palasi.com
IT and management

Análisis y diseño orientados a objetos

Dr. Vicent-Ramon Palasí Lallana
Aurum Solutions.

Presentación de los participantes

- Somos pocos y es interesante que nos conozcamos entre todos.
- Me gustaría conocer su nombre.
- Se pasará una lista donde escribirán su nombre, teléfono y correo electrónico.

Temario del curso

- 1. Introducción al A & D O-O y a UML.
- 2. Análisis orientado a objetos con UML.
- 3. Diseño orientado a objetos con UML.
- 4. Conclusiones finales.

Temario del curso

- 1. Introducción al A & D O-O y a UML.
- 2. Análisis orientado a objetos con UML.
- 3. Diseño orientado a objetos con UML.
- 4. Conclusiones finales.

Parte 1: Introducción al análisis y diseño O-O y UML

- 1.1. Objetivos y metodología del curso.
- 1.2. Conceptos de análisis y diseño.
- 1.3. Modelos de desarrollo.
- 1.4. Utilidad del análisis y diseño.
- 1.5. Conceptos de programación O-O.

Parte 1: Introducción al análisis y diseño O-O y UML

- 1.1. Objetivos y metodología del curso.
- 1.2. Conceptos de análisis y diseño.
- 1.3. Modelos de desarrollo.
- 1.4. Utilidad del análisis y diseño.
- 1.5. Conceptos de programación O-O.

Mail: palasi@palasi.com Web: www.palasi.com

Objetivo del curso

- Enseñar los fundamentos del análisis y diseño orientado a objetos a los asistentes.
- Explicar los artefactos más utilizados de UML (Unified Modeling Language).
- Capacitar a los asistentes para que puedan empezar a realizar el análisis y diseño de programas reales.

Es un curso de introducción

- Enseñar los fundamentos del análisis y diseño orientado a objetos a los asistentes.
- Explicar los artefactos más utilizados de UML (Unified Modeling Language).

¿Por qué?

- Capacitar a los asistentes para que puedan empezar a realizar el análisis y diseño de programas reales.

La razón es la limitación del tiempo

- No es posible explicar el tema de análisis y diseño orientado a objetos en profundidad en 18 horas.
- Si es posible, darles los fundamentos que les permitan comenzar a practicar y a buscar información.
- "Practice makes perfect".

Metodología del curso

- Este es un curso tanto teórico como práctico.
- Se les explicará una parte de teoría sobre análisis y diseño.
- Después ustedes aplicarán esta parte a un ejemplo práctico de desarrollo.
- Esto se repetirá para cada uno de los pasos de los que consta el análisis y diseño.
- Al final del curso, ustedes habrán realizado el

El caso práctico al que ustedes aplicarán la teoría

- Es un ejercicio simplificado de facturación.
- Así podrán conocer simplificadamente un dominio que verán una y otra vez en su vida profesional.
- Los ejercicios se harán en grupos, pero cada uno debe aportar su propia versión.

Evaluación del curso

- 70% de la nota. Una media de algunas tareas realizadas en clase.
  - Se indicará qué tareas puntúan y cuáles no.
- 30% de la nota. Un examen tipo test al final del curso.

Mail: palasi@palasi.com Web: www.palasi.com

palasi.com
IT and management

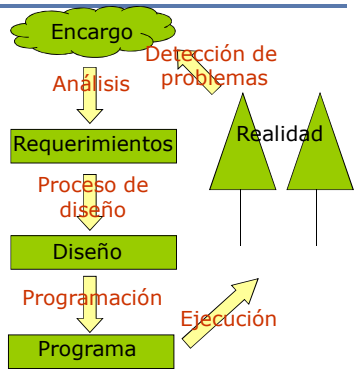
Parte 1: Introducción al análisis y diseño O-O y UML

- 1.1. Objetivos y metodología del curso.
- 1.2. Conceptos de análisis y diseño.
- 1.3. Modelos de desarrollo.
- 1.4. Utilidad del análisis y diseño.
- 1.5. Conceptos de programación O-O.

Objetivo del curso

- Enseñar los fundamentos del análisis y diseño orientado a objetos a los asistentes.
- Con el fin de que ustedes se vean capaces de desarrollar un sistema desde el principio al fin.
- Se debe comenzar con el encargo del cliente o del gerente y acabar en código.

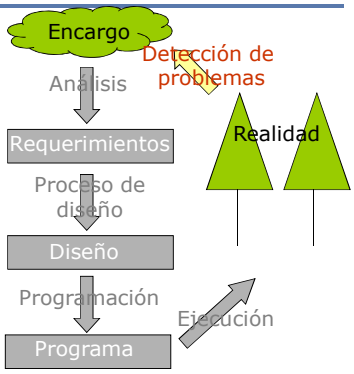
Un proceso de desarrollo simplificado



El encargo

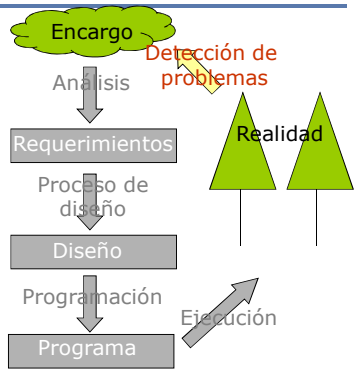
El cliente o jefe nos encarga la realización de un sistema.

Normalmente, el encargo es algo vago: "Queremos desarrollar un sistema de planilla".



El encargo

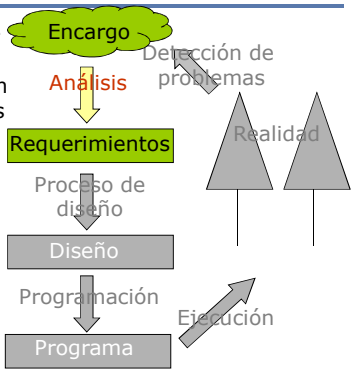
Normalmente, el encargo se basa en una detección de algún problema u oportunidad que se da a partir de la la realidad.



Análisis del sistema

Es el proceso por el cual se descubren cuáles son las características del sistema que se va a desarrollar.

Es decir, se define QUÉ va a hacer el programa.

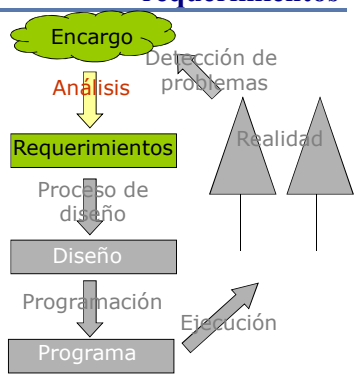


Mail: palasi@palasi.com Web: www.palasi.com

El análisis acaba con el documento de requerimientos

Es el documento que dice con detalle qué es lo que va a hacer el programa.

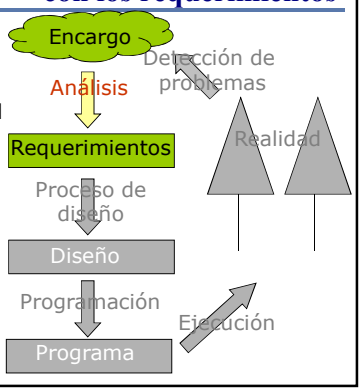
No entra en cuestiones de implementación: sólo define las funciones incluidas en el sistema.



No confundan el análisis con los requerimientos

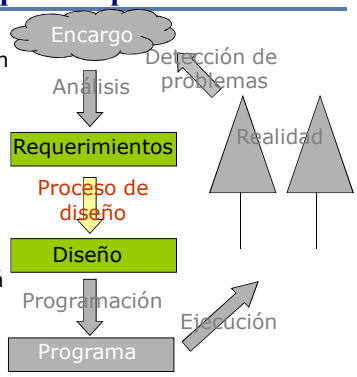
Los requerimientos es un documento que define QUÉ va a hacer el sistema.

El análisis es el proceso de descubrir cuáles son los requerimientos, a través de entrevistas con los usuarios, estudio, etc.



Con los requerimientos podemos empezar el proceso de diseño

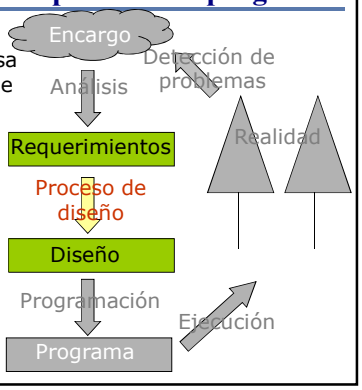
El proceso de diseño culmina en el diseño del sistema, que es una descripción de la estructura del programa: qué clases y métodos tendrá, qué tablas tendrá la BD, cómo se ejecutarán los procesos, etc.



El diseño son los "planos" del programa

Igual que los planos de una casa indican con detalle su estructura, el diseño muestra la estructura del programa.

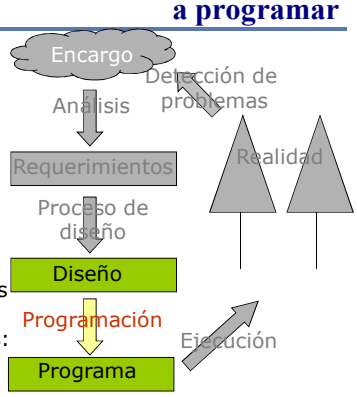
El proceso de diseño nos permite definir estos planos a partir de los requerimientos.



A partir del diseño, podemos comenzar a programar

El resultado del proceso de programación será un programa, que podremos ejecutar.

A partir de ahí, se repite el ciclo para solucionar errores o incorporar nuevas funciones: es el mantenimiento



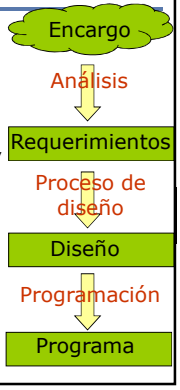
La analogía de la construcción de una casa

Encargo:
Necesitamos una casa grande

Requerimientos:
La casa tendrá capacidad para 6 personas. La casa permitirá dormir, comer, ir al baño, relajarse.

Diseño:
Los planos de la casa, que indican un comedor, una sala, varios dormitorios, un cuarto de baño.

Programa:
La casa física ya construida.



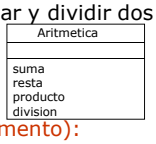
Mail: palasi@palasi.com Web: www.palasi.com

En un programa sencillo

Encargo:
Necesitamos un programa de aritmética.

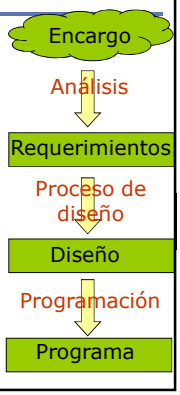
Requerimientos:
El programa permitirá sumar, restar, multiplicar y dividir dos números.

Diseño:



Programa (fragmento):

```
class Aritmetica {
   void suma (int a, int b) {
       return a+b;
   }
}
```



Parte 1: Introducción al análisis y diseño O-O y UML

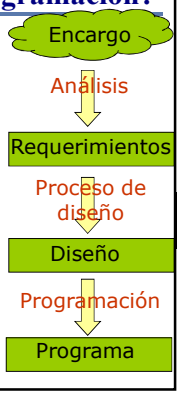
- 1.1. Objetivos y metodología del curso.
- 1.2. Conceptos de análisis y diseño.
- 1.3. Modelos de desarrollo.
- 1.4. Utilidad del análisis y diseño.
- 1.5. Conceptos de programación O-O.

¿En qué orden se produce el análisis, diseño y programación?

➢Si se observa el gráfico, parece que el orden de los procesos es el siguiente:

1. Se lleva a cabo el análisis hasta que se tienen los requerimientos completos.
2. Se ejecuta el proceso de diseño que lleva a cabo el diseño.
3. Se ejecuta la programación.

➢Esto es lo que se llama el "modelo en cascada" (the waterfall model).



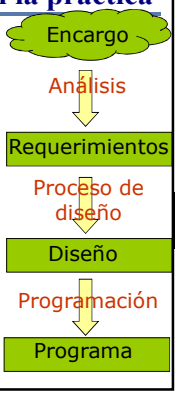
El modelo en cascada presenta muchos problemas en la práctica

➢Es muy difícil "congelar" los requerimientos y el diseño.

➢Conforme se desarrolla aparecen nuevas necesidades que el cliente no había previsto.

➢Mientras se desarrolla, nos damos cuenta de posibles problemas que no habíamos previsto.

➢Esto hace muy difícil de aplicar en la práctica el modelo en cascada: es demasiado rígido.

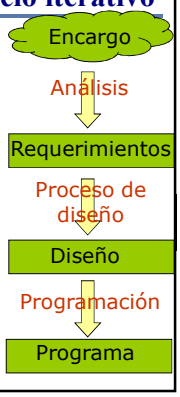


En la mayoría de los casos, es preferible un modelo iterativo

Los modelos iterativos dividen el desarrollo del sistema en periodos de tiempo llamados iteraciones (normalmente de unas semanas).

En cada iteración se intenta realizar una parte del sistema: se hace un poquito de análisis, un poquito de diseño y programación. Se acaba cada iteración con una parte del sistema.

Muchas veces, cuando realizamos parte del sistema debemos reformar partes ya construidas, esto se hace retocando el análisis, diseño y programación.

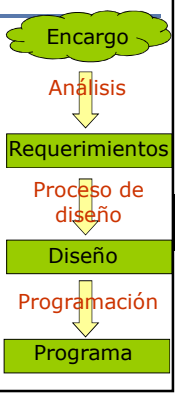


En los modelos iterativos

El desarrollo del sistema ocurre por aproximación.

Las iteraciones posteriores nos dan información para refinar lo que se ha hecho en las iteraciones anteriores, ya que descubrimos nuevos aspectos que anteriormente no se conocían.

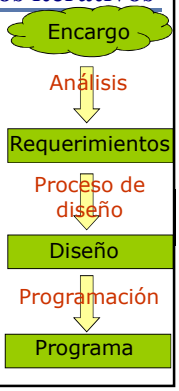
En la mayoría de proyectos, es una forma más realista de enfocar el desarrollo que el modelo en cascada, que peca de excesivamente rígido.



Hay varios tipos de modelos iterativos

Los más importantes son:

- Rational Unified Process (RUP). Es el estándar actual.
- eXtreme Programming (XP). De moda actualmente, con algunas ideas radicales y polémicas.
- Feature Driven Development (FDD). Menos popular, está a medio camino entre los dos anteriores.

