un objeto o clase.

## Ejemplo 3.8

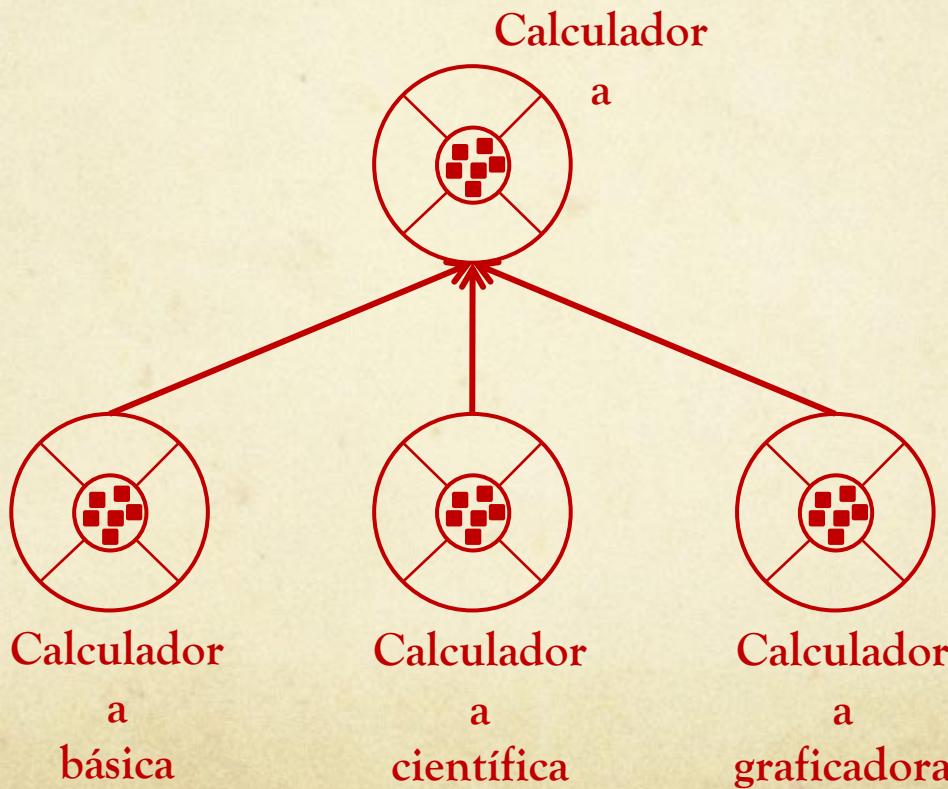


## Herencia

Una de las características básicas de la programación orientada a objetos es la herencia.

La herencia permite definir nuevas clases partiendo de otras ya existentes. Las clases que heredan de otras, obtienen automáticamente todo su comportamiento y, además, son capaces de introducir características propias.

Entonces, es posible definir clases a partir de otras clases ya construidas. Estas nuevas clases son llamadas subclases o clases derivadas y la clase de la que se derivan se llama clase base o superclase.



Cada subclase hereda los estados (variables) y los métodos de la superclase de la cual deriva.

De igual manera, las subclases pueden declarar sus propias variables e implementar sus propios métodos.

Las clases derivadas también pueden sobreescribir los métodos heredados para implementar métodos más especializados con el mismo nombre.

La programación orientada a objetos permite n niveles de herencia, por lo que el árbol de herencias o jerarquía de clases puede ser tan extenso como se necesite. Los métodos y las variables miembro se heredan en forma descendente a través de todos los niveles de la jerarquía.

Por lo anterior, la clase ubicada más abajo de la jerarquía presenta un comportamiento más especializado.

La herencia permite abordar la resolución de un problema de forma organizada, debido a que permite definir una relación jerárquica entre los diferentes conceptos que se estén manejando.

Por lo tanto, se puede afirmar que las clases derivadas o subclases proporcionan comportamientos especializados a partir de elementos comunes que hereda de la clase base.

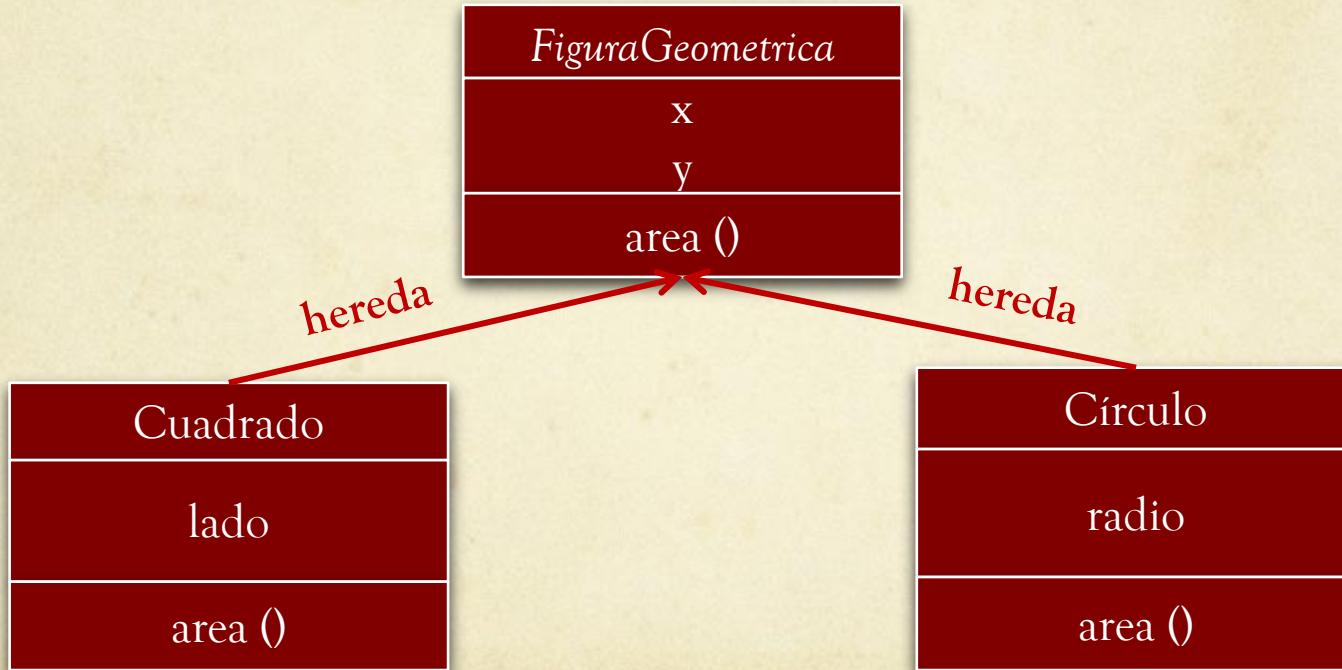
Mediante la herencia, los programadores pueden reutilizar código de una clase base tantas veces como sea necesario.

También, los programadores pueden implementar las superclases abstractas. Este tipo de clases definen e implementan, parcialmente, comportamientos genéricos, sin embargo, la implementación total se lleva a cabo en las clases derivadas.

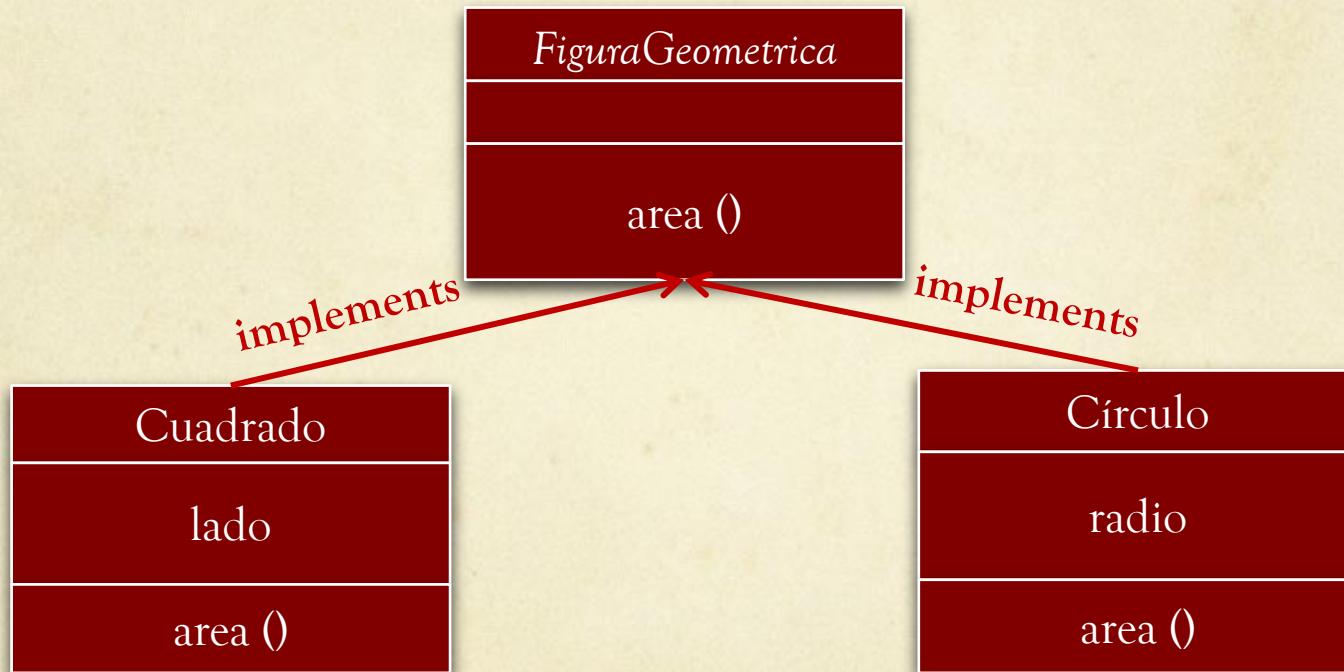
Las clases abstractas sirven como modelo base para la creación de clases derivadas.

Existe dos tipos de abstracciones: las clases abstractas y las interfaces.

Una clase abstracta es una clase que no se puede instanciar, se utiliza, únicamente, para definir subclases. Se utilizan cuando se desea definir una abstracción que englobe objetos de distintos tipos. Hace uso de polimorfismo.



Una interfaz es una clase completamente abstracta, es decir, una clase sin implementación, estas clases no se heredan, se implementan. Al igual que la clase abstracta, se utilizan para definir las clases que los implementan



## Encapsulamiento

Cuando un objeto ejecuta una acción, éste puede hacer uso de uno o más métodos y una o más variables del propio objeto. Sin embargo, lo que importa es el resultado de ejecutar esa acción, no la manera en la que se implementan estos métodos o variables.

La propiedad de ejecutar una acción sin mostrar los mecanismos internos que tiene la clase para obtener su resultado se denomina encapsulamiento.

El encapsulamiento se encarga de mantener ocultos los procesos internos de una acción dándole al usuario acceso a la funcionalidad, no al desarrollo.

La encapsulación permite controlar internamente los errores que pueda generar el usuario, debido a que se controlan los errores internamente.

Por otro lado, al estar oculto el código, se pueden realizar cambios y/o mejoras, siendo éstos transparentes para el usuario.

Para encapsular, ya sean variables o métodos, la programación orientada a objetos maneja distintos niveles de seguridad.

- ◆ **Privado:** variables o métodos visibles solo para la misma clase.
- ◆ **Protegidos:** variables o métodos visibles para esas clases y para sus clases derivadas.
- ◆ **Públicos:** variables o métodos visibles para todos los objetos de la aplicación.

## Polimorfismo

Se refiere a la posibilidad de acceder a un rango de funciones distintas a través de la misma interfaz. Es decir, un mismo identificador puede tener distintas formas (un comportamiento distinto) según el contexto en el que se halle.

El polimorfismo se relaciona con los métodos y se establece mediante tres conceptos: sobrecarga, sobreescritura y enlace dinámico.

## Sobrecarga

Este término hace referencia a un mismo identificador en diversos contextos y con distintos significados, es decir, métodos (funciones) con un mismo nombre pero con funcionalidades distintas.

*Un método consta de una serie de sentencias para llevar a cabo una acción, un número de parámetros de entrada que regularán dicha acción y, posiblemente, un valor de salida (o valor de retorno) de algún tipo.*

La sobrecarga permite crear varios métodos con el mismo nombre y con funcionalidades distintas, es decir, diferentes acciones a realizar, diferentes tipos y número de parámetros.

La sobrecarga permite definir operadores cuyos comportamientos varían de acuerdo a los parámetros que recibe.

Por tanto, se pueden diferenciar varios métodos sobrecargados a través de sus parámetros (ya sea por la cantidad, por el orden, por el tipo, etc.).

**Ejemplo 3.9**

Por ejemplo: Los datos necesarios para realizar un depósito bancario varían según el tipo de depósito a realizar:

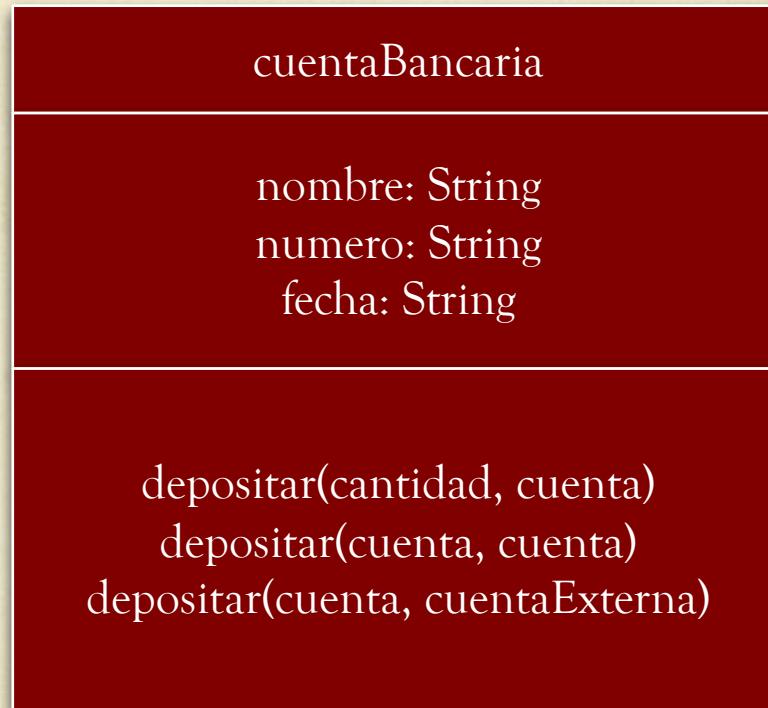
depositar(cantidad, cuenta)

depositar(cuenta, cuenta)

depositar(cuenta, cuentaExterna)

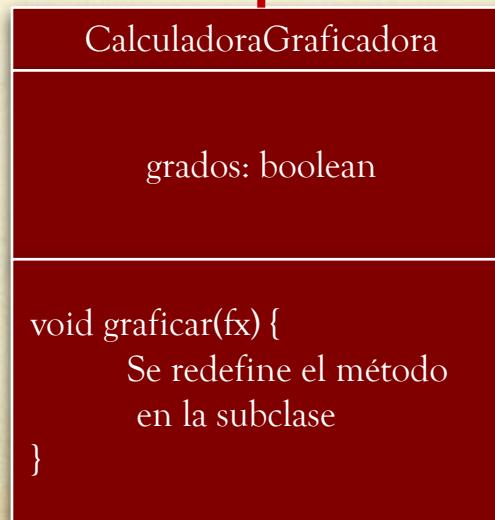
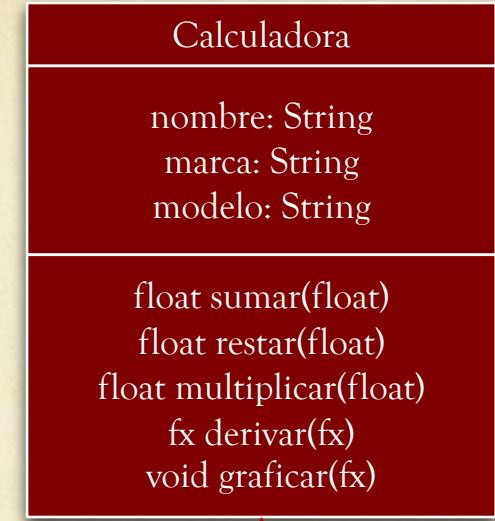
La cantidad puede estar en pesos, en dólares, en euros, en libras esterlinas, etc., o puede ser un cheque.

Las cuentas pueden ser del mismo banco o de diferentes bancos.

**Ejemplo 3.9**

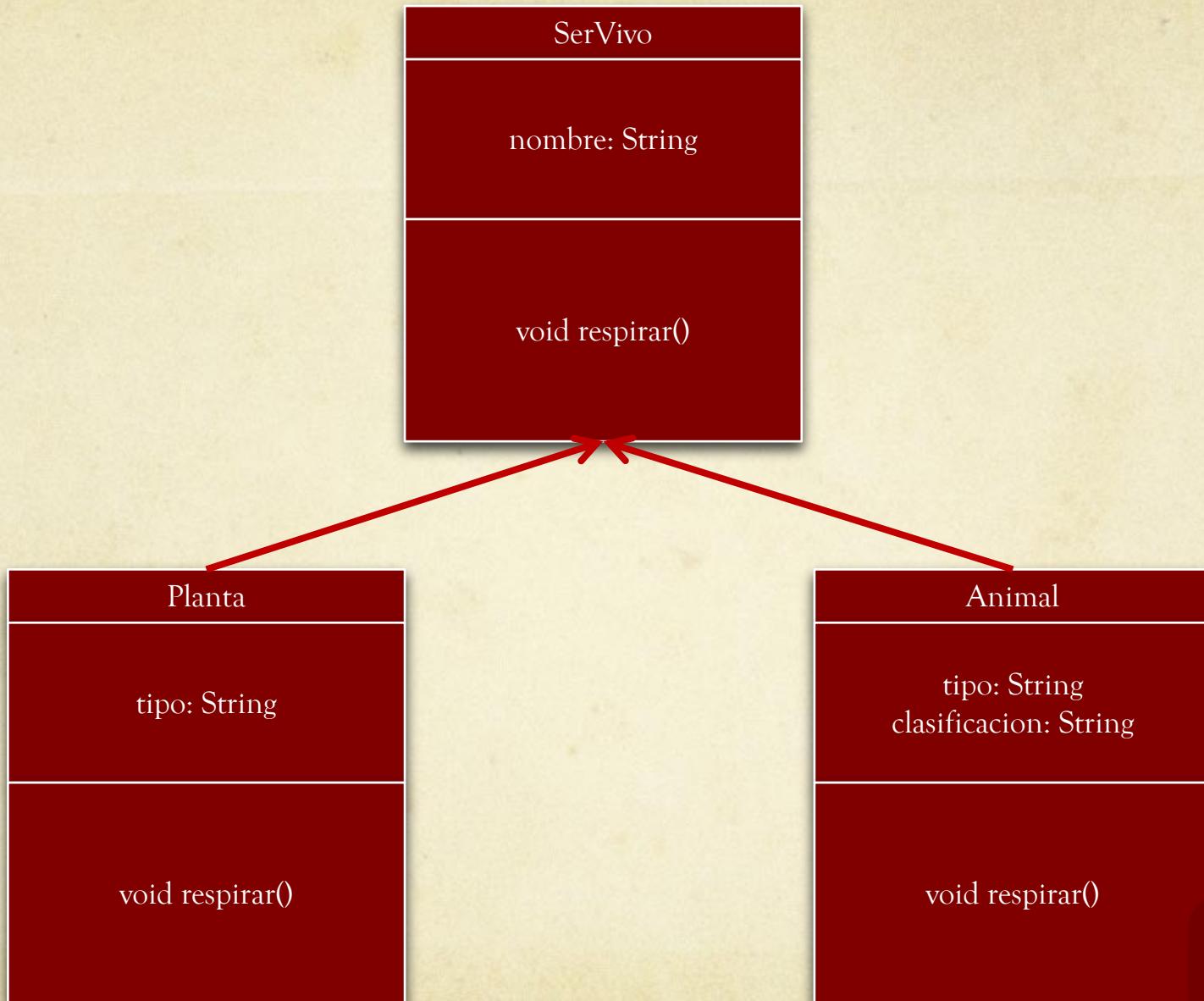
## Sobre-escritura

La sobre-escritura está directamente relacionada con la herencia y se refiere al hecho de redefinir los métodos de la clase base en las subclases.



## Enlace dinámico

El enlace dinámico se presenta cuando existe herencia y sobre-escritura y se refiere al hecho de invocar métodos, obviando el tipo de objeto que invoca el método hasta el momento de ejecutarlo. Esto permite definir elementos de un tipo base e instanciarlos como un tipo heredado.



El enlace dinámico permite realizar un método para invocar el método respirar de cada objeto con base en el tipo de objeto creado.

Se crean los objetos de un tipo específico

...

rosa: Planta

pantera: Animal

respirar (rosa);

respirar (pantera);

...

Se puede crear un solo método para diversos valores de entrada.

```
void respirar(SerVivo sV){  
    sv.respirar ();  
}
```

### 3.1 Diseño de programación orientada a objetos. Notación UML.

El objetivo general al crear software es crear una aplicación que cumpla con los requisitos exigidos.

Para obtener software de calidad es necesario realizar un planteamiento del problema, donde las necesidades del usuario se traducen en requisitos (análisis), para poder realizar el diseño de la solución y, posteriormente, implementar el software (ingeniería de software).

## Ingeniería de software

La ingeniería de software se define como el uso y establecimiento de principios de ingeniería sólidos, a fin de obtener un software que sea económicamente fiable y funcione eficientemente en máquinas reales.

La ingeniería de software provee métodos que indican cómo generar software. Estos métodos abarcan una amplia gama de tareas:

- Planeación y estimación del proyecto.
- Análisis de requerimientos del sistema y software
- Diseño de la estructura de datos, la arquitectura del programa y el procedimiento algorítmico
- Codificación
- Pruebas y mantenimiento (validación y verificación).

Los métodos para la ingeniería de software a menudo presentan un lenguaje orientado especial o una notación gráfica e introducen un conjunto de criterios para la calidad de software.

## Ciclo de vida del software

La ISO (International Organization for Standardization) en su norma 12207 define al ciclo de vida de un software como:

*Un marco de referencia que contiene las actividades y las tareas involucradas en el desarrollo, la explotación y el mantenimiento de un producto de software, abarcando desde la definición hasta la finalización de su uso.*

El ciclo de vida del software está constituido por las fases de análisis, diseño, implementación e instalación.

La calidad de software que se genera se mide en dos grandes rubros:

- ✓ Factores de calidad externos: Correcto, sólido o robusto, confiable, extensible, reutilizable, compatible, eficiente, portable, fácil de usar.
- ✓ Factores de calidad internos: Modular y legible.

## ¿Qué es el análisis?

Es el proceso para averiguar qué es lo que requiere un cliente de un sistema de software (análisis de requisitos) y de definir estos requisitos en forma clara y concisa (especificación de requisitos).

El productor de software se enfrenta al principio de un proyecto con un documento escrito por el cliente, en el cual expresa, en términos de la aplicación, qué se requiere del sistema. Este documento se conoce como declaración de requisitos.

La especificación del sistema debe contener una declaración de las funciones del sistema así como las restricciones con las cuales tendrá que trabajar el productor de software.

También debe contener información adicional como los detalles del hardware que se usará y la capacitación que tendrá que proporcionar el productor, así como una especificación de las herramientas de software especiales que se usarán en el proyecto.

## ¿Qué es el diseño?

Se refiere al manejo de distintas técnicas y principios con el propósito de definir a detalle un producto para poder realizarlo físicamente.

El diseño representa el más alto nivel de abstracción y se puede seguir hasta requisitos más específicos funcionales, de datos o de comportamiento, es decir, permite generar una representación técnica del software a desarrollar.

El diseño debe ser una guía entendible tanto para los desarrolladores del código, como para los que se encargan de realizar pruebas, así como para los encargados del mantenimiento del software.

Por lo tanto, debe proporcionar una idea completa de la funcionalidad y comportamiento del software desde el punto de vista de la implementación.