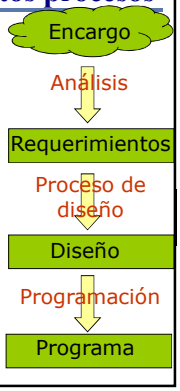


En este curso no podemos explicar estos procesos

Por falta de tiempo, seguiremos algo parecido a un modelo en cascada.

Como los ejercicios serán pequeños, será fácil hacer primero el análisis, después el diseño y después la programación.

Pero tengan en cuenta que esto no debería hacerse así en la realidad. Es más conveniente usar un modelo iterativo.



Parte 1: Introducción al análisis y diseño O-O y UML

- 1.1. Objetivos y metodología del curso.
- 1.2. Conceptos de análisis y diseño.
- 1.3. Modelos de desarrollo.
- 1.4. Utilidad del análisis y diseño.
- 1.5. Conceptos de programación O-O.

¿Para qué tanta complicación? ¡Comencemos a programar ya!

- Este es el estilo de programar de algunas empresas de bajo nivel y poco profesionales.
- Se resume en el chiste del jefe que le dice al programador "Tú empieza a escribir código. Mientras tanto, yo iré a ver qué es lo que el cliente quiere".
- ¿Se fiarían ustedes de un edificio construido sin planos? Tampoco se puede uno fiar de un programa construido sin análisis ni diseño.

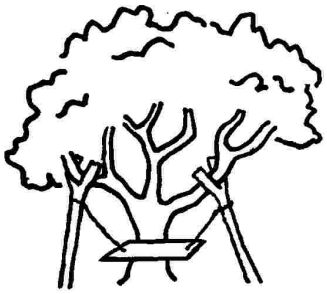
Un programa construido sin análisis y diseño

- Para grandes programas, esto es simplemente imposible. El programa nunca se terminará sin análisis y diseño.
- Para programas más pequeños:
  - Cuesta mucho más tiempo de programar, pues los cambios realizados se hacen con "parches", al no existir una visión clara de la estructura del programa.
  - Tiene una calidad mucho más baja, ya que está lleno de "parches" añadidos con posterioridad a la programación.
  - Contiene muchos más errores.
  - Es mucho más difícil de mantener sin una documentación de la programación ni una programación limpia.

Un programa desarrollado con análisis y diseño

Mail: palasi@palasi.com Web: www.palasi.com

Un programa desarrollado sin análisis y diseño



Estructura inestable y barroca, llena de errores, difícil de mantener.

Utilidad de este curso

- Este curso les enseñará los conceptos básicos del análisis y diseño orientado a objetos.
- Con este curso y un poco de práctica, ustedes serán capaces de desarrollar el análisis y diseño de programas reales.
- Este curso tiene una gran utilidad práctica por dar técnicas que son **necesarias** en todos los desarrollos de todos los

Parte 1: Introducción al análisis y diseño O-O y UML

- 1.1. Objetivos y metodología del curso.
- 1.2. Conceptos de análisis y diseño.
- 1.3. Modelos de desarrollo.
- 1.4. Utilidad del análisis y diseño.
- 1.5. Conceptos de programación O-O.

Punto 1.5: Conceptos de programación O-O

- 1.5.1. Orígenes de la programación orientada a objetos.
- 1.5.2. Conceptos básicos de orientación a objetos.
- 1.5.3. Concepto de clase.
- 1.5.4. Otros Conceptos de programación O-O.
- 1.5.5. Conclusión.

Punto 1.5: Conceptos de programación O-O

- 1.5.1. Orígenes de la programación orientada a objetos.
- 1.5.2. Conceptos básicos de orientación a objetos.
- 1.5.3. Concepto de clase.
- 1.5.4. Otros Conceptos de programación O-O.
- 1.5.5. Conclusión.

Recordemos el objetivo del curso

- Enseñar los fundamentos del análisis y diseño orientado a objetos a los asistentes.
- Hemos visto la utilidad del análisis y diseño, pero ¿por qué orientado a objetos?
- ¿Por qué no otros tipos de análisis y diseño?

Mail: palasi@palasi.com Web: www.palasi.com

¿Por qué análisis y diseño orientado a objetos?

- La orientación a objetos se ha convertido en el estándar actual de programación para el desarrollo.
- Todos los lenguajes de programación modernos son orientados a objetos:
  - C++
  - Java
  - Python
  - C#
  - Visual Basic .NET

Aún quedan algunas empresas que siguen programando sin O-O

- Sin embargo, todas las empresas de calidad a nivel mundial utilizan la programación orientada a objetos desde hace tiempo (excepto con las aplicaciones de legado).
- Sin embargo, ustedes están a punto de egresar y les quedan unos cuarenta años de vida profesional, así que se les va a enseñar el análisis y diseño orientado a objetos que es el actual y el que tiene futuro.
- Las futuras metodologías de programación (por ejemplo, "aspect-oriented programming") ya

Desarrollo orientado a objetos

- Es una metodología para el desarrollo de software que modela el software como objetos que interactúan entre sí.
- Comprende todas las fases del desarrollo: análisis, diseño, programación.
- Se ha convertido en el estándar a nivel mundial.

Este no es un curso para enseñar el paradigma orientado a objetos

- Ustedes ya deberían conocer el desarrollo orientado a objetos
- Sin embargo, aquí haremos un repaso de los conceptos más importantes para refrescarles la memoria.
- Será un repaso rápido pero completo

La crisis del software

- Es el nombre que damos al estado insatisfactorio en que se encuentra el desarrollo de software.
- Las tareas que nos gustaría resolver mediante las computadoras son en la práctica:
  - demasiado difíciles de resolver.
  - suelen extenderse más allá del costo y el tiempo previsto.
  - es muy probable que contengan errores.

La crisis del software

- Los proyectos de programación de gran tamaño consumen 150% del tiempo previsto.
- El 25% de estos proyectos son cancelados.
- El 75% de los proyectos no cancelados:
  - O bien no funcionan como se quería.
  - O bien no se utilizan para nada.

Estadísticas sobre proyectos de software

Tamaño	Temprano	A tiempo	Retraso	Cancelados
1 PF	14.68%	83.16%	1.92%	0.25%
10 PF	11.08%	81.25%	5.67%	2.00%
100 PF	6.06%	74.77%	11.83%	7.33%
1000 PF	1.24%	60.76%	17.67%	20.33%
10000 PF	0.14%	28.03%	23.83%	48.00%
100,000PF	0.00 %	13.67%	21.33%	65.00%

- Fuente: Patterns of Software Systems Failure And Success, Capers Jones.

Soluciones a la crisis del software

- La crisis se detectó en la conferencia de la OTAN de 1968 sobre Ingeniería del Software y sigue vigente.
- Se han investigado varias soluciones:
  - Derivación y verificación automática de software (de interés más teórico).
  - Programación con componentes reusables de software.

Programación con componentes reusables de software

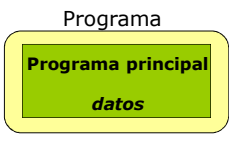
- Una de las técnicas para mitigar los problemas del desarrollo de software.
- Se basa en la idea extendida en otras ingenierías que, para construir un producto, sólo deben ensamblarse piezas preconstruidas.
- La idea básica es:
  - existen componentes reusables
  - programar una aplicación consiste en ensamblarlos.

Programación con componentes reusables de software

- Objetivo perseguido durante muchos años.
- Para conseguirlo se han desarrollado varias técnicas de programación:
  - programación no estructurada.
  - programación procedimental.
  - programación modular.
  - programación orientada a objetos.

Programación no estructurada

- Es el primer tipo de programación que apareció.
- Hay un único programa principal que trabaja con unos datos globales.

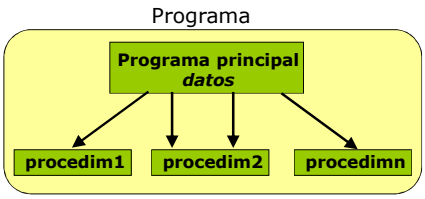


Características de la programación no estructurada

- Hace muy difícil la programación a gran escala:
  - no hay ninguna reusabilidad del código.
  - si hay que realizar una tarea dos veces, se debe volver a escribir el código.
- Prácticamente no se utiliza en el mundo real.

Programación procedimental

- El código de tareas que se repiten se escribe en procedimientos ("subprogramas"). El programa principal los llama cada vez que debe hacer dichas tareas.



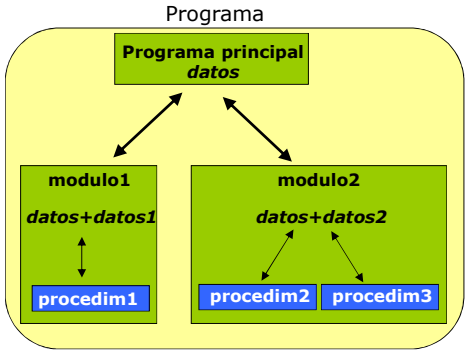
Características de la programación procedimental

- Reusabilidad a nivel elemental:
  - se reusa el código de los procedimientos
  - se evita la tarea de repetir código dentro de un programa
- El problema es que los procedimientos no pueden ser reusados por otros programas.

Programación modular

- Los procedimientos referentes a un mismo conjunto de datos se agrupan en un módulo (compilado separadamente).
- Cada programa consta de varios módulos. El programa principal coordina las llamadas a los procedimientos de los otros módulos.

Programación modular



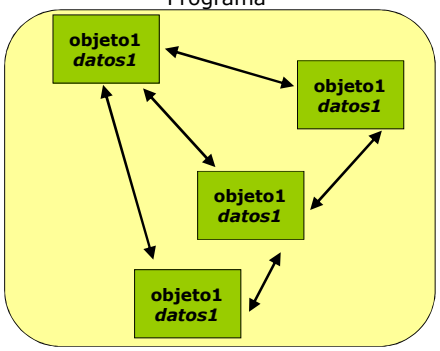
Características de la programación modular

- Hay una reusabilidad de los módulos.
- De hecho, los módulos tienen sus propios datos y se convierten en parecidos a un tipo de datos con sus operaciones (ejemplo: lista).
- Pero hay inconvenientes:
  - los módulos no se acaban de comportar como un tipo de datos.
  - se vuelve complicado tener y gestionar varias instancias del módulo (varias listas).

Programación orientada a objetos

- Resuelve los problemas mencionados.
- Desaparece el concepto de programa principal.
- Lo que hay es una red de *objetos* interactuantes.
- Los objetos
  - contienen datos y operaciones sobre estos datos.
  - interactuan intercambiando mensajes (análogo a llamadas a procedimiento).

Programación orientada a objetos



La programación O-O no es algo aislado

- Como veremos ahora, introduce una nueva forma de pensar.
- Esto implica que el análisis y diseño también deben cambiar para adecuarse a esta nueva forma de pensar.
- Así, se habla del análisis y diseño orientado a objetos, que es el objetivo de este curso.
- Por ahora, nos centraremos en la programación orientada a objetos y después hablaremos del análisis y diseño.

Punto 1.5: Conceptos de programación O-O

- 1.5.1. Orígenes de la programación orientada a objetos.
- 1.5.2. Conceptos básicos de orientación a objetos.
- 1.5.3. Concepto de clase.
- 1.5.4. Otros Conceptos de programación O-O.
- 1.5.5. Conclusión.

Programación orientada a objetos

- Es una de las técnicas para implementar programación con componentes reusables de software.
- Estándar dominante actualmente de programación.
- Evoluciona de otras formas anteriores de programación, pero introduce una manera diferente de pensar (más parecida a la vida real).
- Se basa en el concepto de objeto.

Un objeto es análogo a un objeto de la vida real

- Por ejemplo, un objeto real es el Toyota Corola que es propiedad del licenciado.
- De un objeto, tanto de software, como real, nos interesan dos cosas.

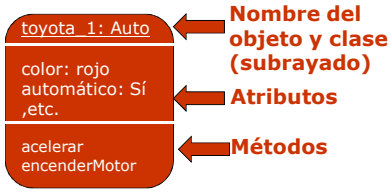
Las dos cosas que nos interesan de un objeto

- Sus características (su color, tamaño, su potencia, si tiene caja de cambios automática, etc.). A esto se le llama **atributos o propiedades.**
- El tipo de acciones que pueden realizarse con él (acelerar, encender el motor, apagarlo, etc). A esto se le llama **métodos**.

Mail: palasi@palasi.com Web: www.palasi.com

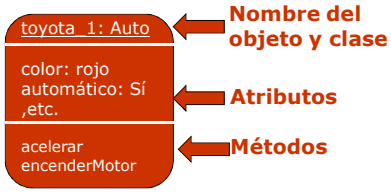
Un objeto de software es parecido

- De hecho suele modelar un objeto de la vida real: por eso se le parece.
- Nos interesan sus atributos (características) y las cosas que pueden hacerse con él: sus métodos.



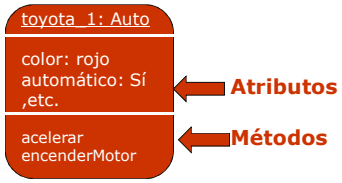
Un objeto de software es parecido

- Los atributos son variables (que tienen un valor) y los métodos son parecidos a procedimientos.
- Pero todo esto está unido en una sola entidad: a ello se le llama encapsulación.



Objeto

- Concepto principal de la programación O-O.
- Encapsulación. Contiene:
  - datos (atributos).
  - métodos para efectuar tareas sobre esos datos.
- Intenta modelar una entidad de la vida real.



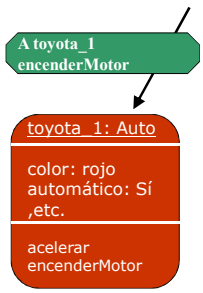
El concepto de objetos es algo natural

- Un objeto de la vida real es una asociación de características y funciones que.
- Esto se modela naturalmente como un objeto de software, con atributos y métodos.
- En anteriores tipos de programación, los datos y las funciones se programaban por separando agrandando la brecha entre la representación y la realidad.

Operaciones con objetos

- Si queremos que un objeto realice una operación de las que sabe hacer, debemos "pedir" esta información de alguna manera.
- Esta "petición" se le llama "mensaje" en la programación O-O.
  - En el caso del auto, la petición de que encender el motor se produce girando la llave.
  - En un objeto de software, se haría con un mensaje al objeto de que se quiere encender el motor.