How the customer explained it



How the Project Leader understood it



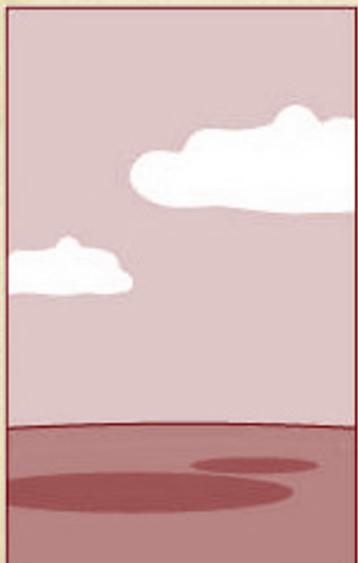
How the Analyst designed it



How the Programmer wrote it



How the Business Consultant described it



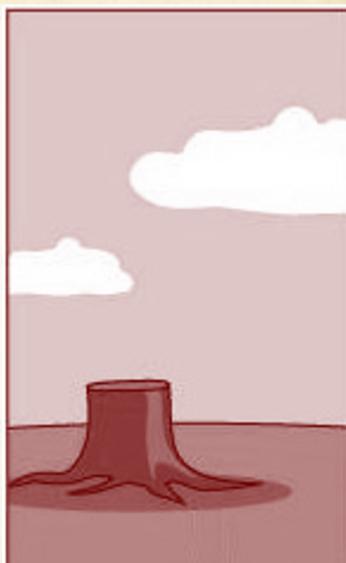
How the project was documented



What operations installed



How the customer was billed



How it was supported

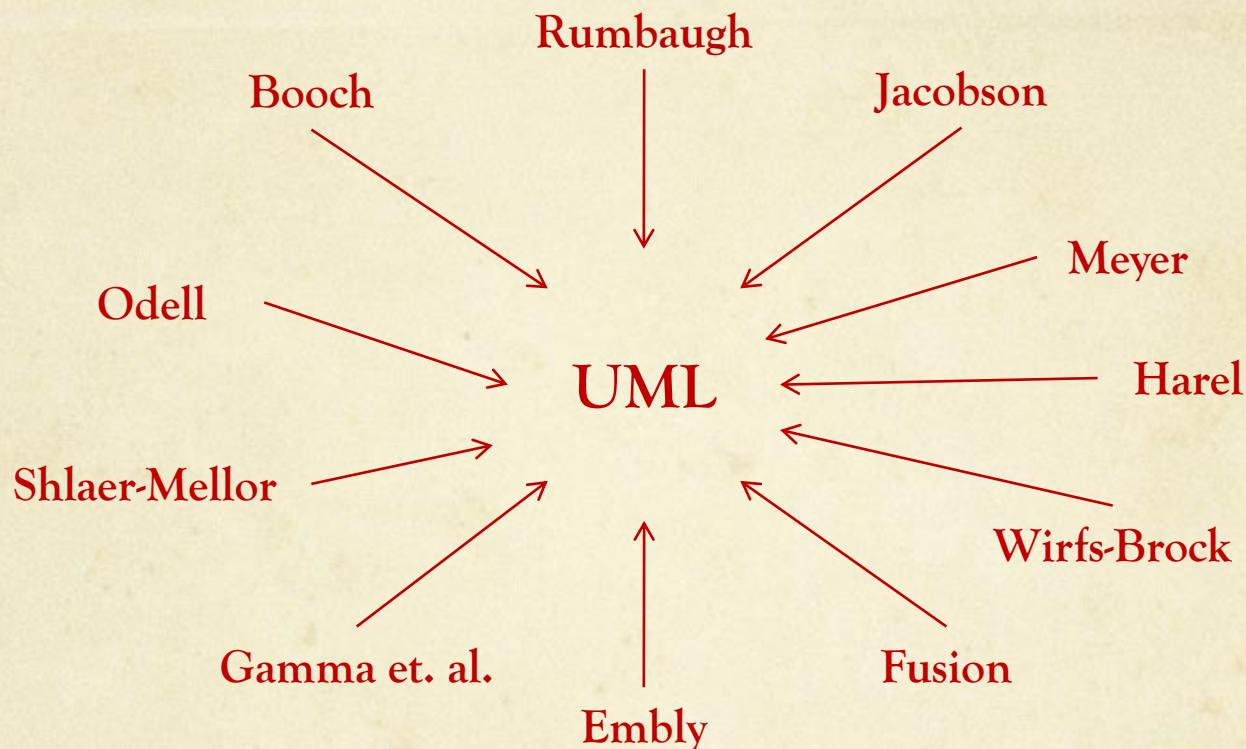


What the customer really needed

## Notación UML



El Lenguaje de Modelado Unificado (UML - Unified Modeling Language) es un lenguaje gráfico que permite visualizar, especificar y documentar cada una de las partes que comprende el desarrollo de software.



Los diagramas UML permiten modelar aspectos conceptuales como procesos de negocios o funcionalidades del sistema, cumpliendo con los siguientes objetivos:

- **Visualizar:** expresa de forma gráfica la solución y/o flujo del sistema.
- **Especificar:** muestra las características del sistema.
- **Construir:** a partir de los modelos realizados se pueden generar soluciones de la aplicación.
- **Documentar:** la generación de diagramas UML son un tipo de documentación muy utilizada.

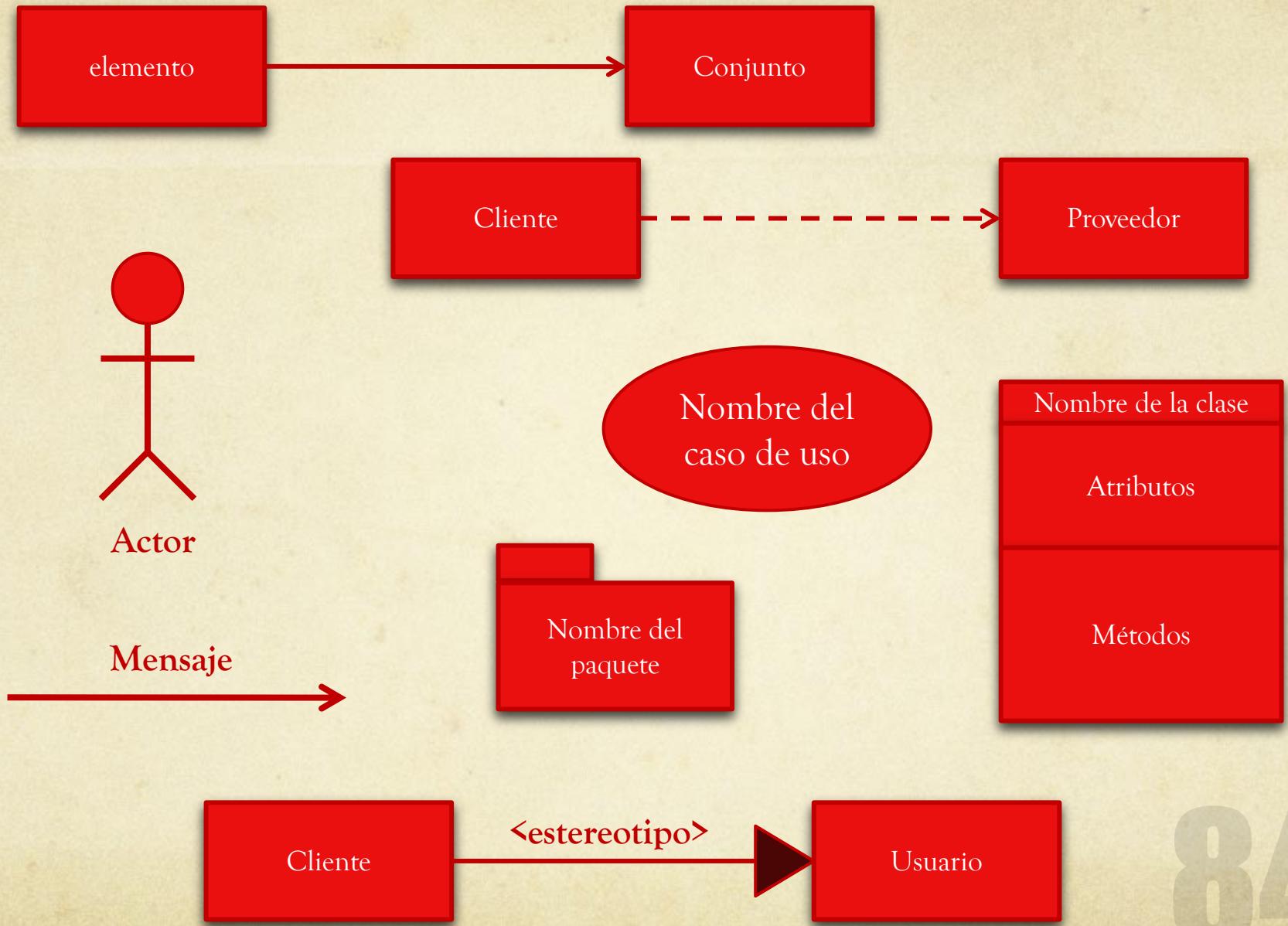
El lenguaje UML está compuesto por tres bloques generales:

- ✓ Elementos: son representaciones de entes reales (usuarios) o abstractos (objetos, acciones, clases, etc.).
- ✓ Relaciones: es la unión de los diferentes elementos.
- ✓ Diagramas: se refiere al conjunto de elementos con sus relaciones.

Los elementos describen los componentes que van a interactuar dentro del diagrama, como pueden ser los actores, las clases, los paquetes, etc.

Las relaciones permiten unir y/o comunicar los diferentes elementos entre sí.

Los elementos y las uniones crean un conjunto de esquemas que pueden dibujar un flujo, una dependencia, una interacción, una comunicación, etc, entre clases. Estos esquemas son llamados diagramas y es la el objetivo final de la parte del diseño.



UML permite crear diferentes tipos de diagramas según las necesidades del analista o los requerimientos del usuario.

- Diagramas de clases
- Diagramas de objetos
- Diagramas de casos de uso
- Diagramas de interacción (de secuencia o de colaboración)
- Diagramas de actividades
- Diagramas de estados
- Diagramas de componentes
- Diagramas de despliegue / distribución

## Diagramas de casos de uso

Estos diagramas definen la manera en la que el usuario (o cliente) interactúa con el sistema.

Los diagramas de casos de uso permiten modelar el comportamiento de un sistema, un subsistema o una clase.

**Los elementos que componen un diagrama de casos de uso son:**

- Actor
- Caso de uso
- Relaciones (uso, herencia y comunicación)

El elemento *actor* modela el rol que juega el usuario en el sistema, es decir, modela la labor que un usuario (cualquiera) realiza frente al sistema.



Nombre del  
caso de uso

El elemento *caso de uso* se refiere a una acción, operación o tarea específica que se realiza tras la orden de un agente externo, es decir, la ejecución de la acción puede venir de un actor o de otro caso de uso.

Existen diferentes tipos de relaciones para los diagramas de casos de uso:

■ **Asociación.** Indica la invocación desde un actor o caso de uso a otra operación (caso de uso). 

■ **Dependencia o instanciación.** Denota la relación entre clases, es decir, cuando una clase instancia otra. 

■ **Generalización.** Permite hacer uso o heredar, dependiendo del estereotipo que se especifique. Esta relación está orientada a casos de uso, únicamente.

 <estereotipo>

89

**Ejemplo 3.10**

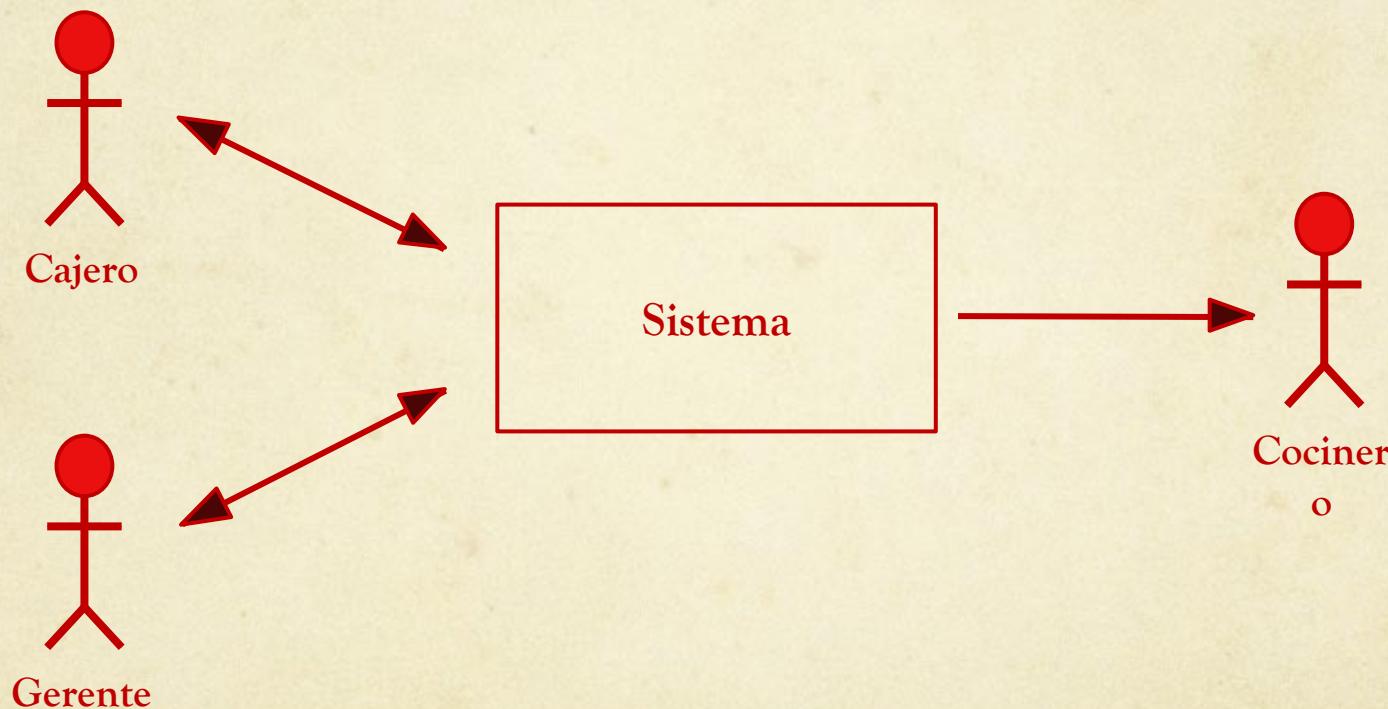
La comida rápida “El rey de las hamburguesas” tiene un sistema para atender a sus clientes, mediante un proceso repetitivo.

Los clientes llegan y pueden ordenar un paquete o cada elemento por separado, el cajero se encarga de tomar la orden. El sistema captura la orden y calcula el monto a pagar por el usuario. En caso de que el cajero se equivoque o el usuario cambie de opinión, el gerente puede cancelar la orden.

Una vez confirmado el pedido y realizado el pago, el sistema envía la solicitud a la cocina para la preparación de los alimentos.

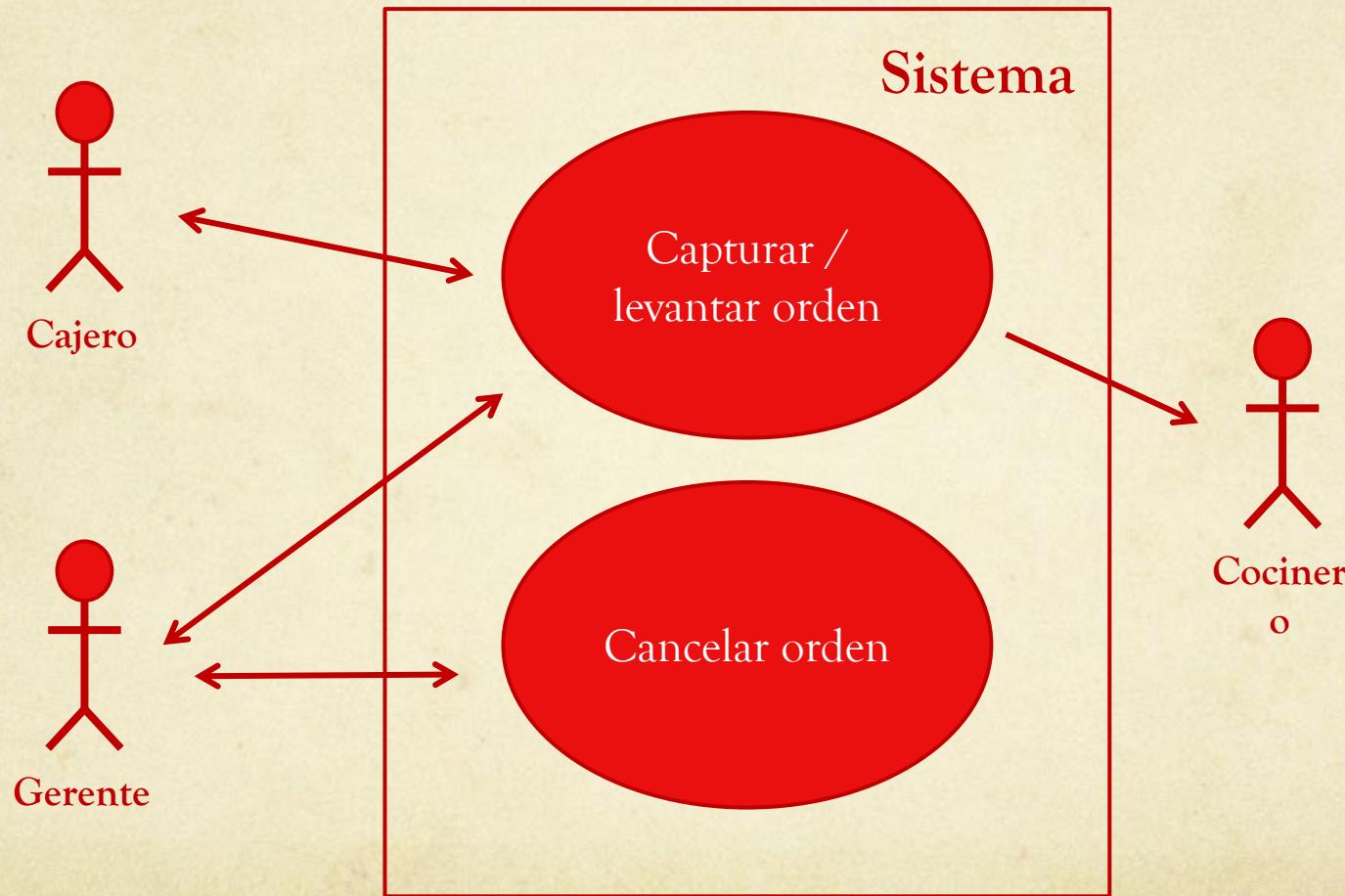
## Ejemplo 3.10

Para realizar los diagramas de casos de uso del sistema anterior primero se identifican a los actores que interactúan con el sistema



## Ejemplo 3.10

También se generan los diagramas las acciones que puede realizar cada actor.



**Ejemplo 3.10****Escenario Capturar / levantar orden****Autor:** Jorge A. Solano**Fecha:****Descripción:** Permite capturar una nueva comanda.**Actor(es):** Cajero, Gerente**Precondiciones:**

**Flujo normal:** El comensal selecciona los elementos que van a componer su orden y se lo hace saber a la persona que lo atiende (cajero o gerente). El encargado ingresa los elementos que conforman la orden en el sistema. Al final de recibir todos los elementos que conforman la orden, el encargado confirma la orden y, de estar correcta, proporciona el monto total del encargo.

Una vez recibido el pago por la orden, se envía una confirmación para que el sistema haga llegar la misma al cocinero.

**Excepciones:** Si el número de elementos que ordena el comensal es mayor al número de elementos que se tiene en bodega, la orden no puede ser surtida.

**Post-condiciones:**

## Escenario Cancelar orden

## Ejemplo 3.10

**Autor:** Jorge A. Solano

**Fecha:**

**Descripción:** Permite cancelar una comanda.

**Actor(es):** Gerente

**Precondiciones:** Se debe tener una comanda capturada en el sistema.

**Flujo normal:** El comensal selecciona un alimento y es agregado a su orden. Posteriormente, el comensal decide cancelar ese alimento de su orden.

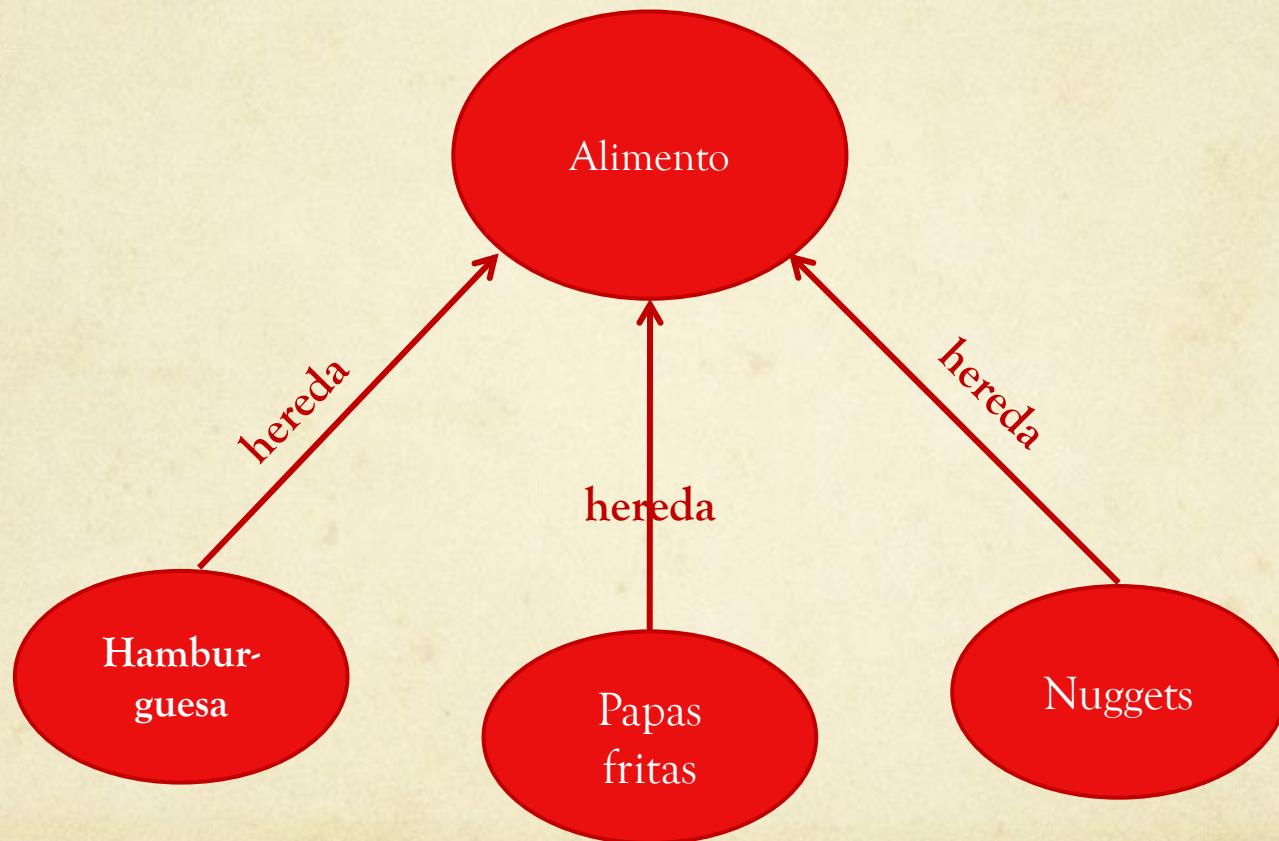
El gerente debe ingresar su número de trabajador para que el sistema valide que la cancelación está autorizada por el mismo.

**Excepciones:** Si la nota ya fue impresa ya no es posible cancelar la orden, debido a que la orden ya fue enviada al cocinero.

**Post-condiciones:**

## Ejemplo 3.10

Una orden puede constar de diferentes alimentos, los cuales pueden tener características comunes:



## Diagramas de clases

Un diagrama de clases sirve para visualizar las relaciones que existen entre las clases que involucran al sistema.

Por lo tanto, los dos elementos principales de este tipo de diagrama son las clases y las relaciones.

La clase es la unidad básica que encapsula toda la información de un objeto y permite modelarlo. Su representación gráfica en UML es:



Existen diferentes permisos de acceso tanto a los atributos como a los métodos:

- **public (+):** el atributo o método es visible para todas las clases, dentro o fuera de el paquete de la clase.
- **private (-):** el atributo o método es visible sólo dentro de la clase donde está declarado.
- **protected (#):** el atributo o método es visible para todas las clases y subclases que pertenecen al mismo paquete donde está declarada la clase.
- **friendly:** indica que el atributo será accesible desde cualquier otra clase dentro de su paquete.