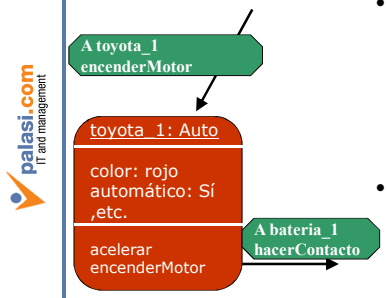
Mensaje o llamada a método



- Es una petición a un objeto para que realice una tarea de las que sabe hacer este objeto.
- El objeto realiza la tarea siguiendo el método que tiene definido.

Mail: palasi@palasi.com Web: www.palasi.com

Método

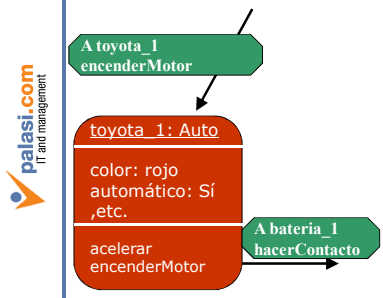


- El objeto para realizar la tarea que ha pedido el mensaje tiene un método definido (a veces, enviar otro mensaje a otro objeto)
- Este método está oculto:
  - sólo lo conoce el objeto
  - no se conoce desde el exterior (excepto su nombre).

Esto es análogo a la vida real

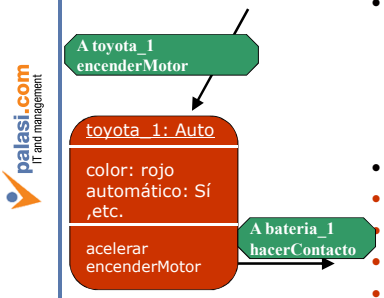
- Cuando cambio de cadena en un televisor, le hago una petición mediante un botón al televisor.
- Él conoce un método para cambiar de canal, pero yo no lo conozco ni me interesa.
- Cuando enciendo el motor de un auto, no conozco el método con el que se realiza esto, pero el auto sí.
- A esto se le llama "ocultación de la implementación" y es una de las características de la programación orientada a objetos

Desde el punto de vista de la programación



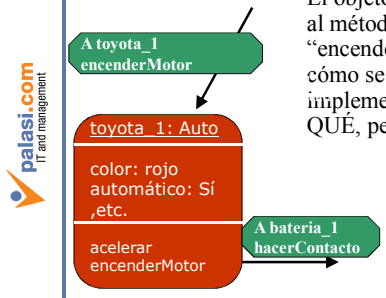
- Los atributos no son más que variables asociadas al objeto.
- Como toda variable, tienen
- Nombre.
- Tipo.
- Valor.

Desde el punto de vista de la programación



- Los métodos no son más que procedimientos, funciones o subrutinas.
- Por ello tienen
- Nombre.
- Parámetros.
- A veces, resultado.
- Una implementación o cuerpo.

Ocultación de la implementación



- El objeto que ha llamado al método "encenderMotor", no sabe cómo se realiza, cómo está implementado (sabe el QUÉ, pero no el CÓMO)
  - Esta ocultación de la implementación es de lo más importante en O-O. Facilita la programación. Modelo de delegación.

Viéndolo desde el punto de vista de la programación convencional

- Un método es análogo a un procedimiento o subrutina
- Un fragmento de código que hace algo y que (a veces) retorna un resultado.
- La implementación del método (cómo está construido, su cuerpo) no se puede ver desde fuera del objeto. Esto hace más simple la programación: ocultamiento de la implementación.
- El método se puede llamar desde fuera del objeto, a esto se le llama "mensaje"

Mail: palasi@palasi.com Web: www.palasi.com

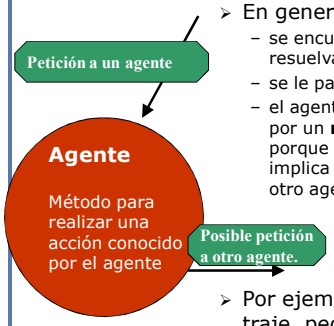
La programación orientada a objetos

- Soluciona los problemas de la misma manera que se solucionan los problemas de la vida real.
- ¿Cómo resolvemos un problema en la vida real?
- Veamos un ejemplo.

Problema real: cómo enviar una remesa a El Salvador

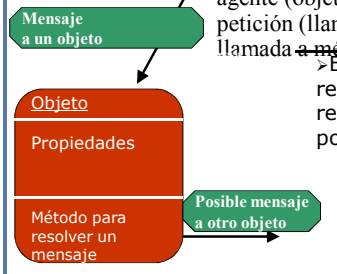
- La solución es:
  - llegar a una empresa de mensajería privada, Remesa Express.
  - pedirles que quiero enviar el dinero a El Salvador.
    - Remesa Express resolverá el problema con un método que no me interesa saber.
      - –normalmente llamará la sucursal a El Salvador y les pedirá que entreguen el dinero.

Solución de problemas en la vida real



- En general:
  - se encuentra un **agente** que resuelva el problema.
  - se le pasa una **petición**.
  - el agente resuelve mi petición por un **método** que no tengo porque conocer (a menudo implica enviar una petición a otro agente)
- Por ejemplo, comprar un traje, pedir una pizza por teléfono...

Solución de problemas en un entorno orientado a objetos

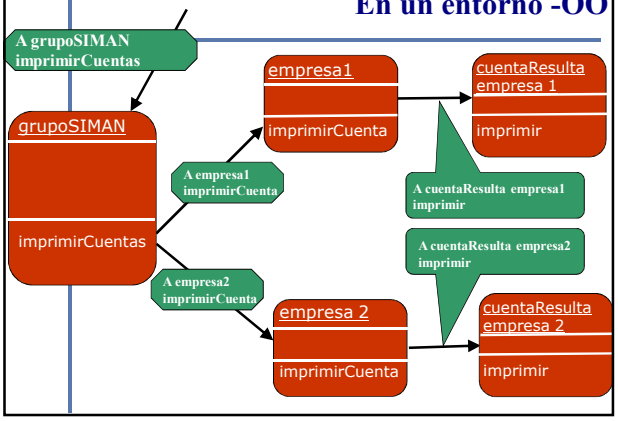


- La acción se inicia cuando a un agente (objeto) se le pasa una petición (llamada mensaje o llamada a método).
  - El objeto que recibe el mensaje resuelve la petición por un método
    - –que no tenemos por que saber
    - –que a menudo implica enviar otro mensaje a otro objeto.

Un ejemplo de programación 0-0

- Supongamos que tenemos un sistema informático para el grupo de empresas SIMAN y queremos imprimir por pantalla la cuenta de resultado de cada empresa.

En un entorno -OO



Mail: palasi@palasi.com Web: www.palasi.com

Modelo de delegación

- El objeto del grupo SIMAN delega (parte del) trabajo a los objetos empresa.
- Los objetos empresa delegan (parte del) trabajo a los objetos cuenta de resultado.
- En general, normalmente los objetos delegan (parte del) trabajo a otros objetos.
- Esto también es como la vida real.

Resumen de los conceptos de O-O hasta ahora

- Un programa O-O está formado por una serie de objetos que interactúan intercambiando mensajes (es decir, llamando métodos de otros objetos).
- Un objeto es un conjunto de datos y de métodos para realizar tareas con esos datos

Punto 1.5: Conceptos de programación O-O

- 1.5.1. Orígenes de la programación orientada a objetos.
- 1.5.2. Conceptos básicos de orientación a objetos.
- 1.5.3. Concepto de clase.
- 1.5.4. Otros Conceptos de programación O-O.
- 1.5.5. Conclusión.

Clases y objetos

- Un programa utiliza una serie de objetos y normalmente muchos de ellos son similares.
- Al conjunto de estos objetos similares se le llama **clase**

Clases y objetos

- En la vida real, los objetos que son similares forman una clase de objetos. Es decir, llamamos clase al **conjunto** de objetos similares.
- Los Toyota Corola forman una clase ("¿De qué clase es tu auto?").
- Es propio de la mente humana ordenar el Universo en clases para un mejor manejo y comprensión.

Otra definición de clase

- La descripción o modelo (o molde) de estos objetos similares se le llama **clase**
- Esta definición es equivalente a la anterior.

Mail: palasi@palasi.com Web: www.palasi.com

Una clase es un documento de instrucciones de fabricación

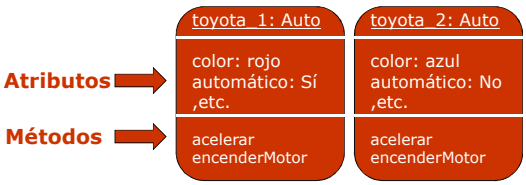
- Una clase es una descripción o modelo de diseño de todos los objetos que forman parte de ella. Así, "Toyota Corola" es una especificación para crear autos de una cierta marca o modelo.

Clases y objetos

- La clase indica cómo serán los objetos que se crearán con ese modelo
- Una clase sería un documento que nos describiera cómo es un "Toyota Corola"
- Un objeto sería el auto "Toy.Corola" de Juan

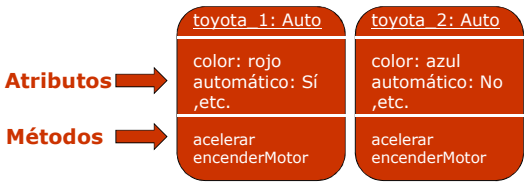
¿Cómo sabemos si dos objetos son de la misma clase?

- De un objeto sólo me interesan:
  - Sus propiedades o **atributos.**
  - El tipo de acciones que pueden realizarse con él o **métodos**

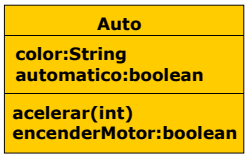


Dos objetos son de la misma clase si

- Tienen los métodos idénticos.
- Sus atributos tienen el mismo nombre y tipo (pero pueden variar en valor).

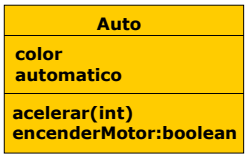


Las clases las representaremos de esta manera



- Especificamos:
  - Nombre de la clase.
  - Nombres y tipos de los atributos. Valores no, pues pueden variar entre objetos.
  - Nombres y parámetros de los métodos. Su cuerpo o implementación no, porque no cabría.

A veces se puede simplificar la notación de las clases



- Se puede dejar de identificar el tipo de los atributos, siempre que no se considere conveniente entrar en detalle sobre los mismos.

Mail: palasi@palasi.com Web: www.palasi.com

A veces se puede simplificar la notación de las clases

- Si sólo nos interesa la existencia de la clase y no queremos entrar en detalle sobre sus atributos y métodos podemos usar la siguiente notación simplificada.

Auto

Importante: el nombre de la clase siempre va en singular

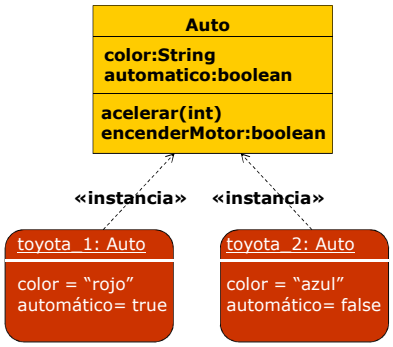
- Aunque realmente representa un conjunto plural de cosas.
- Así, tenemos la clase "Auto" que representa un conjunto de autos.

Auto

Representa a varios **autos**

La relación entre clases y sus instancias la representamos así

Fíjense que los objetos instancia contienen el valor de los atributos



La notación de objetos también se puede simplificar

toyota_1: Auto

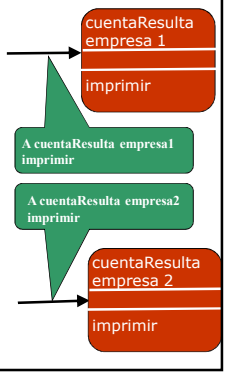
- Sólo debemos incluir el nombre del objeto y la clase a la que pertenece, si no queremos entrar en detalles.

Resumen: Cuatro formas de ver lo mismo

- Una clase es el conjunto de todos los objetos del mismo tipo.
- Una clase es un "molde" para hacer objetos del mismo tipo.
- Una clase son las instrucciones para crear objetos de este tipo.
- Una clase son los objetos con los mismos métodos y atributos con el mismo nombre y tipo.

En nuestro ejemplo del grupo Simán

- Los objetos "cuentaResulta empresa1" y "cuentaResulta empresa2" tienen los mismos métodos y atributos con el mismo nombre y tipo.
- Forman parte de la clase "CuentaResultados".



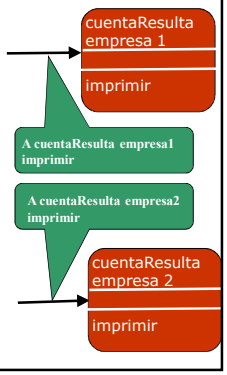
Mail: palasi@palasi.com Web: www.palasi.com

Un poco de terminología

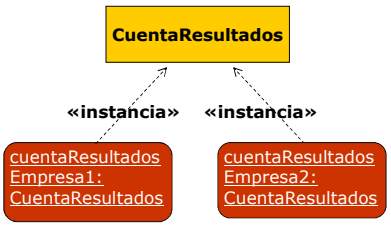
- Cuando un objeto pertenece a una clase, se dice que es una **instancia** de esta clase.
- Así, los diferentes autos del mundo son instancias de la clase Auto.

En nuestro ejemplo del grupo Simán

- Los objetos "cuentaResulta empresa1" y "cuentaResulta empresa2" serían instancias de la clase "CuentaResultados".

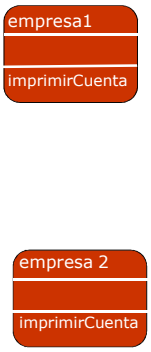


Los dos objetos son instancias de la clase "CuentaResultados"

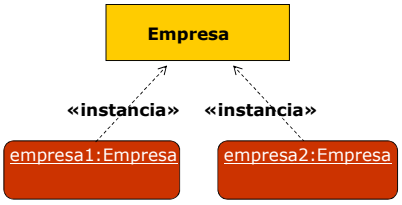


En nuestro ejemplo del grupo Simán

- Los objetos "empresa1" y "empresa2" tienen los mismos métodos y atributos con el mismo nombre y tipo.
- Forman parte de la clase "Empresa".
- Son instancias de "Empresa".