## Cardinalidad

Entre los diagramas, la unión entre clases se lleva a cabo por medio de relaciones. Dentro de las relaciones existe 3 tipos diferentes de cardinalidad (nivel de dependencia) y pueden ser:

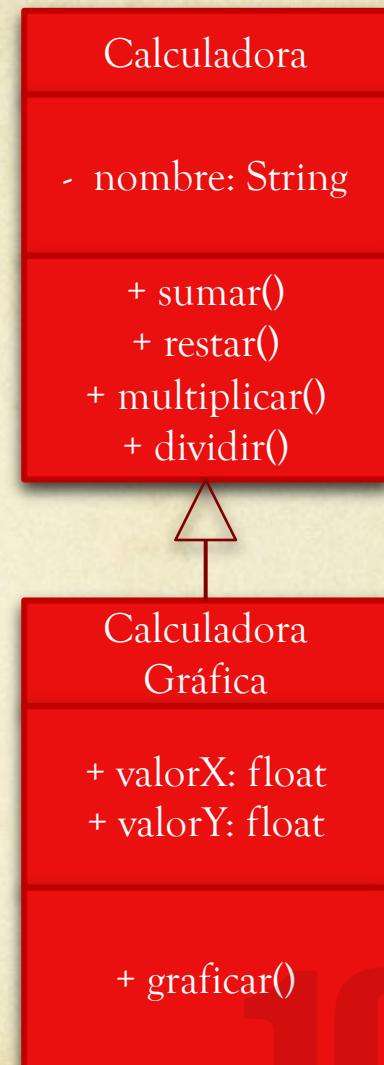
- Uno a muchos: 1 ... \*
- Cero a muchos: 0 ... \*
- Número fijo: n

## Relaciones entre clases

Existen 4 relaciones básicas en los diagramas de clases:  
herencia, agregación, asociación y dependencia o  
instanciación.

## Herencia

Indica la relación de herencia entre la clase y la superclase, es decir, los atributos y métodos que hereda la clase de la superclase.



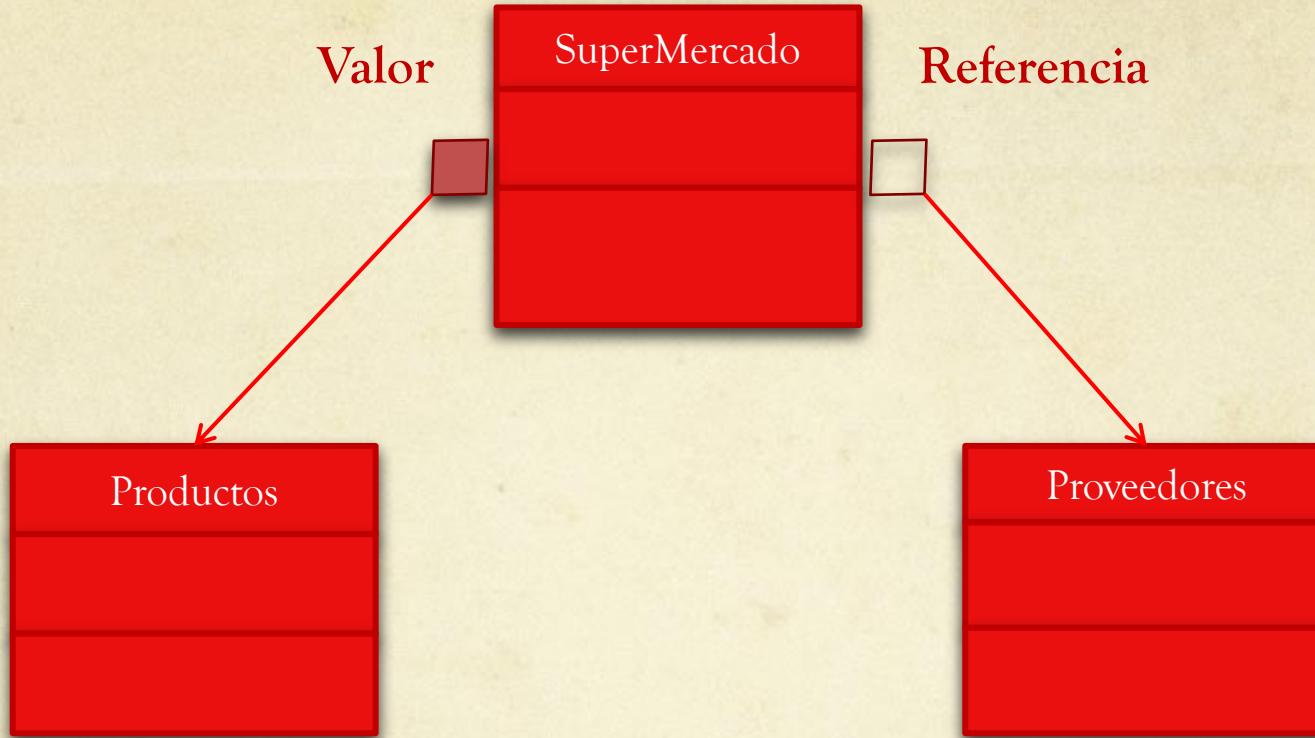
## Agregación

Permite modelar objetos complejos, es decir, aquellos tipos de datos que son instancias de clases definidas por el desarrollador. Se tienen dos tipos de relaciones:

- Por valor: o composición, donde el objeto base construye al objeto incluido, siendo ambos parte del todo.
- Por referencia: o agregación, donde el objeto base utiliza al objeto incluido en su funcionamiento o desempeño.

10

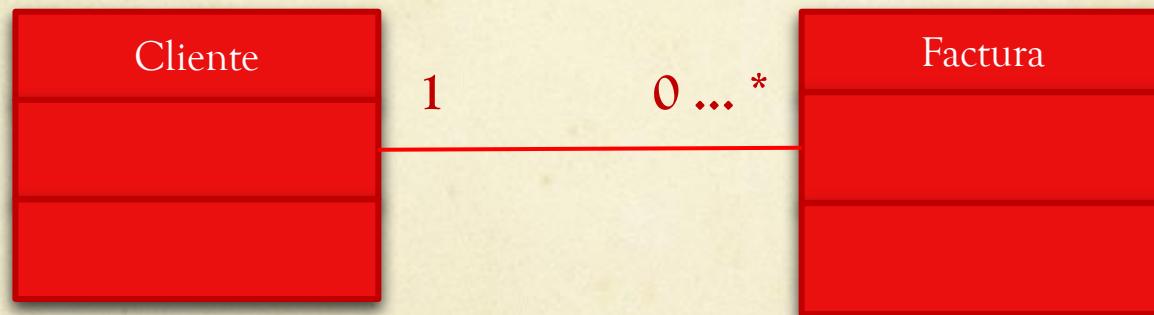
2



Un super mercado posee productos y proveedores. Si el super mercado quiebra, los productos se eliminan con él, sin embargo, los proveedores siguen trabajando con otros clientes.

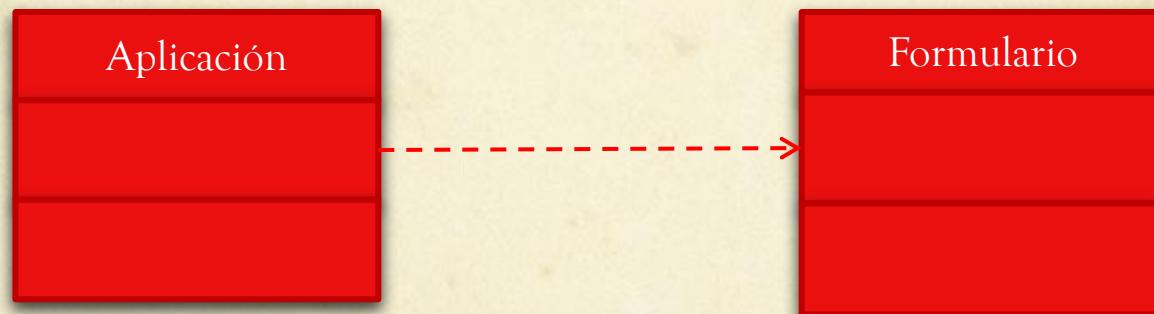
## Asociación

Permite unir objetos que colaboran entre sí. Las uniones permiten establecer la cardinalidad entre las clases.



## Dependencia o instanciación

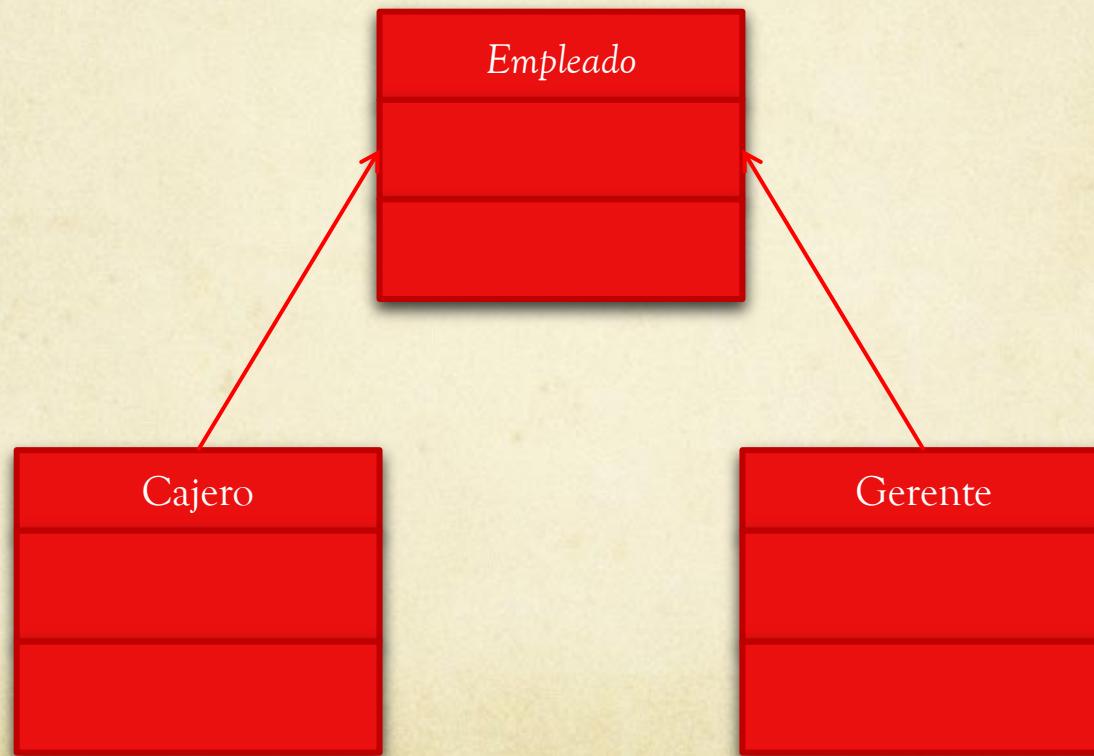
Representa la instanciación entre clases debido a que un objeto es dependiente de otro objeto y al crear uno se tiene que crear el otro. El objeto externo no se almacena dentro del que lo crea.



El objeto ventana crea un objeto tipo formulario, pero el formulario no se almacena en el objeto ventana.

## Abstracciones

Permiten definir atributos y métodos comunes a cierto tipo de entes para, posteriormente, implementarlos y poder definirlos según cada ente quiera. Este tipo de clases se representan con la letra del nombre en cursiva.



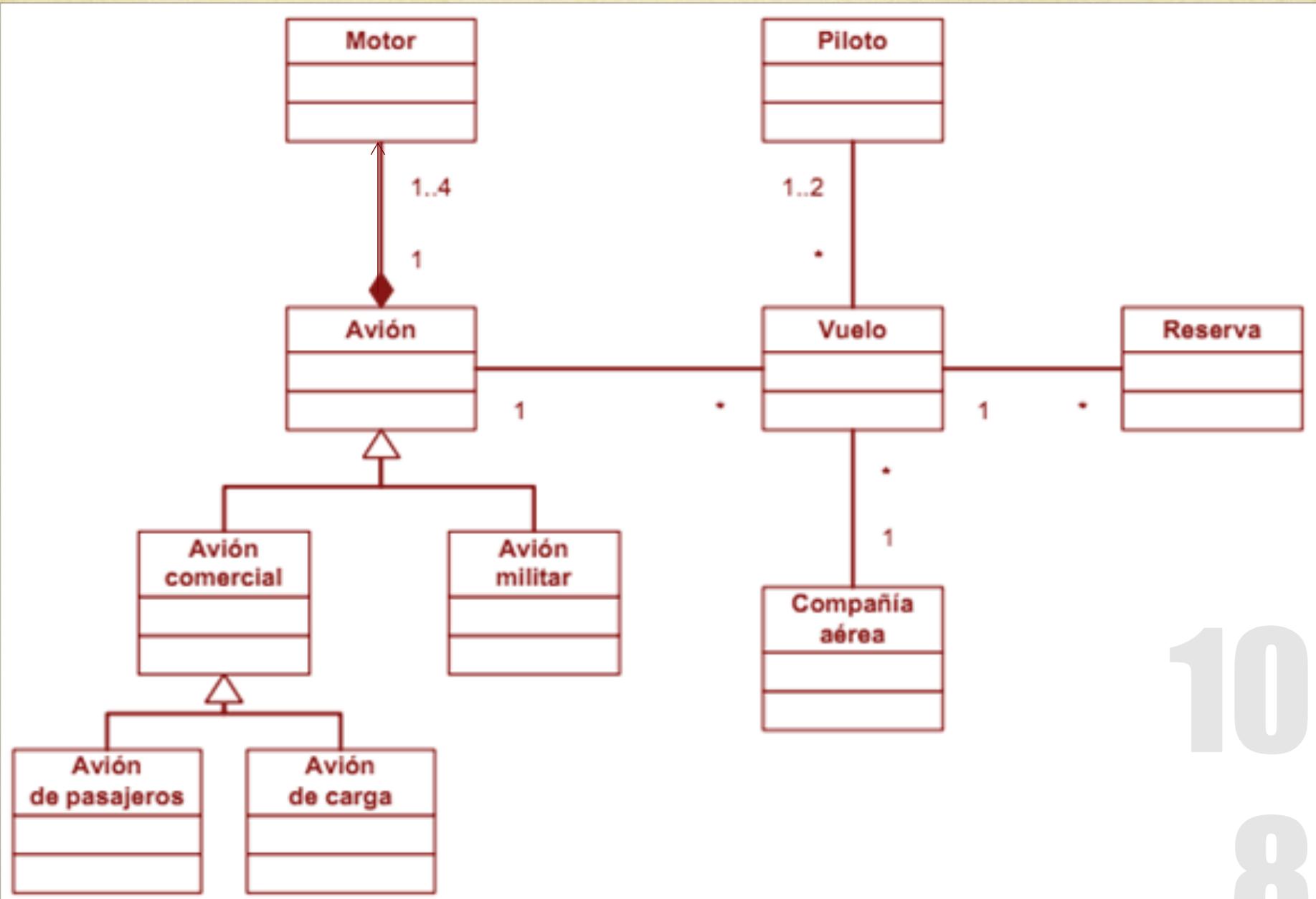
## Clases parametrizadas

Son aquellas clases que necesitan parámetros de entrada al momento de ser instanciadas. Los parámetros se denotan con un recuadro arriba del nombre de la clase.



# Clases que involucran un vuelo

Ejemplo 3.11



10

8

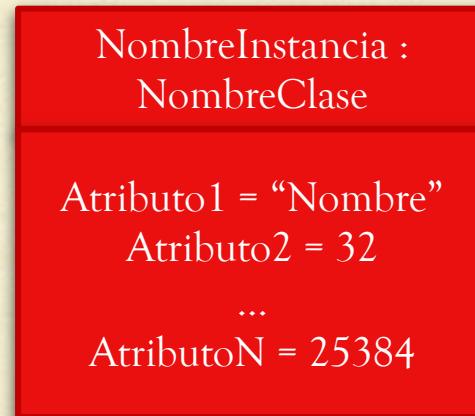
## Diagramas de objetos

Estos diagramas modelan las instancias de clases y se utilizan para describir al sistema en un tiempo (o acción) en particular.

Permiten mostrar los objetos y las relaciones entre ellos en un momento dado, por lo tanto, representa la parte estática de la interacción entre objetos (una situación específica en un momento determinado).

Los elementos utilizados para generar este tipo de diagramas son objetos y asociaciones.

Los objetos se representan con el nombre de la instancia seguido de dos puntos seguido del nombre de la clase. Las propiedades se denotan con el nombre del atributo y el valor asignado.

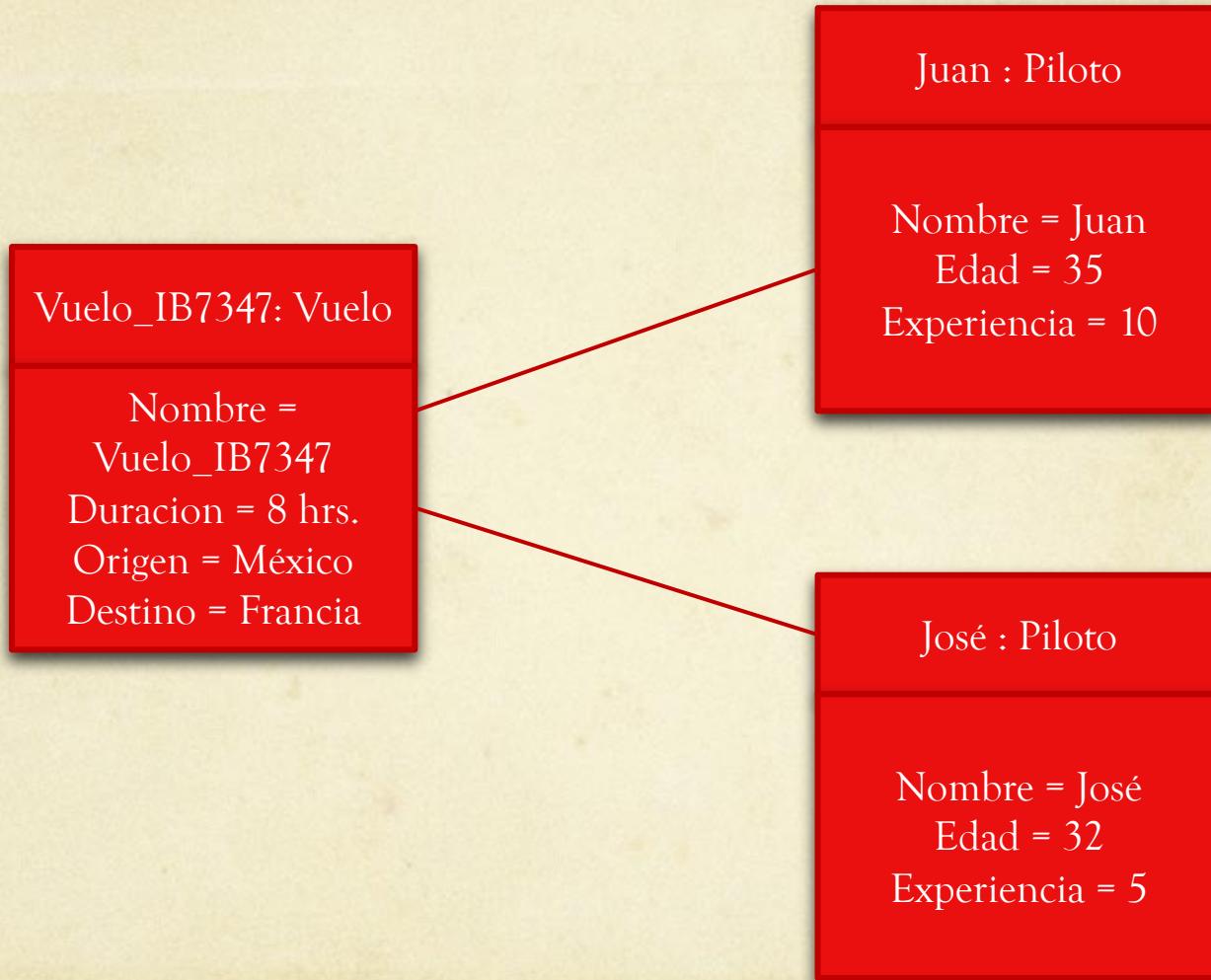


Los objetos se unen utilizando una línea entre cada instancia.

---

**Ejemplo 3.12**

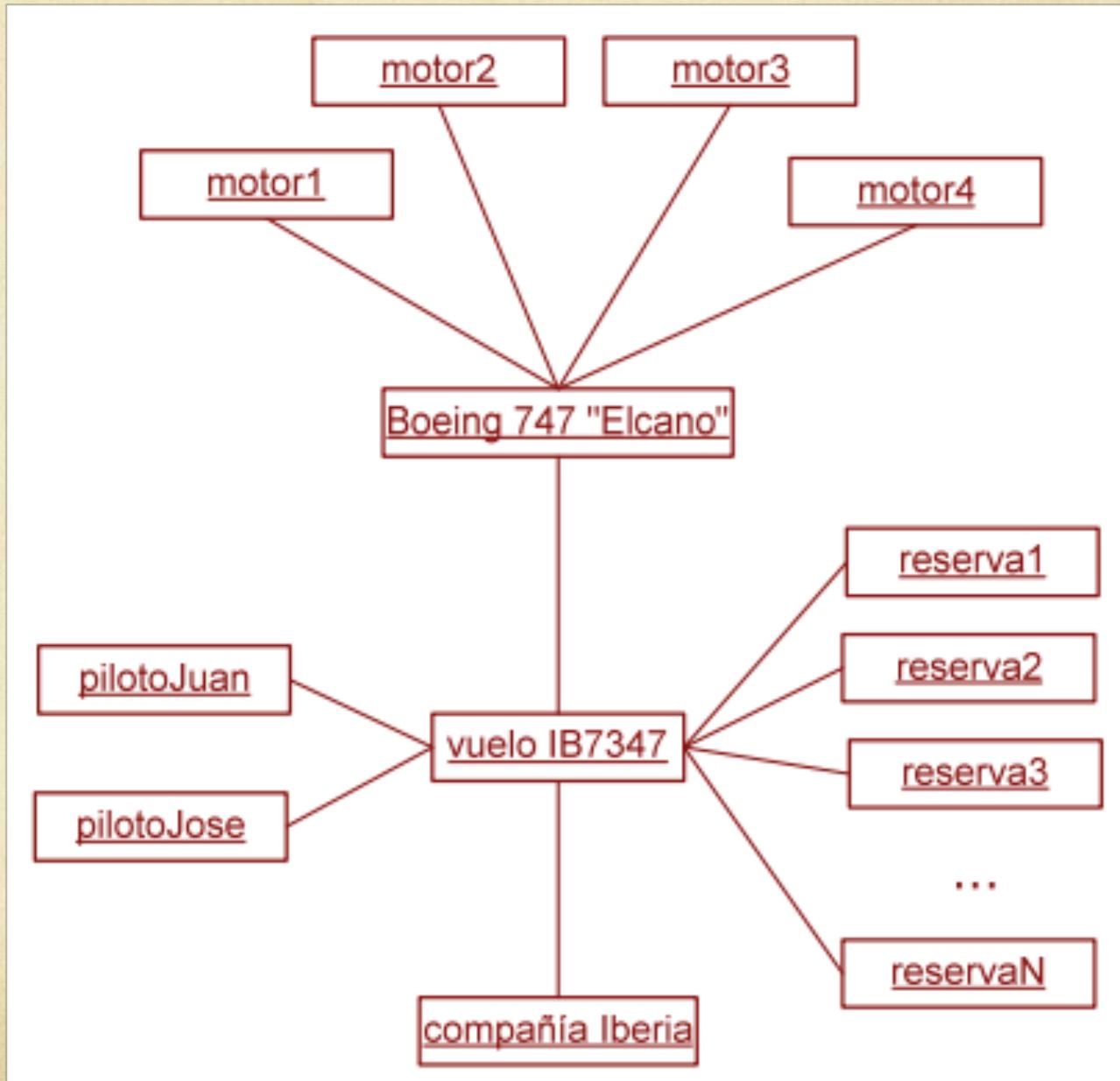
# Objetos tipos Piloto



# Objetos de un vuelo

3.3 Diseño de programación orientada a objetos

Ejemplo 3.12



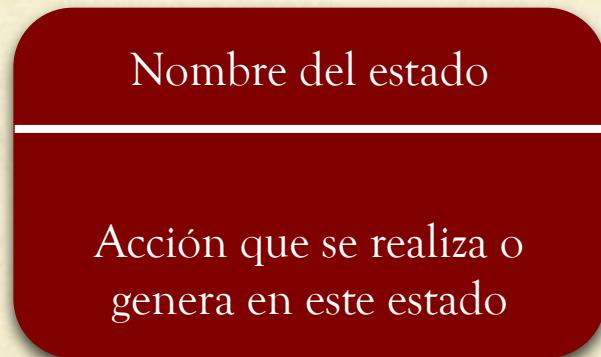
11  
2

## Diagramas de estados

Estos diagramas describen los diferentes estados por los que pasa un objeto durante su tiempo de vida en la aplicación, así como la manera en que cambia el estado del objeto.

Los elementos de estos diagramas son los nodos de entrada y salida del objeto, los estados del objeto y las transiciones entre estados.

Un estado se representa como un rectángulo redondeado que posee dos separaciones, en la primera se escribe el nombre del estado y en la segunda se escriben las acciones de entrada, salida o internas.