Los dos objetos son instancias de la clase "Empresa"



Un hecho incuestionable

- Todos los objetos pertenecen a una clase.
- No hay objetos sin clase.
- Por eso, cada vez que creamos un objeto debemos especificar a qué clase pertenece.
- Es la plantilla ("o molde") a partir de la cual se crearán los objetos.

Mail: palasi@palasi.com Web: www.palasi.com

Definición y creación de un objeto

- Para crear un objeto, debe indicarse a partir de qué clase se creará el objeto.
- Se crea el objeto siguiendo ese "molde" (esa especificación) que es la clase.

Un hecho importante

- **Las clases son tipos de datos.**
- Pueden manipularse como cualquier otro tipo de datos.
- Se pueden declarar variables, atributos, valores, parámetros, etc. cuyo tipo sea una clase.

Las clases son tipos de datos

- Los atributos de la clase **`Persona`** son de tipos de datos que son clases (**`Persona`**, **`Empresa`**).
- El parámetro del método **`casarseCon`** es de tipo **`Persona`**, que es una clase.

Persona
padre:Persona madre:Persona empleador: Empresa
casarseCon(Persona)

Resumen de los conceptos de O-O hasta ahora

- Un programa O-O está formado por una serie de objetos que interactúan intercambiando mensajes.
- Para realizar una tarea, se envía un mensaje a un objeto que es el responsable de ejecutar esa tarea mediante un método que tiene definido para hacerlo.
- El método no es conocido por los otros objetos.

Resumen de los conceptos de O-O hasta ahora

- Un objeto es un conjunto de datos (atributos) y de métodos para realizar tareas con esos datos.
- Los objetos con el mismo nombre y tipo de datos y los mismos métodos se agrupan en clases.

Punto 1.5: Conceptos de programación O-O

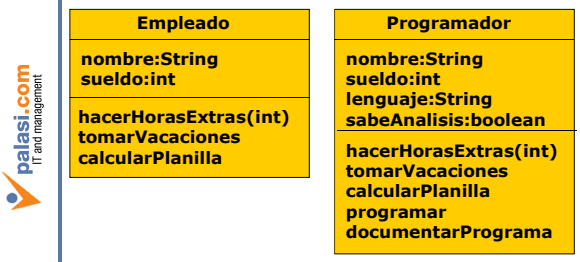
- 1.5.1. Orígenes de la programación orientada a objetos.
- 1.5.2. Conceptos básicos de orientación a objetos.
- 1.5.3. Concepto de clase.
- 1.5.4. Otros Conceptos de programación O-O.
- 1.5.5. Conclusión.

Mail: palasi@palasi.com Web: www.palasi.com

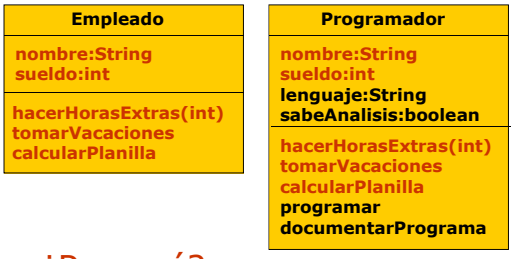
Herencia

- Supongamos que existe una clase "empleado" que modela los empleados de una empresa.
- Queremos definir una clase "programador", que modela nuestros programadores.

Herencia

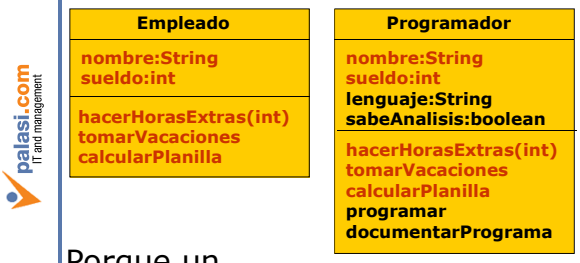


Programador tiene todos los atributos y métodos de Empleado



¿Por qué?

Programador tiene todos los atributos y métodos de Empleado

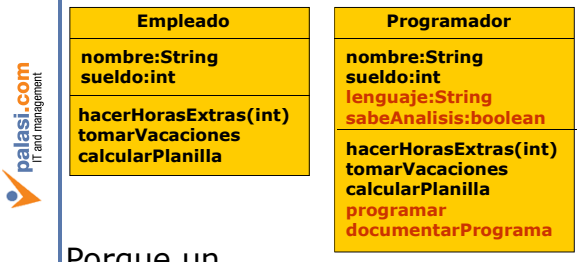


Porque un programador ES UN empleado.

Un programador es un empleado

- Claramente la clase "programador"
  - tendrá todos los atributos y métodos de la clase empleado, ya que un programador ES UN empleado.
  - tendrá algunas características peculiares de los programadores que no tienen los otros empleados.

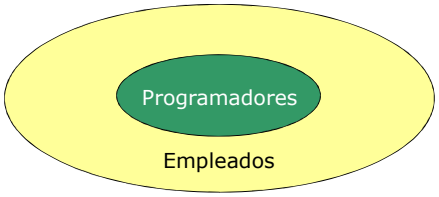
Atributos y métodos de programador que carece empleado



Porque un programador ES UN empleado.

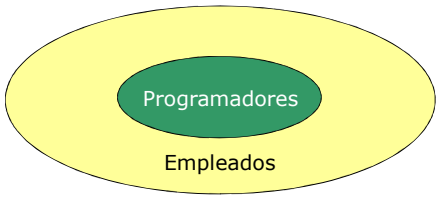
Mail: palasi@palasi.com Web: www.palasi.com

El conjunto de los programadores es un subconjunto del de los empleados



- Todos los programadores son empleados.
- NO TODOS los empleados son programadores.

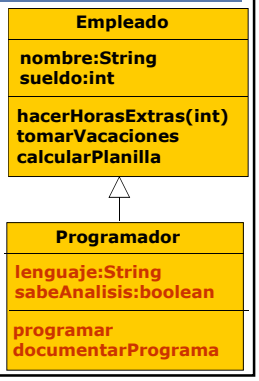
El conjunto de los programadores es un subconjunto del de los empleados



- Se dice que el conjunto de los programadores es un subconjunto del conjunto de empleados.
- **Se dice que la clase programadores es una subclase de empleados.**
- **Se dice que la clase empleados es una superclase de programadores.**

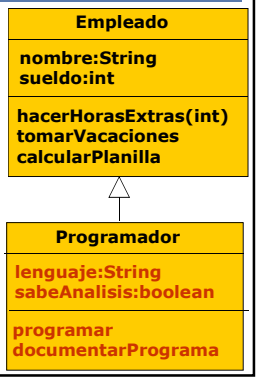
Esto se representa así

- Una flecha triangular apunta de la subclase a la superclase.
- Como ven, en la clase Programador ya no especificamos los atributos y métodos que comparte con empleado.
- La flecha nos indica que programador es subclase de empleado y, por lo tanto, tendrá todos los atributos y métodos de empleado, por lo cual no hace falta repetirlos.



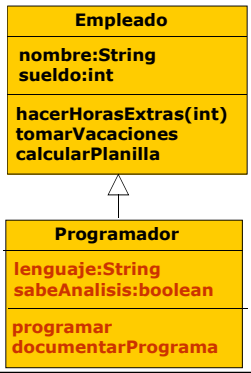
Se dice que la clase Programador

- Hereda los atributos y métodos de Empleado.
- Para abreviar, se dice que Programador hereda de Empleado.
- Una clase que hereda de otra, tiene todos los atributos y métodos de la otra.
- Esto se representa con la



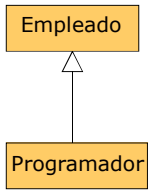
En los lenguajes de programación orientada a objetos

- Se indica que una clase hereda (es subclase de) otra con una palabra clave.
- En la subclase no se repiten los atributos y métodos heredados (los que estaban en la superclase) pues ya se dan por supuesto.
- Esto facilita la programación y el mantenimiento (como

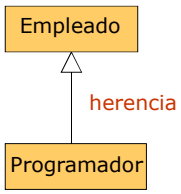


Si no queremos entrar en detalles

- Podemos prescindir de atributos y métodos.

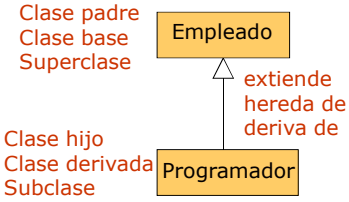


A la relación entre estas dos clases se le llama herencia



- También puede llamarse relación ES-UN (is a), porque cada empleado "es un" programador.

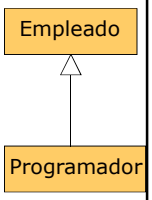
Más terminología



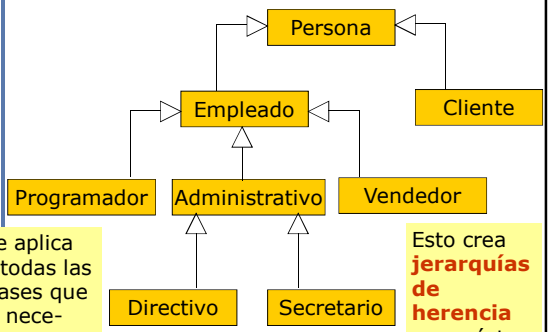
- Se dice que Programador hereda de Empleado o bien que Programador extiende Empleado o bien que Programador deriva de Empleado.

Recapitulando: herencia

- La clase "programador" tiene
  - todos los atributos y métodos de "empleado".
  - además algunos específicos de ella.
- Se dice
  - "programador" hereda de "empleado".
  - "programador" es subclase de "empleado".
- La herencia nos permite construir clases
  - sin tener que escribir todos los datos y métodos.
  - de modo más ordenado.



La herencia no se da entre dos únicas clases

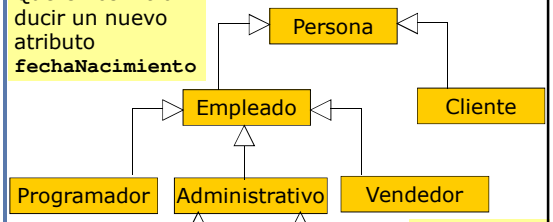


Se aplica a todas las clases que la nece-siten

Esto crea **jerarquías de herencia** como ésta

La her. mejora la programación y el mantenimiento

Queremos intro-ducir un nuevo atributo `fechaNacimiento`

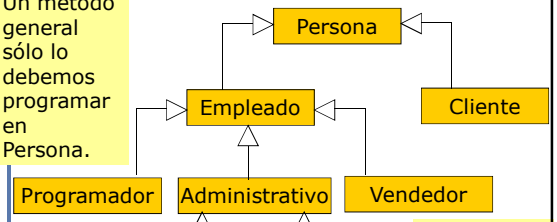


Lo añadiremos a la clase Persona.

Sin heren-cia, debería-mos progra-marlo en todas las clases.

Con los métodos es lo mismo

Un método general sólo lo debemos programar en Persona.

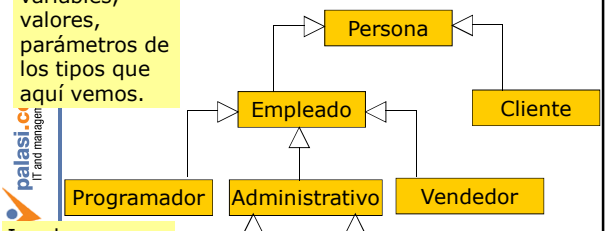


Nos ahorramos mucho trabajo.

Evitamos inconsisten-cias que tendríamos sin herencia

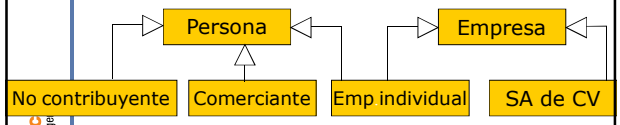
Nota: No olvidemos que las clases son tipos de datos

Así, podemos definir variables, valores, parámetros de los tipos que aquí vemos.



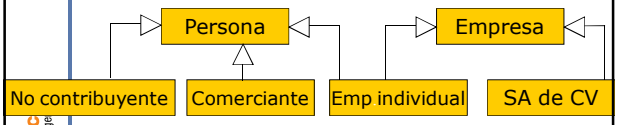
Igual que definimos variables, valores y parámetros de tipo entero.

Herencia múltiple



- "Emp. Individual" tiene herencia múltiple:
  - hereda todos los datos y métodos de empresa.
  - hereda todos los datos y métodos de persona física.
- Es una persona y es una empresa
- A veces, esto puede producir conflictos.
- La mayoría de lenguajes orientados a objetos no lo permiten. Nosotros tampoco.

Conflictos en herencia múltiple



- Supongamos que:
  - "Persona" define un método llamado "imprimir estado financiero".
  - "Empresa" define un método con el mismo nombre pero que es diferente (adecuado a una empresa)
- ¿Cuál es el método que hereda "Emp. Individual"?

Otros conceptos de la orientación a objetos

- **Polimorfismo:** Habilidad de los métodos de aceptar parámetros de tipos diferentes ligados por la herencia. Así, un método que se ha definido para aceptar parámetros de tipo empleado, aceptará parámetros de tipo administrativo, vendedor, programador, secretario y directivo.
- **Genericidad:** Facultad de algunos lenguajes orientados a objetos de definir estructuras de datos genéricas que permiten contener diferentes tipos de datos. Así se puede definir un tipo lista y usarlo para crear listas de enteros, listas de cadenas, etc.

Punto 1.5: Conceptos de programación O-O

- 1.5.1. Orígenes de la programación orientada a objetos.
- 1.5.2. Conceptos básicos de orientación a objetos.
- 1.5.3. Concepto de clase.
- 1.5.4. Otros Conceptos de programación O-O.
- 1.5.5. Conclusión.

La programación O-O es una herramienta muy útil

- La reutilización de clases:
  - reduce en mucho el esfuerzo de análisis y programación.
  - permite construir las aplicaciones de forma modular lo que es más fácil
  - facilita el mantenimiento de las aplicaciones
- Los conceptos de objeto y mensaje permiten razonar de un modo más fácil y natural sobre la programación.
- Pero las mayores ventajas de la programación orientada a objetos es la flexibilidad y escalabilidad.

Mail: palasi@palasi.com Web: www.palasi.com

Pregunta

- ¿Cuál es el problema más grande con el desarrollo de software?
- Pista (elijan entre estos):
  - El análisis.
  - El diseño.
  - La programación.
  - La instalación.
  - Las pruebas.
  - El mantenimiento.

El problema más grande de desarrollo

- **Es el mantenimiento.**
- Un programa puede ser mantenido por años y años.
  - Para eliminar defectos.
  - Para añadir nuevas funciones.
  - Para adecuarlo a una realidad cambiante.
- Mantener es difícil, porque muchas veces la estructura inicial del programa no acepta fácilmente los cambios.
- No ha sido concebido para ello.

La programación orientada a objetos

- Es mucho más flexible y fácil de modificar que la programación convencional.
- La estructura acepta mejor los cambios.
- Es por eso que facilita muchísimo el mantenimiento.
- También facilita la programación de los proyectos, sobre todo, si son grandes.
- Cuando más grande es un programa, más provecho obtiene de la orientación a objetos.
- La programación O-O es mucho más escalable que la

La programación O-O no es una solución mágica

- Es extremadamente útil. Por su flexibilidad, facilita:
  - la tarea de programar proyectos de gran escala
  - reaprovechar el trabajo hecho por programadores
- Pero desarrollar un sistema grande sigue siendo una tarea extremadamente difícil.
- Sigue requiriendo:
  - creatividad, talento, capacidad de abstracción, lógica y experiencia.
  - Mucho orden.
  - Y, por supuesto, análisis y diseño.

Parte 1: Introducción al análisis y diseño O-O y UML

- 1.1. Objetivos y metodología del curso.
- 1.2. Conceptos de análisis y diseño.
- 1.3. Modelos de desarrollo.
- 1.4. Utilidad del análisis y diseño.
- 1.5. Conceptos de programación O-O.

Conceptos de UML

- Después de este breve repaso sobre los conceptos básicos de programación orientada a objetos, volvemos al tema básico de este curso: el análisis y diseño orientado a objetos.
- Nos vamos a dedicar a explicar algunos conceptos de UML.
- Lo estudiaremos con más detalle en el resto del curso.
- Pero, ¿qué es UML?

Los seres humanos transmitimos información

- Los seres humanos intercambiamos información entre nosotros, con los animales y con nuestras máquinas.
- Se llama emisor al que emite la información y receptor al que la recibe.
- La información se transmite del emisor al receptor en un lenguaje que los dos entienden.
  - Entre humanos, es un lenguaje natural.
  - Entre humano y máquina, es un lenguaje de programación.

Una misma información puede expresarse en lenguajes diferentes

- Así, la información que indica la salida de un edificio, puede expresarse en los siguientes lenguajes
- Español: Salida.
- Inglés: Exit.
- Francés: Sortie.
- Catalán: Eixida.
- Lenguaje icónico: 
- Lenguaje de signos, etc.

De la misma forma, la información de un programa puede ser expresada

- Con muchos lenguajes, llamados lenguajes de programación.
- Así, un programa que sume dos números, será expresado de forma muy diferente.
  - En código máquina. Con ceros y unos.
  - En ensamblador. Con mnemónicos.
  - En C, Pascal.
  - En C++, Java.

Con la programación estamos claros

- Ahora bien, ¿en qué lenguaje vamos a expresar la información sobre análisis y diseño de un programa?
- Hay varias alternativas:
  - En lenguaje natural.
  - Con lenguajes de A & D tradicionales: DFDs, Entidad-Relación, etc.
  - Con lenguajes de A & D orientado a objetos: Jacobson, Booch, UML.

¿En qué lenguaje expresamos el análisis y diseño orientado a objetos?

- En lenguaje natural al 100% es contraproducente, pues la complejidad de la estructura del programa se aprende mejor de manera visual.
- Con lenguajes de análisis y diseño tradicionales es difícil, pues no podemos modelar los conceptos básicos de la orientación a objetos.
- Debemos encontrar un lenguaje de análisis y diseño que esté especialmente diseñado para la orientación a objetos.

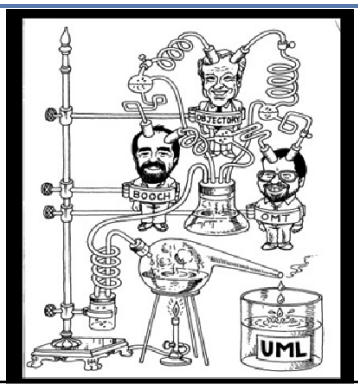
Lenguajes para expresar el análisis y diseño orientado a objetos

- Antes de 1994, habían varios lenguajes. Los más importantes:
  - Objectory method, de Ivar Jacobson.
  - OMT (Object Modeling Technique) de Jim Rumbaugh.
  - El método Booch, de Grady Booch.
- Esto era muy inconveniente, pues los métodos eran incompatibles y competían entre sí.
- Cada uno tenía sus fuerzas y debilidades, sus seguidores y detractores.
- Malo para los programadores, para los empresas y

A partir de 1994, estos métodos comenzaron a unirse

- Sus creadores se asociaron ("The three amigos") y los combinaron para hacer un lenguaje **UNICO** para el análisis y diseño.
- Lo llamaron UML (Unified Modeling Language).
- Con el tiempo, el OMG (Object Management Group) perfeccionó el UML y éste se convirtió en el estándar.

"The three amigos"



Hoy sólo hay un lenguaje estándar para el análisis y diseño

- Y este es el lenguaje UML, que es el que aprenderán en este curso.
- Es un lenguaje especialmente apropiado para el análisis y diseño orientado a objetos.
- Es independiente del lenguaje de programación.
- Produce una serie de documentos (diagramas de diferentes clases, documentos de texto, código) que se llaman **artefactos.**

Ustedes ya saben un poquito de UML

- Los diagramas que vimos con la programación orientada a objetos eran UML.
- Como el UML es independiente del lenguaje de programación, esto me permitió enseñarles la orientación a objetos sin enseñarles ningún lenguaje orientado a objetos.
- Esta es una de las múltiples ventajas del

Un momento, por favor.

- Aprender UML no quiere decir aprender análisis y diseño.
- UML sólo es la notación o lenguaje en que se expresa el análisis y diseño.
- Se necesita saber UML, pero no sólo eso.
- De la misma forma, aprender español no quiere decir aprender a escribir una novela.
- Para escribir una novela, hace falta saber español,

Hay que distinguir

- El análisis y diseño que es un proceso de descubrimiento de la estructura del programa.
- El UML es un lenguaje para expresar el análisis y diseño.
- ¿Qué aprenderemos en este curso?
- Aprenderemos los dos.

Mail: palasi@palasi.com Web: www.palasi.com

El UML es muy amplio

- Tiene más de un centenar de artefactos.
- Ahora bien, estos **artefactos** son opcionales, no obligatorios.
- Hay que escoger sólo los artefactos que sean necesarios en un momento dado y olvidar los otros.
- UML ofrece muchos artefactos para escojamos los más adecuados, no para que los implementemos todos.
- En proyectos que no son enormes, sólo

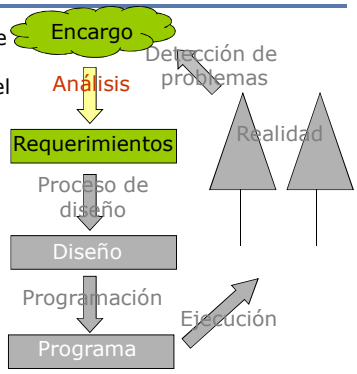
Este es un curso limitado de tiempo

- No podemos aprender todo el UML, pues éste es muy amplio.
- No podemos aprender todas las técnicas de análisis y diseño O-O, pues no hay tiempo.
- Pero aprenderemos lo fundamental de ambos aspectos.

Análisis de un sistema

Es el proceso que investiga las características del sistema a desarrollar.

Requiere entrevistas con los usuarios, consulta de documentos de la empresa, etc.



Metodología del curso

- Se les explicará un procedimiento de análisis y diseño paso a paso.
- Para cada paso, se les enseñará la parte de UML que permite expresarlo.
- Así, la enseñanza de análisis y diseño y de UML se entremezclarán.

Temario del curso

- 1. Introducción al A & D O-O y a UML.
- 2. Análisis orientado a objetos con UML.
- 3. Diseño orientado a objetos con UML.
- 4. Conclusiones finales.

Parte 2: Análisis orientado a objetos con UML

- 2.1. Introducción al análisis orientado a objetos.
- 2.2. Introducción a los casos de uso.
- 2.3. Completando casos de uso.
- 2.4. Precondiciones y postcondiciones.
- 2.5. Un método para detectar los casos de uso.
- 2.6. Diagramas de casos de uso.
- 2.7. Diagramas de secuencia del sistema.
- 2.8. Modelo de dominio: diagrama de clases conceptuales.

Mail: palasi@palasi.com Web: www.palasi.com

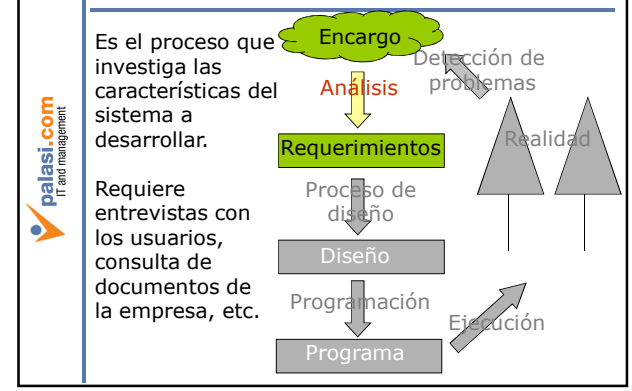
Parte 2: Análisis orientado a objetos con UML

- 2.1. Introducción al análisis orientado a objetos.
- 2.2. Introducción a los casos de uso.
- 2.3. Completando casos de uso.
- 2.4. Precondiciones y postcondiciones.
- 2.5. Un método para detectar los casos de uso.
- 2.6. Diagramas de casos de uso.
- 2.7. Diagramas de secuencia del sistema.
- 2.8. Modelo de dominio: diagrama de clases conceptuales.

Análisis de un sistema

Es el proceso que investiga las características del sistema a desarrollar.

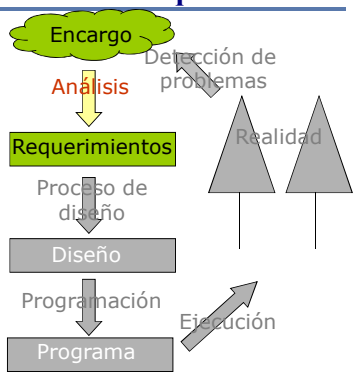
Requiere entrevistas con los usuarios, consulta de documentos de la empresa, etc.



El análisis acaba con el documento de requerimientos

El documento de requerimientos indica QUÉ va a hacer el programa.

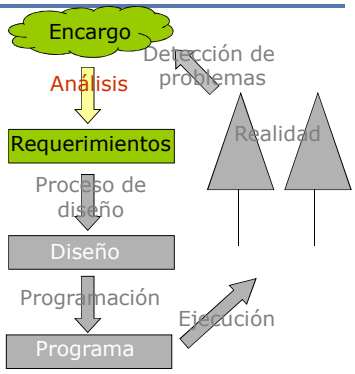
Por lo tanto, todo el proceso de análisis debe estar enfocado en las características: **en el QUÉ, no en el CÓMO.**



El análisis es la parte más importante del desarrollo

Se debe tener claro el sistema que se quiere obtener antes de empezar nada.

Primero debemos saber adónde llegar antes de comenzar el camino.

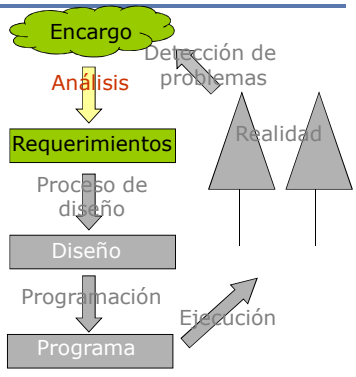


El análisis es la parte más importante del desarrollo

Los errores en el análisis son los **más costosos** en tiempo y dinero de solucionar.

Aún así, muchas veces es difícil conseguir un análisis correcto a la primera.

Por ello, es recomendable un modelo iterativo.

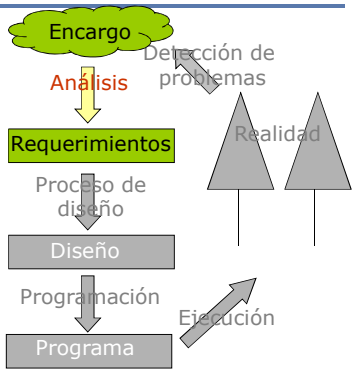


El análisis es la parte más importante del desarrollo

Los errores en el análisis son los **más costosos** en tiempo y dinero de solucionar.

Aún así, muchas veces es difícil conseguir un análisis correcto a la primera.

Por ello, es recomendable un modelo iterativo.



Mail: palasi@palasi.com Web: www.palasi.com

Los requerimientos pueden ser de estos tipos (FURPS+)

- **Funcionales:** características, capacidades, seguridad.
- **Usabilidad:** diseño usable, ayuda, documentación.
- **Confiabilidad:** frecuencia de fallo, recuperabilidad, predecibilidad.
- **Rendimiento:** tiempo de respuesta, uso de recursos.
- **Soportabilidad:** adaptabilidad, mantenibilidad, internacionalización, configurabilidad.
- **Otros:** implementación, interfaz con otros sistemas, operacionales, empaquetado y distribución, legales, etc.

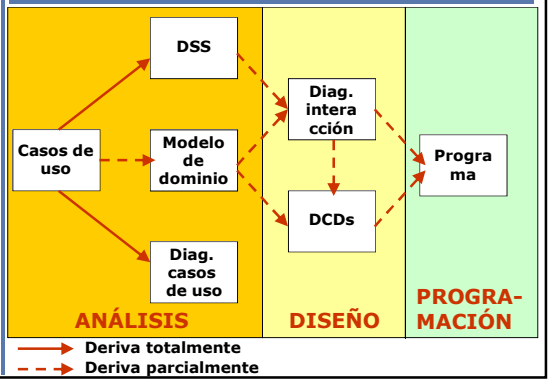
Esto se puede dividir en dos tipos de requerimientos

- **Requerimientos funcionales:** indican las funciones que debe efectuar el sistema (lo que debe hacer).
  - El sistema deberá generar la planilla del seguro.
  - El sistema deberá almacenar los datos de los clientes.
- **Requerimientos no funcionales:** son condiciones que debe cumplir el sistema y que no son funcionales:
  - El sistema deberá estar escrito en Java.
  - Los datos del sistema deberán almacenarse en una base de datos Oracle.
  - El sistema deberá validar una tarjeta de crédito en menos de 3 segundos.
  - El sistema tendrá ayuda en línea.