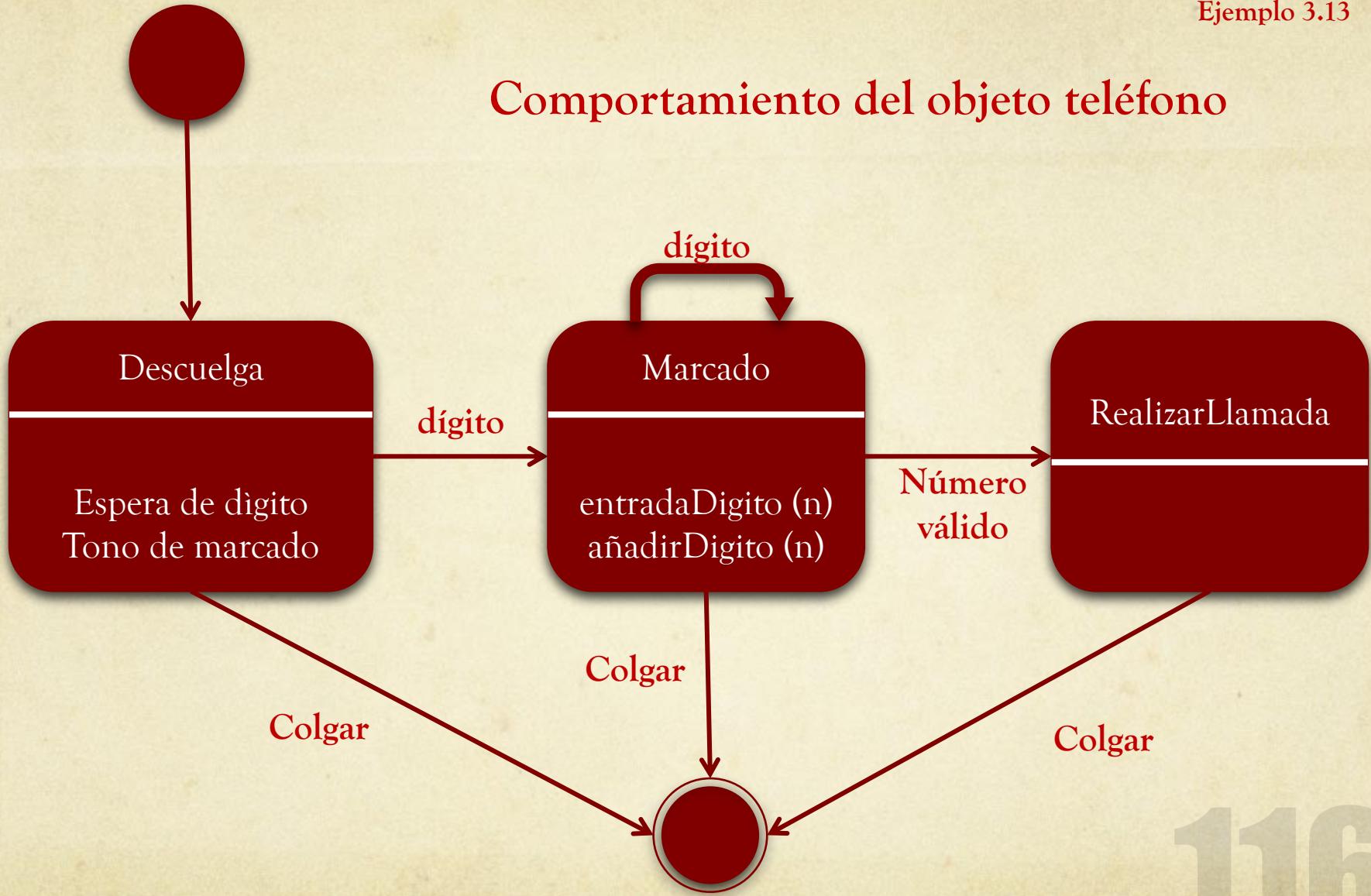
Una transición se representa mediante una flecha desde el estado origen hasta el estado destino.



Los nodos de entrada y salida representan el inicio y fin del diagrama.



## Comportamiento del objeto teléfono



## Diagramas de actividades

Los diagramas de actividades muestran el orden en el que se van realizando tareas dentro de un sistema, es decir, muestran el flujo de control entre actividades.

Por lo tanto, muestran las operaciones que se ejecutan entre objetos.

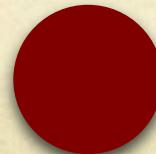
Los componentes básicos de estos diagramas son los estados (de actividad y de acción) y las transiciones.

El estado de actividad es un elemento cuyo flujo de control se compone de otros estados (de actividad o de acción)

El estado de acción ejecuta una acción propia del objeto (método).

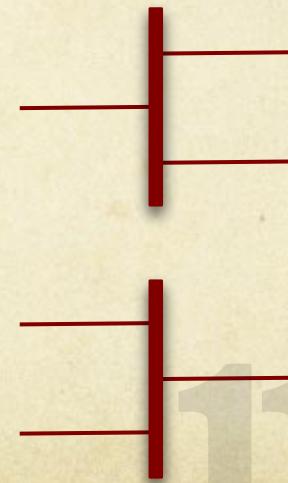
Actividad

También se presentan representaciones para el estado inicial y final.



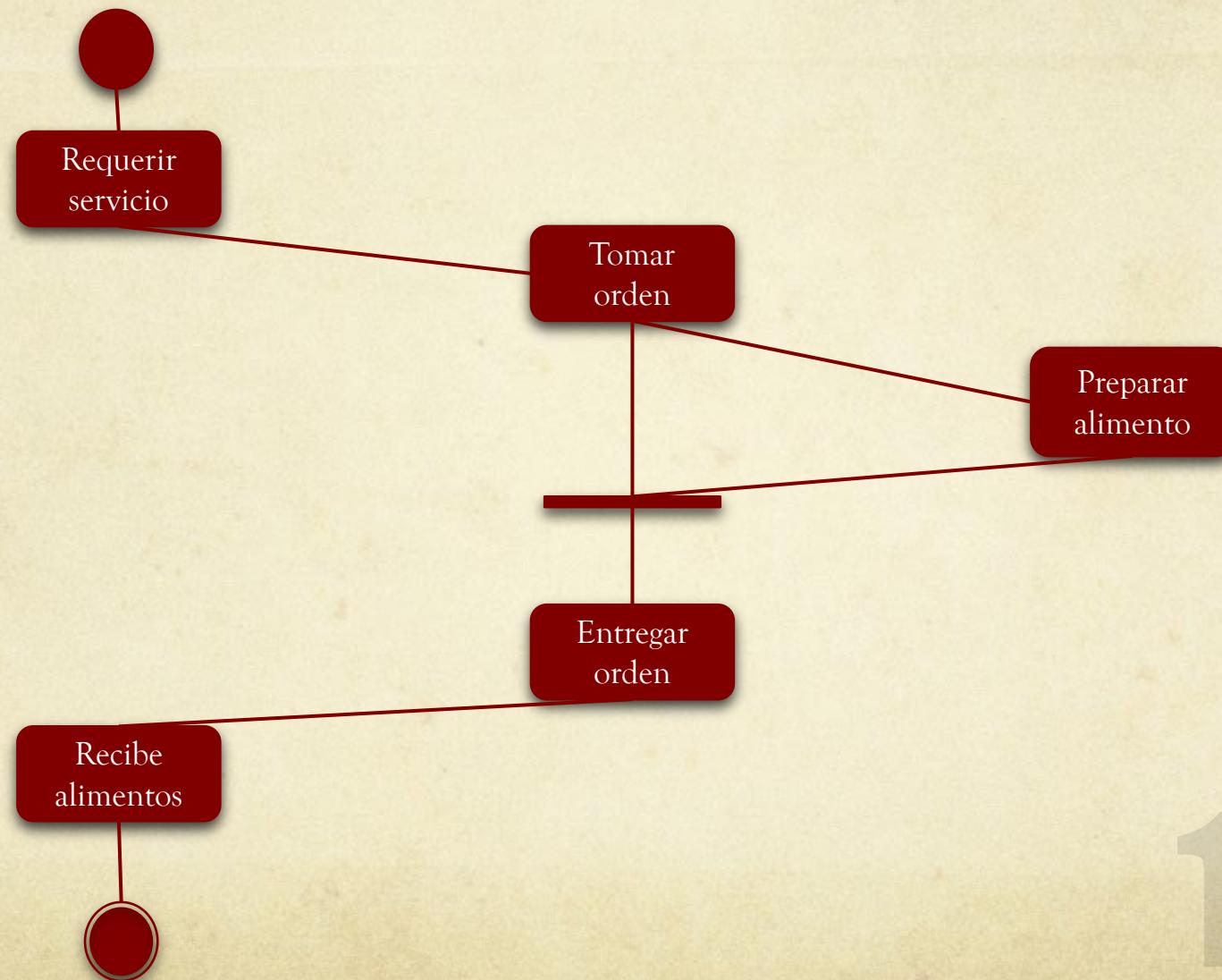
Las transiciones se representan mediante una línea que va del estado inicial al siguiente. Existen 3 tipos de transiciones para estos diagramas:

- Secuencial o sin disparadores
- Bifurcación
- División y unión



# Generar comanda en un restaurante

Ejemplo 3.14

**Comensal****Mesero****Cocinero**

## Diagramas de interacción

Estos diagramas representan la comunicación que se lleva a cabo entre un cliente (actor) o un objeto (clase) cuando se ejecuta una acción en el sistema.

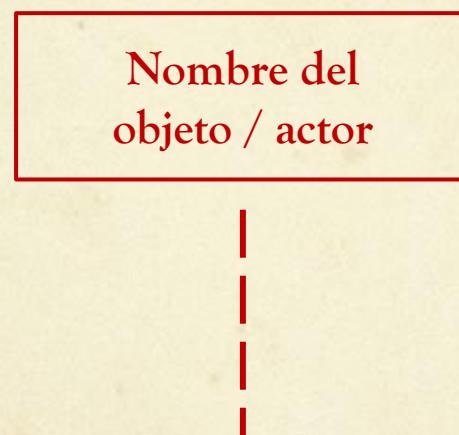
Los elementos básicos de este tipo de diagramas son:

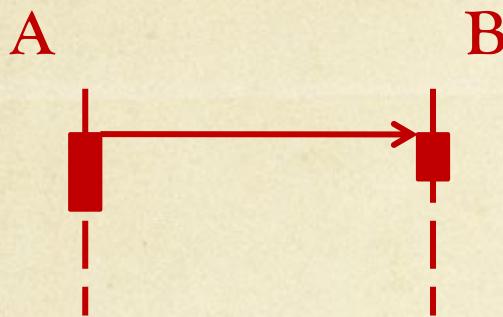
- Objeto o actor.
- Mensaje entre objetos.
- Mensaje de un objeto a sí mismo.

Los diagramas de interacción se dividen en dos tipos: de secuencia o de colaboración.

Los diagramas de secuencia muestran una interacción ordenada de eventos, visualizando los objetos participantes en la interacción así como los mensajes que intercambian éstos.

Los objetos o actores se representan mediante un rectángulo y las llamadas a métodos del objeto son representadas mediante una línea punteada.





Los mensajes entre objetos se representan por una flecha entre un objeto y otro.

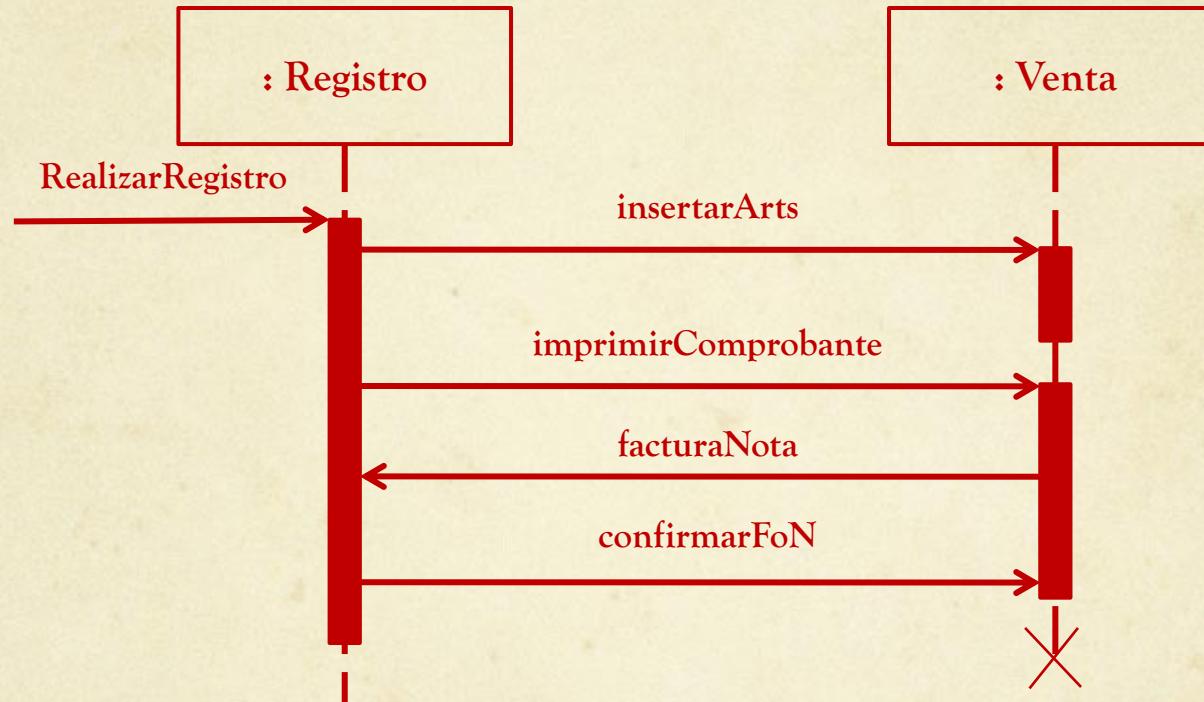
También es posible visualizar las llamadas a métodos propios del objeto, es decir, mensajes al mismo objeto.