El análisis en UML

- El análisis en UML consiste en varios artefactos.
- Nosotros sólo veremos tres de estos artefactos:
  - Casos de uso.
  - Modelo de dominio: diagrama de casos conceptuales.
  - Diagramas de secuencia del sistema.

Método de desarrollo (se obvian los aspectos iterativos)



Utilidad de estos artefactos

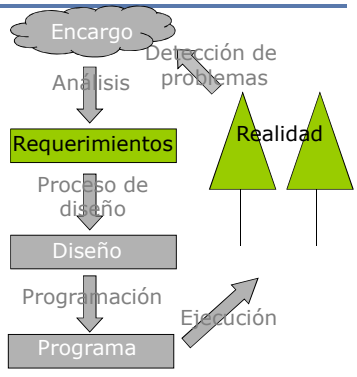
- **Casos de uso.** Son el principal artefacto para expresar los requerimientos funcionales en UML (también pueden expresar requerimientos no funcionales).
- **Diagramas de secuencia del sistema.** Sirven para expresar la interacción entre el sistema y los diferentes usuarios. Pueden considerarse un anexo a los casos de uso.
- **Modelo de dominio.** diagrama de casos conceptuales. Sirve para expresar la estructura del dominio del sistema.

Concepto de dominio

El dominio **es la parte** de la realidad asociada con el sistema.

Por ejemplo, en un sistema de planilla, el dominio es la política de planillas.

En un sistema eléctrico, el dominio son las instalaciones eléctricas, etc.

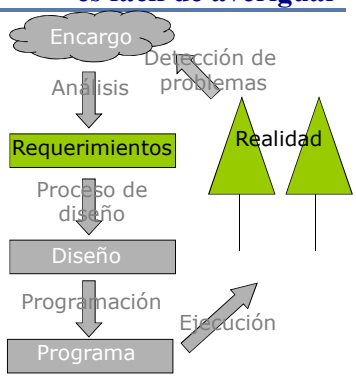


Mail: palasi@palasi.com Web: www.palasi.com

En general, el dominio es fácil de averiguar

Si vemos la expresión **sistema de X** o **sistema para X** es que el dominio del sistema es X.

¿Cuál sería el dominio de un sistema de contabilidad?

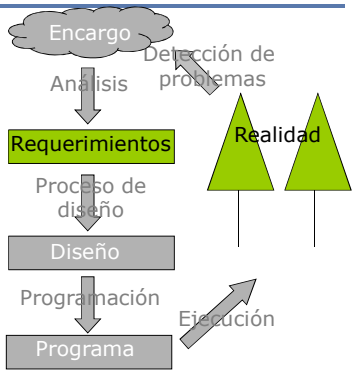


Es importante conocer el dominio

Pues debemos saber la realidad que deseamos mecanizar.

Además, en O-O, el programa modela la realidad.

El dominio lo expresamos en el diagrama de casos conceptuales, o modelo del dominio.



Parte 2: Análisis orientado a objetos con UML

- 2.1. Introducción al análisis orientado a objetos.
- 2.2. Introducción a los casos de uso.
- 2.3. Completando casos de uso.
- 2.4. Precondiciones y postcondiciones.
- 2.5. Un método para detectar los casos de uso.
- 2.6. Diagramas de casos de uso.
- 2.7. Diagramas de secuencia del sistema.
- 2.8. Modelo de dominio: diagrama de clases conceptuales.

¿Qué es un caso de uso?

- Jacobson:
  "Una secuencia de interacciones relacionadas por comportamiento ejecutada por un actor en un diálogo con el sistema que provee un resultado medible al actor".

¡¡Uf!!

Una definición más digerible

- **Son historias de cómo se usa un sistema para conseguir un objetivo.**
- Ejemplo. Sistema de planilla. Objetivos de un sistema:
  - Registrar los datos de los clientes.
  - Obtener la planilla del seguro.
  - Obtener los descuentos de un empleado dado.
- Los objetivos pueden ser unos u otros, dependiendo de las características del sistema.

Casos de uso: Historias sobre cómo conseguir objetivos

- La historia para obtener la planilla del seguro (simplificada):
  - El Operador del sistema selecciona la opción de seleccionar la planilla del seguro.
  - El Sistema pide el mes y el año.
  - El Operador introduce el mes y el año.
  - El Sistema presenta la planilla del seguro.
- ¿Cuál era el objetivo? Obtener la planilla del seguro.
- ¿Cuál es el caso de uso? Lo que está en rojo.

Mail: palasi@palasi.com Web: www.palasi.com

Casos de uso: Historias sobre cómo conseguir objetivos

- La historia para obtener la planilla del seguro:
  - El Operador del sistema selecciona la opción de obtener la planilla del seguro.
  - El Sistema pide el mes y el año.
  - El Operador introduce el mes y el año.
  - El Sistema presenta la planilla del seguro.
- ¿Cuál era el objetivo? Obtener la planilla del seguro.
- ¿Cuál es el caso de uso? Lo que está en rojo

Casos de uso: Historias sobre cómo conseguir objetivos

- La historia para inscribir un nuevo empleado:
  - El Empleado llega al Departamento de Informática.
  - El Operador del sistema selecciona la opción de inscribir la planilla del seguro.
  - El Sistema pide los datos del empleado.
  - El Operador introduce los datos del empleado.
  - El Sistema presenta los datos y pide una confirmación.
  - El Usuario introduce la confirmación.
  - El Sistema **guarda la información** e imprime un recibo para el nuevo empleado.
- ¿Cuál era el objetivo? Inscribir un nuevo empleado.
- ¿Cuál es el caso de uso? Lo que está en rojo

Visto de otra manera

- Los casos de uso son la respuesta a la pregunta "¿Qué pasos deben seguirse para conseguir el objetivo O?"
- ¿Qué pasos deben seguirse para inscribir un nuevo empleado?
- ¿Qué pasos deben seguirse para obtener la planilla del seguro?
- **Un caso de uso es una serie de pasos que hay que seguir para conseguir un objetivo**.

Los casos de uso no deben especificar la tecnología

- Caso de uso de inscribir un nuevo empleado (simplificado):
  - Se selecciona la opción de inscribir empleado.
  - Se introducen los datos del empleado.
  - El Sistema guarda los datos.
- Esto no es un caso de uso:
  - Se hace clic en el menú de inscribir empleado.
  - Se ponen los datos en los cuadros de texto.
  - Se guardan los datos en la base de datos.

Los casos de uso no deben especificar la tecnología

Menú, Cuadro de texto, Base de Datos son términos tecnológicos.

En general, los casos de uso no hacen referencia a la interfaz de usuario, a la base de datos ni a cualquier otra tecnología.

- Esto no es un caso de uso:
  - Se hace clic en **el menú** de inscribir empleado.
  - Se ponen los datos **en los cuadros de texto**.
  - Se guardan los datos en **la base de datos**.

Los casos de uso no deben especificar la tecnología

En general, los casos de uso no hacen referencia a la interfaz de usuario, a la base de datos ni a cualquier otra tecnología.

¿Por qué?

palasi.com
IT and management

Recordemos qué son los requerimientos

- Los requerimientos explican QUÉ (hace el sistema) y no CÓMO (lo hace).
- Un caso de uso es parte de los requerimientos.
- Decir: el sistema guarda los datos del empleado es el QUÉ.
- Es independiente si lo hacemos en una base de datos, en un archivo, en papel o en dónde sea: Esto es el CÓMO.

Los casos de uso no deben especificar la tecnología

- **Frases que suelen usarse**:
  - Para salida de información:
    El Sistema presenta (también señala, emite)
  - Para entrada de información:
    El Usuario entra
  - Para almacenamiento de información:
    El Sistema guarda (también registra).
  - Para salida de información en papel:
    El Sistema imprime.

(Nota: esto no es tecnología, podría ser por impresora, plotter o "a mano" (si el sistema la tuviera). Aquí "imprime" es una forma corta de decir: se genera una copia en papel)

Sin embargo, cuando el sistema interactúa con otros sistemas

- A veces es difícil evitar un nivel mínimo de tecnología.
- Así:
  - Paga con tarjeta de crédito (si el procedimiento de pago es diferente con esta modalidad).
  - Envía la información al sistema X (aunque mejor: envía al sistema que hace Y).
  - Y otros similares.

  están permitidos, aunque es mejor evitarlos si se puede.
- Sin embargo, no se debe ir más allá de esto ¡¡NO!!
  - Paga con American Express.
  - Introduce la tarjeta en el lector de tarjetas.

Recordemos qué son los requerimientos

- Los requerimientos explican QUÉ (hace el sistema) y no CÓMO (lo hace).
- Un caso de uso es parte de los requerimientos.
- Decir: se selecciona la opción es el QUÉ.
- Es independiente si lo hacemos con un menú, un hiperenlace, con una pantalla táctil o con comandos de voz (o con el pensamiento). Esto es el CÓMO.

Nombrando los casos de uso

- Los casos de uso tienen nombres sencillos.
- Los nombres indican el objetivo que se cumple con este caso de uso.
- Suelen estar compuestos por un verbo y un nombre.
- Son lo más cortos posibles y cada palabra va en mayúsculas.
- Por ejemplo:
  - Obtiene Planilla Seguro .
  - Inscribe Empleado .

Un caso de uso es una historia

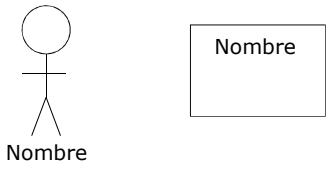
- **Toda historia tiene unos protagonistas.**
- En el caso de uso "Obtiene Planilla Seguro"
  - El Operador del sistema selecciona la opción de obtener la planilla del seguro.
  - El Sistema pide el mes y el año.
  - El Operador introduce el mes y el año.
  - El Sistema presenta la planilla del seguro.
- ¿Cuáles son los protagonistas de esta historia?
- Los protagonistas de este caso de uso son el Operador y el Sistema. Los escribimos en

Mail: palasi@palasi.com Web: www.palasi.com

Los protagonistas de los casos de uso se dividen en dos clases diferentes

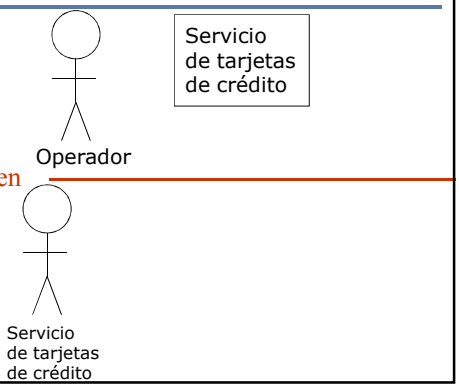
- **El Sistema** (es siempre el protagonista principal de la historia).
- Los otros protagonistas. Se les llama **Actores**.
- Un Actor es una entidad *externa* que interactúa con el sistema.
- Puede ser:
  - Una persona.
  - Otro sistema o máquina.
  - Existe el actor "Tiempo" (como veremos).

Un actor se representa con la siguiente notación



- Se puede utilizar cualquiera de las dos notaciones.
- Es común utilizar la primera notación para actores humanos y la segunda para otros sistemas, pero esto no es un estándar.

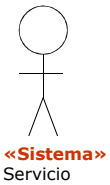
Ejemplo



A veces, se utiliza la siguiente notación

- Se utilizan comillas francesas para distinguir que es un sistema.
- Si nos resulta difícil escribir las comillas francesas podemos usar "<<" y ">>"

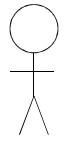
Estereotipo



- A la parte encerrada entre comillas francesas, se le llama **estereotipo.**
- Un estereotipo sirve para categorizar de alguna manera un elemento de UML.
- Se puede usar en cualquier elemento de UML y tiene muchos usos, como veremos más

El actor "Tiempo"

- Hay tareas que se ejecutan cada cierto periodo de tiempo sin que ningún actor las inicie.
- Suelen ser tareas de mantenimiento, respaldo, etc.
- Estas tareas se le atribuyen al actor "Tiempo"



Mail: palasi@palasi.com Web: www.palasi.com

Atención

- Una tentación grande es pensar que "Usuario" es un actor.
- Esto puede ser cierto en algunos casos, pero hemos de pensar que, para la mayoría de los sistemas, hay diferentes tipos de usuario.
- Está el Operador, el Gerente General, el Administrador del Sistema.
- **Cada uno de ellos es un actor diferente.**

Atención

- Tanto los actores como el sistema se escriben siempre con la primera letra mayúscula.
- Esto es para diferenciar que son los protagonistas de un caso de uso.
- Esto nos lleva a plantear cuál es la notación para los casos de uso.

Notación para los casos de uso

- Los casos de uso son historias, que se expresan como texto. **No son diagramas.**
- Pero este texto no puede ser cualquier texto: debe seguir un formato.
- Explicar todo el formato de golpe sería antipedagógico.
- Comenzaremos por explicar una versión simplificada de este formato.
- Después iremos refinando hasta acabar explicando el formato completo.

Formato (simplificado) de caso de uso

Nombre: *<Nombre del caso de uso>*
Actor primario:
<Aquí el actor principal del caso de uso>

Escenario principal:
<Aquí vienen los diferentes pasos que forman el caso de uso, numerados consecutivamente>

Un momento

- ¿Qué es un escenario?
- Un escenario es una **secuencia de pasos** (de interacción entre los actores del sistema).
- Le llamamos principal porque veremos que hay otros. Por ahora, veámoslo como que fuera "escenario", sin más.

Ejemplo

- Supongamos un sistema en que un cliente pide un crédito en línea a un banco.

Nombre: Otorga Crédito.
Actor primario: Cliente.
Escenario principal:
1. El Cliente selecciona la opción de crédito.
2. El Sistema pide el monto del crédito y el número de libreta para depositar.
3. El Cliente entra el monto del crédito y el número de libreta.
4. El Sistema registra el crédito, deposita el monto en la libreta y presenta que el crédito ha sido

Cada uno de los pasos tiene un protagonista que es sujeto de la frase

- Esto es una buena práctica con casos de uso. (El protagonista es el sistema u otro actor).

Nombre: Otorga Crédito.
Actor primario: Cliente.
Escenario principal:
1. **El Cliente** selecciona la opción de crédito.
2. **El Sistema** pide el monto del crédito y el número de libreta para depositar.
3. **El Cliente** entra el monto del crédito y el número de libreta.
4. **El Sistema** registra el crédito, deposita el monto en la libreta y presenta que el crédito ha sido

Recordemos: un caso de uso no especifica la tecnología

- Veamos los pasos 1 y 4 de ejemplo. El sistema podría estar implementado en una página Web y cajero automático.

1. El Cliente selecciona la opción de crédito (**¿hiperenlace?¿botones de cajero?**).

4. El Sistema registra el crédito (**¿con una BD? ¿otro sistema?**).

Formato (simplificado) de caso de uso

Nombre: *<Nombre del caso de uso>*
Actor primario:
<Aquí el actor principal del caso de uso>

Escenario principal:
<Aquí vienen los diferentes pasos que forman el caso de uso, numerados consecutivamente>

¿Qué falta aquí? ¿Qué es lo que faltaba en el ejemplo anterior?

¿Qué pasa si el cliente no tiene derecho a crédito?

- Un banco no se puede mantener si todos los créditos que se solicitan son otorgados automáticamente.
- Hay que contemplar la opción en el que al cliente se le deniega el crédito.
- ¿Por qué no hemos incluido esta opción en el "Escenario Principal"?

El escenario principal es una historia de éxito

- Cada caso de uso tiene un objetivo. El escenario principal es el escenario (los pasos) **en el que se consigue ese objetivo.**
- El escenario principal es el caso en que todo sale bien y el proceso se lleva hasta el final.
- En el escenario principal no puede haber un problema: es el escenario de un mundo perfecto ("the happy path").
- Es por ello que no podemos incluir problemas como este en el escenario principal.

Lo incluimos en los escenarios alternativos

Nombre: *<Nombre del caso de uso>*
Actor primario:
<Aquí el actor principal del caso de uso>

Escenario principal:
<Aquí vienen los pasos que forman la historia en que se consigue el objetivo>

Escenarios alternativos
<Aquí se especifican todas las otras posibilidades y condiciones que se pueden dar y que no se recogen en el escenario principal>

Ejemplo

Nombre: Otorga Crédito.
Actor primario: Cliente.
Escenario principal:
1. El Cliente selecciona la opción de crédito.
2. El Sistema pide el monto del crédito y el número de libreta para depositar.
3. El Cliente entra el monto del crédito y el número de libreta.
4. El Sistema registra el crédito, deposita el monto en la libreta y presenta que el crédito ha sido concedido.

Escenarios alternativos:
4a. El Cliente no tiene derecho a crédito.
1. El Sistema presenta rechazo del crédito.

Algunas observaciones

- Fíjense que el escenario alternativo se numera como 4a.
- Es por que es una alternativa al punto 4 del escenario principal: es allí donde se produce.

...
3. El Cliente entra el monto crédito y nº. de libreta.
4. El Sistema registra el crédito, deposita el monto en la libreta y presenta que el crédito ha sido concedido ...

Escenarios alternativos:
4a. El Cliente no tiene derecho a crédito.
1. El Sistema presenta rechazo del crédito.
2. El Sistema cancela el proceso de otorgar crédito.

Estructura de un escenario alternativo

Número

Condición que produce el escenario alternativo

4a. El Cliente no tiene derecho a crédito.
1. El Sistema presenta rechazo del crédito.
2. El Sistema cancela el proceso de otorgar crédito.

Pasos que hay que seguir para tratar esta condi-ción.

Fíjense que estos pasos se numeran igual que en el escenario principal, es decir, consecutivamente

Pueden haber otros escenarios alternativos en este punto

- Se numerarán como 4a, 4b, 4c, etc.

3. El Cliente entra el monto crédito y nº. libreta.
4. El Sistema registra el crédito, deposita el monto en la libreta y presenta que el crédito ha sido concedido

Escenarios alternativos:
4a. El Cliente no tiene derecho a crédito.
1. El Sistema presenta rechazo del crédito.
2. El Sistema cancela el proceso de otorgar crédito.

4b. El número de libreta no es correcto.
1. El Sistema señala error y rechaza entrada

4c. El monto del crédito no es correcto.

Incluso pueden haber escenarios alternativos en cualquier punto

- Se numerarán como *a, *b, etc.

Escenarios alternativos:
*a. En cualquier momento el Sistema falla.
1. El Administrador reinicia el Sistema.
2. El Sistema recupera su estado.

4a. El Cliente no tiene derecho a crédito

• • •

Otra posibilidad es que un escenario alternativo

- Pueda producirse en varios puntos pero no en todos.
- Se utilizan notaciones como 3-6a, 3-6b, si el escenario puede producirse entre los puntos 3 y 6 del escenario principal

Escenarios alternativos:
2-3a. El Cliente suspende el proceso.
1. El Sistema cancela la petición de crédito.

Caso de uso "Otorga Crédito" (casi) completo (1)

Nombre: Otorga Crédito.
Actor primario: Cliente.
Escenario principal:
1. El Cliente selecciona la opción de crédito.
2. El Sistema pide el monto del crédito y el número de libreta para depositar.
3. El Cliente entra el monto del crédito y el número de libreta.
4. El Sistema registra el crédito, deposita el monto en la libreta y presenta que el crédito ha sido concedido

Escenarios alternativos:
*a. En cualquier momento el Sistema falla.

Caso de uso "Otorga Crédito" (casi) completo (2)

2-3a. El Cliente suspende el proceso.
1. El Sistema cancela la petición de crédito.
4a. El Cliente no tiene derecho a crédito.
1. El Sistema presenta rechazo del crédito.
2. El Sistema cancela el proceso de otorgar crédito.
4b. El número de libreta no es correcto.
1. El Sistema señala error y rechaza entrada
4c. El monto del crédito no es correcto.
1. El Sistema señala error y rechaza entrada.

Supongamos que el sistema permite conceder varios créditos a la vez

- El caso de uso no se llama "Otorga Crédito" sino "Otorga Créditos".
- Ahora el cliente puede entrar varios montos y números de libreta para poder pedir varios créditos a la vez.
- ¿Cómo se reflejaría esto en un caso de uso?

Comencemos con el escenario principal

Nombre: Otorga Créditos.
Actor primario: Cliente.
Escenario principal:
1. El Cliente selecciona la opción de crédito.
2. El Sistema pide el monto del crédito y el número de libreta para depositar.
3. El Cliente entra el monto del crédito y el número de libreta.
4. El Sistema registra el crédito, deposita el monto en la libreta y presenta que el crédito ha sido concedido

El Cliente repite pasos 2-4 hasta que acaba.

5. El Sistema presenta un total de créditos

Esta es la forma de tratar tareas o pasos que se repiten

Nombre: Otorga Créditos.
Actor primario: Cliente.
Escenario principal:
1. El Cliente selecciona la opción de crédito.
2. El Sistema pide el monto del crédito y el número de libreta para depositar.
3. El Cliente entra el monto del crédito y el número de libreta.
4. El Sistema registra el crédito, deposita el monto en la libreta y presenta que el crédito ha sido concedido

El Cliente repite pasos 2-4 hasta que acaba.

5. El Sistema presenta un total de créditos

Fíjense que ahora la cancelación no deshace los créditos otorgados

Escenarios alternativos:
*a. En cualquier momento el Sistema falla.
1. El Administrador reinicia el Sistema.
2. El Sistema recupera su estado.
2-4a. El Cliente suspende el proceso.
1. El Sistema acaba el proceso de otorgar créditos.
3a. El Cliente no tiene derecho a crédito.
1. El Sistema presenta que el crédito no se puede otorgar.
3b. El número de libreta no es correcto.
1. El Sistema señala error y rechaza entrada
3c. El monto del crédito no es correcto.

Mail: palasi@palasi.com Web: www.palasi.com

Fíjense que ahora la denegación de crédito no cancela el proceso

Escenarios alternativos:

*a. En cualquier momento el Sistema falla.
   1. El Administrador reinicia el Sistema.
   2. El Sistema recupera su estado.
2-4a. El Cliente suspende el proceso.
   1. El Sistema acaba el proceso de otorgar créditos.
3a. El Cliente no tiene derecho a crédito.
   1. El Sistema presenta que el crédito no se puede otorgar.
3b. El número de libreta no es correcto.
   1. El Sistema señala error y rechaza entrada
3c. El monto del crédito no es correcto.

Ejercicio

- Especificar la función de un sistema de notas. El sistema permite entrar las diferentes notas del alumno en una materia, presenta la nota global (que es la media aritmética) e indica si está aprobado. Si lo está, imprime un certificado de haber pasado la materia.
- Se supone que el aprobado está en 6.

Parte 2: Análisis orientado a objetos con UML

- 2.1. Introducción al análisis orientado a objetos.
- 2.2. Introducción a los casos de uso.
- 2.3. Completando casos de uso.
- 2.4. Precondiciones y postcondiciones.
- 2.5. Un método para detectar los casos de uso.
- 2.6. Diagramas de casos de uso.
- 2.7. Diagramas de secuencia del sistema.
- 2.8. Modelo de dominio: diagrama de clases conceptuales.

Hasta ahora hemos trabajado con una versión simplificada de casos de uso

- Nos ha permitido enfocarnos en las ideas principales.
- Nos ha permitido aprender con facilidad.
- Hemos dejado aparte varios aspectos importantes que vamos a considerar ahora.

Formato completo de casos de uso

Nombre: *<Nombre del caso de uso>*
Actor primario: *<Actor principal>*
Interesados e intereses:
<Los interesados en que se ejecute el proceso junto con los objetivos de estos interesados>
Precondición: *<Precondición del caso de uso>*
Poscondición: *<Poscondición del caso de uso>*
Escenario principal:
<Escenario principal>
Escenarios alternativos:
<Escenarios alternativos>
Requerimientos especiales:

Parte que no hemos visto y que veremos ahora

Nombre: *<Nombre del caso de uso>*
Actor primario: *<Actor principal>*
Interesados e intereses:
<Los interesados en el proceso junto con los objetivos de estos interesados>
Precondición: *<Precondición del caso de uso>*
Poscondición: *<Poscondición del caso de uso>*
Escenario principal:
<Escenario principal>
Escenarios alternativos:
<Escenarios alternativos>
Requerimientos especiales:

Parte que no hemos visto y que veremos ahora

Nombre: *<Nombre del caso de uso>*
Actor primario: *<Actor principal>*
Interesados e intereses:
<Los interesados en el proceso junto con los objetivos de estos interesados>
Precondición: *<Precondición del caso de uso>*
Poscondición: *<Poscondición del caso de uso>*
Escenario principal:
<Escenario principal>
Escenarios alternativos:
<Escenarios alternativos>
Requerimientos especiales:

Frecuencia de uso

- ¿Con qué frecuencia se ejecutará el procedimiento que indicamos en el caso de uso?
- Por ejemplo, esporádicamente, de vez en cuando, frecuentemente, casi continuamente, continuamente.
- Este dato es importante saberlo para valorar la importancia del caso de uso y su efecto sobre los recursos (esto lo veremos más adelante, no en el análisis).

Parte que no hemos visto y que veremos ahora

Nombre: *<Nombre del caso de uso>*
Actor primario: *<Actor principal>*
Interesados e intereses:
<Los interesados en el proceso junto con los objetivos de estos interesados>
Precondición: *<Precondición del caso de uso>*
Poscondición: *<Poscondición del caso de uso>*
Escenario principal:
<Escenario principal>
Escenarios alternativos:
<Escenarios alternativos>
Requerimientos especiales:

Interesados

- Son personas, sistemas, empresas y organizaciones que tienen un interés en el proceso que describe el caso de uso.
- No es lo mismo que los actores.
- Los actores son los que interactúan con el sistema durante el caso de uso.
- Los interesados no tienen porqué interactuar con el sistema (aunque pueden hacerlo) pero siempre tienen intereses en él.

Ejemplo: Interesados en el programa de otorgar créditos en línea

- Cliente.
- Administrador.
- Banco.
- Superintendencia del Sistema Financiero.
- Ministerio de Hacienda.
- Sistema integrado del banco.

Muchos de estos interesados no son actores

- Cliente.
- Administrador.
- Banco. ⟵ NO ACTOR
- Superintendencia del Sistema Financiero. ⟵ NO ACTOR
- Ministerio de Hacienda. ⟵ NO ACTOR
- Sistema integrado del banco. ⟵ NO ACTOR

No son actores porque no interactúan con el Sistema de créditos. El sistema integrado no interactúa con él: sólo usa la misma BD.

Mail: palasi@palasi.com Web: www.palasi.com

Todos los interesados tienen intereses en el caso de uso

- Cliente. Interés: que se le otorgue el crédito.
- Superintendencia del Sistema Financiero. Interés: que se sigan las leyes.
- Sistema integrado del banco. Interés: que la información en la BD sea correcta.
- **Los intereses son diferentes, pero todos tienen intereses.**

Formato de la sección de "Interesados e intereses"

Interesados e intereses:
-Interesado 1: Intereses del Interesado 1.
-Interesado 2: Intereses del Interesado 2.
. . .
-Interesado n: Intereses del Interesado n.

- Como ven, es un formato muy sencillo.

Sección de "Interesados e intereses" para el programa de créditos (1)

Interesados e intereses:

- Cliente : Quiere obtener crédito, con la mayor rapidez posible y el mínimo esfuerzo. Quiere una comprobación en papel del otorgamiento del crédito.
- Administrador: Quiere que el proceso esté disponible el mayor tiempo posible y que cuando falle pueda restaurarse fácilmente.
- Banco. Quiere mejorar el servicio al cliente y aumentar el número de posibles clientes. Quiere que los créditos otorgados sean a personas solventes.

Sección de "Interesados e intereses" para el programa de créditos (2)

- Superintendencia del Sistema Financiero. Quiere comprobar que todos los créditos se otorguen dentro de la legalidad.
- Ministerio de Hacienda. Quiere recaudar impuestos de los créditos.
- Sistema integrado del banco. Quiere que la información del proceso sea registrada correctamente para procesarla con el sistema integrado.

¿Para qué sirve esta sección?