124

## Ejemplo 3.15

## Crear nota o factura de una venta



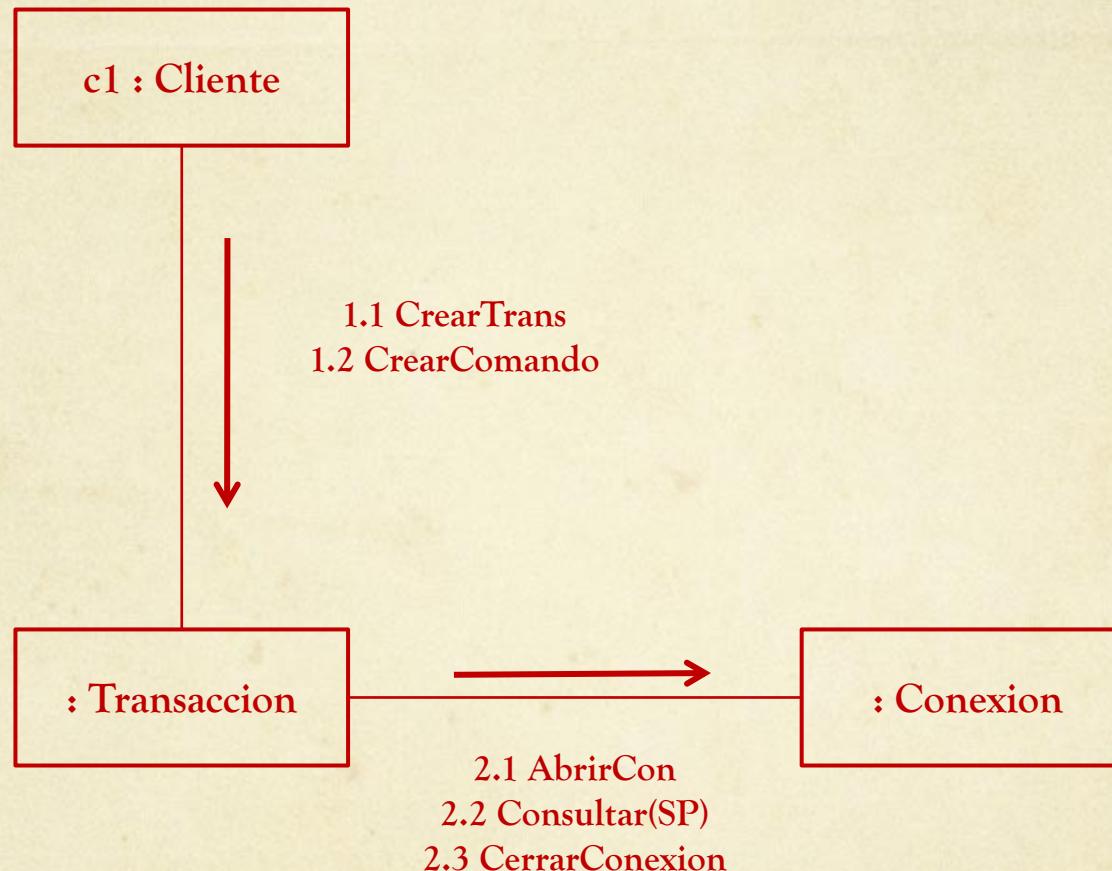
Los diagramas de colaboración muestran la interacción entre varios objetos y los enlaces que existen entre ellos.

La secuencia de los mensajes y los flujos de ejecución concurrentes se determinarán mediante números secuenciales.

Los elementos de este tipo de diagramas son los objetos, los enlaces y los mensajes.

## Ejemplo 3.16

## Consultar datos de una Base de Datos



## Diagramas de componentes

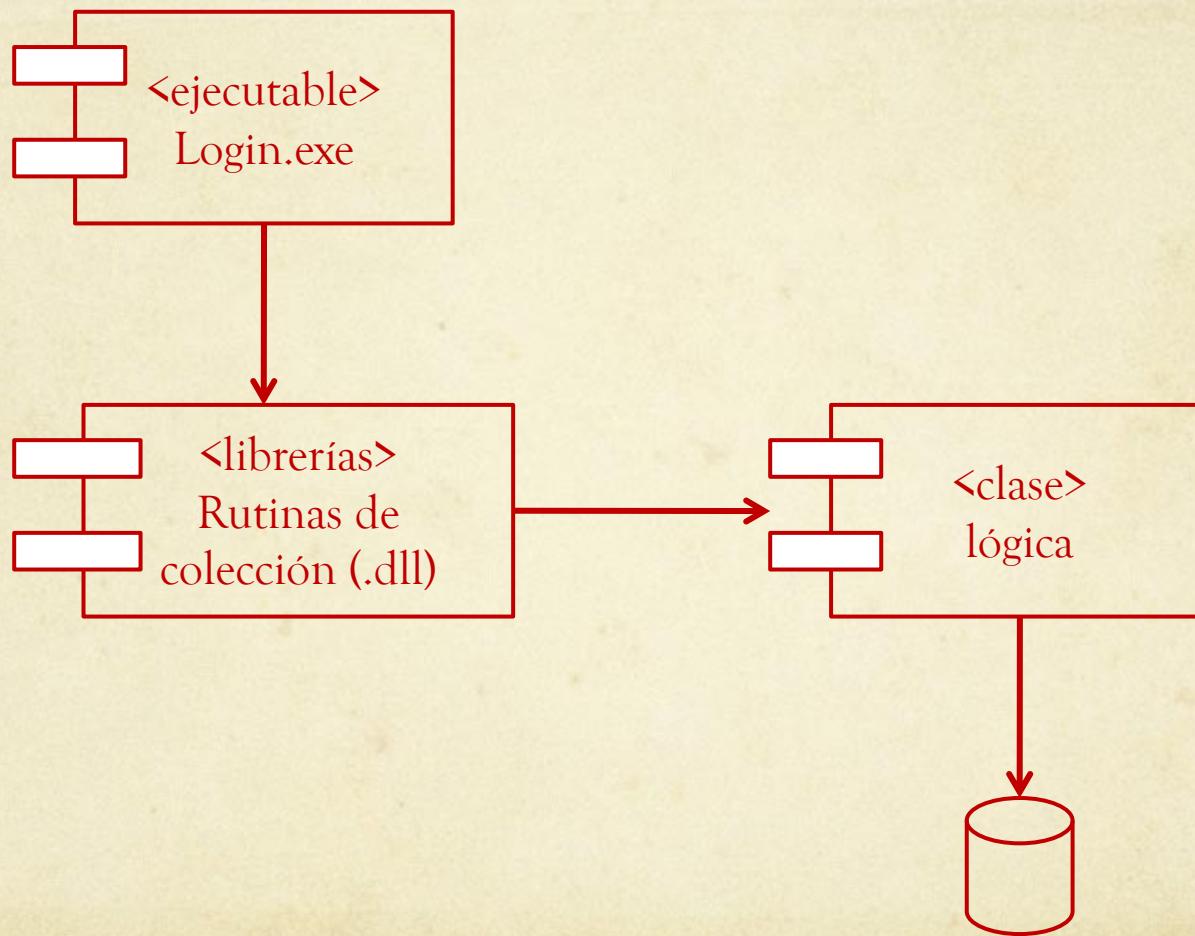
Un componente es una parte física de un sistema, por lo tanto, los diagramas de componentes describen los elementos físicos del sistema así como sus relaciones.

Los elementos básicos son los componentes y los enlaces:



## Ejemplo 3.17

# Mensajero



129

## Diagramas de despliegue / distribución

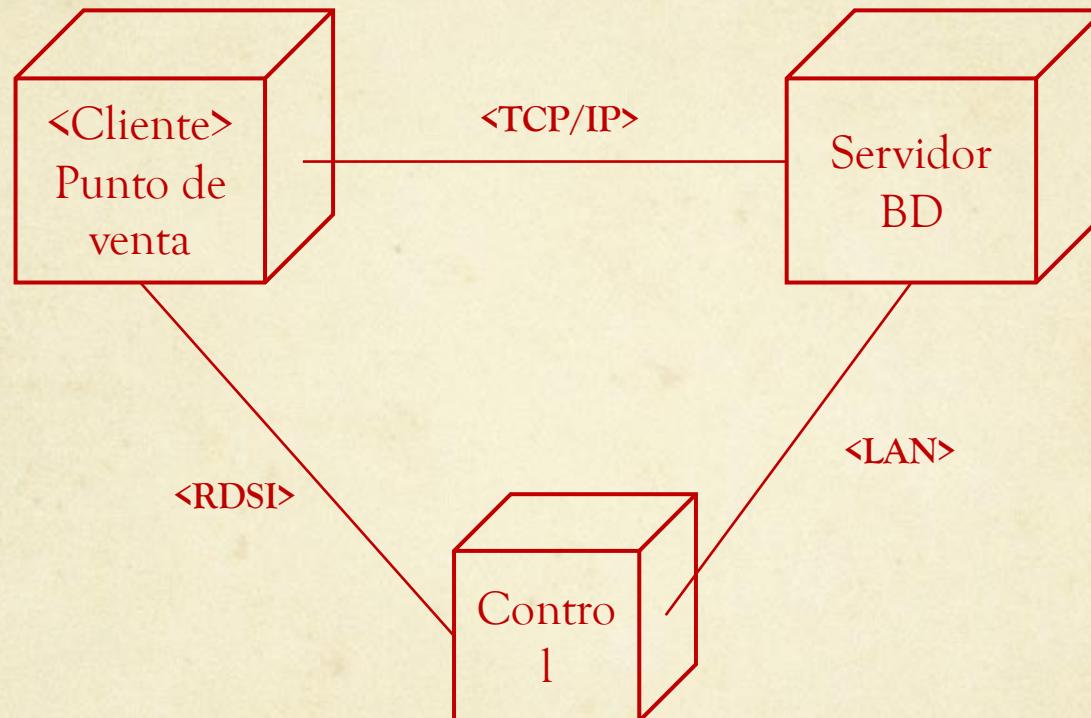
Muestran la disposición física de los distintos nodos que componen un sistema.

Un nodo es un elemento físico que existe en tiempo de ejecución y representa un recurso computacional.

Los nodos se utilizan para modelar la topología del hardware sobre la que se ejecuta el sistema.

## Ejemplo 3.18

## Realizar una compra con tarjeta bancaria



Para realizar diagramas UML existen diversas herramientas, entre los programas más conocidos se pueden encontrar:

- Dia
- Umbrello
- BoUML
- ArgoUML
- IBM Rational Rose
- Poseidon
- SmartDraw
- Enterprise Architect

## Tarea 3.1

## Diagramas UML

Para un mensajero (messenger, skype, google talk, etc.), realizar los siguientes diagramas:

- Los diagramas de casos de uso, de clases y de despliegue de la aplicación.
- El diagrama de estados de una conversación.
- Realizar el diagrama de interacción (secuencia) al establecer una conversación.
- Realizar un diagrama de actividades que describa la acción de iniciar sesión.

Los diagramas se envían en un archivo en formato PDF.

**Tema 3:** Explicar los diferentes paradigmas de programación, así como los conceptos y diseño de la programación orientada a objetos en solución de problemas.

- **Subtema 3.1:** Describir las características principales de los paradigmas de programación.
- **Subtema 3.2:** Explicar las bases de la programación orientada a objetos.
- **Subtema 3.3:** Diseñar programas utilizando notación UML.