- Esta sección es mucho más importante de lo que parece a simple vista.
- Se recomienda escribirla antes que nada.
- ¿Por qué? Sugiere y limita lo que el sistema debe hacer.
- **"El caso de uso … captura *todos y sólo* los comportamientos relacionados con satisfacer los intereses de los interesados".** Writting Effective Use Cases, Allistair Cockburn.

¿Qué es lo que hay en un caso de uso?

- Problema es usual: a veces no se sabe qué incluir en un caso de uso y que excluir.
- La cita de Cockburn lo deja claro:
  - En el caso de uso debe haber **todo lo necesario para satisfacer los intereses de los interesados.**
  - **Y nada más.**
- Primero que nada, escribimos la sección de "Interesados e intereses":
  - Nos dará ideas para escribir el caso de uso.
  - Nos dará una prueba para comprobar que el caso de uso es correcto.

Mail: palasi@palasi.com Web: www.palasi.com

Nosotros ya hemos escrito nuestro caso de uso

- No podemos beneficiarnos de obtener ideas de la sección de "Interesados" para escribir nuestro caso de uso.
- Pero sí, podemos beneficiarnos de tener una prueba para ver si nuestro caso de uso es correcto.
- Vamos a repasar cada uno de los intereses de los interesados y comprobar si se cumple en el caso de uso que hemos escrito.

Repasando los intereses (1)

- **Cliente** : Quiere obtener crédito, con la mayor rapidez posible y el mínimo esfuerzo. Quiere una comprobación en papel del otorgamiento del crédito.
- El mínimo esfuerzo sí que parece que lo cumple el caso de uso, pero no parece especificarse nada sobre la **rapidez**. Tampoco se obtiene **comprobación en papel**.
- Son dos intereses que no están cubiertos y que deberían estar incluidos en el caso de uso.
- Después veré cómo los incorporo.

Repasando los intereses (2)

- **Administrador**: Quiere que el proceso esté disponible el mayor tiempo posible y que cuando falle pueda restaurarse fácilmente.
- Sobre la restauración, ya está incluida en nuestro caso de uso. Pero **la disponibilidad** no se dice nada, ~~deberemos incluirla.~~
- **Banco**. Quiere mejorar el servicio al cliente y aumentar el número de posibles clientes. Quiere que ~~los créditos otorgados sean a personas solventes.~~
- El servicio al cliente se mejora dando una forma fácil de acceder al crédito y esto aumenta el número de posibles clientes. El sistema comprueba que los créditos sean a personas que están autorizadas.

Repasando los intereses (3)

- Superintendencia del Sistema Financiero. Quiere comprobar que todos los créditos se otorguen dentro de la legalidad.
- Ministerio de Hacienda. Quiere recaudar impuestos de los créditos.
- Sistema integrado del banco. Quiere que la información del proceso sea registrada correctamente para procesarla con el sistema integrado.

Todo esto se cumple, ya que se registran los créditos y así se puede comprobar la legalidad, se puede deducir los impuestos y se puede "alimentar" otros sistemas del banco

Nuestro caso de uso no es perfecto

- No cubre los siguientes intereses:
  - Que el sistema haga un certificado en papel de otorgar el crédito.
  - Que el sistema sea rápido.
  - Que el sistema esté disponible el mayor tiempo posible.
- Esto es normal: sería extraño que se consiguiera un caso de uso perfecto a la primera.
- Escribir la sección "Interesados e intereses" nos ha servido para mejorarlo, ya que nos ha mostrado las deficiencias.

Mejorando el caso de uso

- Debemos incluir en él:
  - Que el sistema haga un certificado en papel de otorgar el crédito.
  - Que el sistema sea rápido.
  - Que el sistema esté disponible el mayor tiempo posible.
- El primer interés es fácil: basta con incluir esta información en el escenario principal que nos imprima el certificado.

¿Qué tal con los otros intereses?

- Que el sistema sea rápido.
- Que el sistema esté disponible el mayor tiempo posible.

- No parecen tan fáciles de incluir en el caso de uso.
- ¿Alguien tiene alguna idea de cómo se haría?

Esto son requerimientos no funcionales

- Recordemos:
- **Requerimientos funcionales:** indican las funciones que debe efectuar el sistema (lo que debe hacer).
- **Requerimientos no funcionales:** son condiciones que debe cumplir el sistema y que no son funcionales (usabilidad, confiabilidad, rendimiento, soportabilidad, etc.)
- Estos son requerimientos no funcionales:
  - Que el sistema sea rápido.
  - Que el sistema esté disponible el mayor tiempo

¿Dónde se incluyen los requerimientos no funcionales?

- Junto a los funcionales no, pues sería mezclar cosas muy diferentes.
- Por ello, no se incluyen en los escenarios.
- Tienen una sección en los casos de uso para este tipo de requerimientos.

Es la sección de requerimientos especiales

Nombre: *<Nombre del caso de uso>*
Actor primario: *<Actor principal>*
Interesados e intereses:
<Los interesados en el proceso junto con los objetivos de estos interesados>
Precondición: *<Precondición del caso de uso>*
Poscondición: *<Poscondición del caso de uso>*
Escenario principal:
<Escenario principal>
Escenarios alternativos:
<Escenarios alternativos>
Requerimientos especiales:

En esta sección, incluimos los requerimientos no funcionales

- Que pueden ser muy variados. En nuestro caso:

Requerimientos especiales:
- Que el sistema tenga un tiempo de respuesta de menos de dos segundos.
- Que el sistema tenga una disponibilidad del 99% del tiempo.

- Esto cubre los intereses de que el sistema sea rápido y disponible.

Un momento. ¿Por qué tanto detalle?

- Los intereses eran simplemente que el sistema fuera rápido y disponible, pero los requerimientos son mucho más específicos.
- Esto es bueno: ¿qué quiere decir que un sistema es rápido? Esto está sujeto a opiniones y es muy ambiguo. No es verificable.
- En cambio, si digo que el tiempo de respuesta es menor de 2 segundos esto es verificable.
- En general, cuanto más comprobables sean los requerimientos mejor.
- Esto tanto para los requerimientos funcionales como no funcionales.

Mail: palasi@palasi.com Web: www.palasi.com

Teniendo requerimientos verificables

- 1. Podemos tener una guía clara a la hora de desarrollar.
- 2. Podemos saber si un programa está correctamente implementado o no, respecto a los requerimientos.
- 3. Podemos exigir responsabilidades entre los programadores.

Recordemos: Los casos de uso no deben especificar la tecnología

En general, los casos de uso no hacen referencia a la interfaz de usuario, a la base de datos ni a cualquier otra tecnología.

- Esto no es cierto ahora.
- Los casos de uso pueden hacer referencia a la tecnología, **pero sólo en la sección de requerimientos especiales**.

Ejemplo de requerimientos tecnológicos que podemos incluir

- En la sección "Requerimientos especiales".
  - El sistema debe ser accesible mediante la Web.
  - El sistema debe usar una base de datos Oracle.
  - El sistema debe usar "middleware".
  - El sistema debe ser independiente de la plataforma

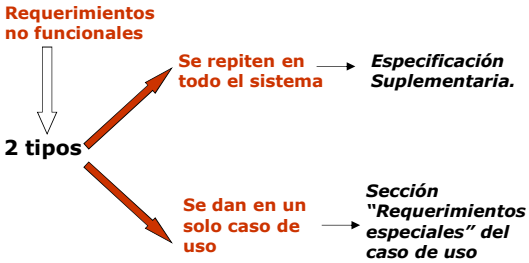
Los requerimientos no funcionales son muy importantes

- Examinemos uno de ejemplo:
- Que el sistema tenga una disponibilidad del 99% del tiempo.
- Esto puede ser determinante a la hora de
  - Elegir sistema operativo.
  - Elegir lenguaje de desarrollo.
  - Fragmentar el sistema en varios servidores paralelos (clustering).
  - Elegir una solución middleware como Enterprise Javabeans.

Los req. no funcionales están en "Req. especiales" del caso de uso

- Esto es verdad, pero no toda la verdad.
- Hay requerimientos no funcionales que se repiten en todos los casos de uso porque son generales al sistema.
- Por ejemplo, los que acabamos de ver.
- Estos se escriben en un artefacto UML llamado la "Especificación Suplementaria".
- Para el caso de uso, sólo se dejan los requerimientos no funcionales asociados con el caso de uso.

Dicho de otra manera



Mail: palasi@palasi.com Web: www.palasi.com

Nosotros no estudiaremos la Especificación Suplementaria

- Por falta de tiempo.
- Por lo tanto suponemos que todos los requerimientos no funcionales se escriben en la sección correspondiente en los casos de uso.
- Pero es útil que sepan la existencia de Especificación Suplementaria.

Lo más importante de lo último que hemos estudiado

- La sección "Frecuencia de uso" nos indicará cuán frecuentemente se usa el proceso que modela el uso de caso.
- La sección "Interesados e intereses" nos sirve para:
  - Dar ideas para escribir el caso de uso.
  - Para saber si éste es correcto o no o refinarlo.
- Los requerimientos no funcionales se registran en la sección "Requerimientos especiales".

Ejercicio

- Añadan estas tres secciones al caso de uso "Registrar Notas", que han comenzado a escribir antes.

Parte 2: Análisis orientado a objetos con UML

- 2.1. Introducción al análisis orientado a objetos.
- 2.2. Introducción a los casos de uso.
- 2.3. Completando casos de uso.
- 2.4. Precondiciones y postcondiciones.
- 2.5. Un método para detectar los casos de uso.
- 2.6. Diagramas de casos de uso.
- 2.7. Diagramas de secuencia del sistema.
- 2.8. Modelo de dominio: diagrama de clases conceptuales.

Ya sólo nos queda por ver

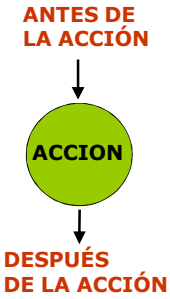
Nombre: *<Nombre del caso de uso>*
Actor primario: *<Actor principal>*
Interesados e intereses:
<Los interesados en el proceso junto con los objetivos de estos interesados>
Precondición: *<Precondición del caso de uso>*
Poscondición: *<Poscondición del caso de uso>*
Escenario principal:
<Escenario principal>
Escenarios alternativos:
<Escenarios alternativos>
Requerimientos especiales:

¿Qué es una precondición y una poscondición?

- El término no se aplica sólo a los casos de uso, sino a otros muchos campos.
- Supongamos que tenemos una acción o secuencia de acciones (como la que define un caso de uso).
- Cualquier acción o secuencia de acciones cambia las cosas.
- Hay una diferencia entre un antes y un después de la acción.

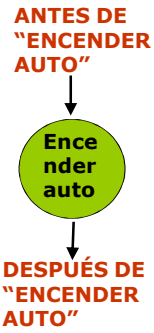
Mail: palasi@palasi.com Web: www.palasi.com

Hay un antes y después de una acción



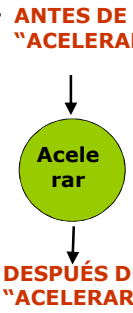
La acción "Encender auto"

- Antes de la acción "encender auto", las cosas son de una manera (el auto está apagado) y después de la acción "encender auto" son de otras (el auto está encendido)



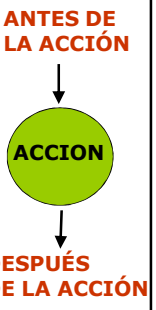
La acción "Acelerar"

- Antes de la acción "acelerar", las cosas son de una manera (el auto va a una velocidad) y después de "acelerar" las cosas son de otras (va a una velocidad más alta)



Precondición y Poscondición

- Precondición:
  - Condiciones que deben cumplirse antes de ejecutar la acción.
- Poscondición:
  - Condiciones que deben cumplirse después de ejecutar la acción.

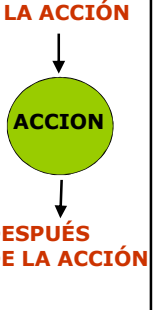


Pero ejecutar una acción puede hacerse de muchas maneras posibles

- Aquí sólo nos interesaran las maneras correctas de ejecutar una acción.
- Todos tenemos una idea de lo que es ejecutar una acción "correctamente".
- Más adelante, definiremos más específicamente qué queremos decir con esto.
- Por ahora, sólo concentremos en que la acción debe ejecutarse correctamente

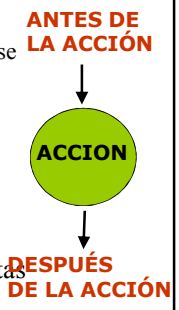
Precondición y Poscondición

- Precondición:
  - Condiciones que deben cumplirse antes de ejecutar la acción **correctamente**.
- Poscondición:
  - Condiciones que deben cumplirse después de ejecutar la acción **correctamente**.



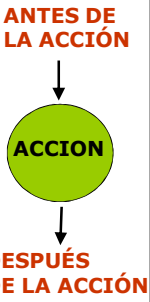
Veámoslo con más detalle

- Precondición:
  - Condiciones que **deben** cumplirse antes de ejecutar la acción correctamente.
- ¿Qué quiere decir esto de "DEBEN"?
- ¿Qué pasa si no se cumplen estas condiciones?



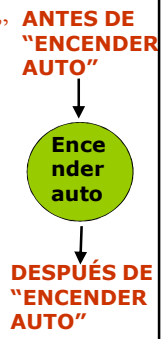
Si estas condiciones, no se cumplen

- O bien la acción no puede ejecutarse.
- O bien la acción no se ejecuta correctamente.
- Precondición:
  - Condiciones que **deben** cumplirse antes de ejecutar la acción correctamente.



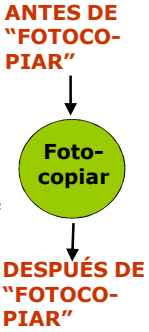
Veamos un ejemplo

- Precondición de "encender auto" (condiciones que **deben** cumplirse antes de ejecutar la acción correctamente):
  - La batería debe estar cargada.
  - El auto debe tener gasolina
- ¿Qué pasa si no se cumplen?
  - La acción no puede ejecutarse.
  - No puede encenderse el auto.



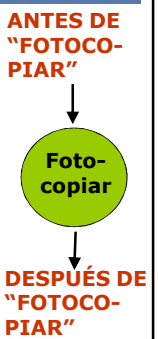
Veamos otro ejemplo

- Precondición de "fotocopiar" (condiciones que **deben** cumplirse antes de ejecutar la acción correctamente):
  - La fotocopiadora debe estar encendida.
  - El "toner" debe contener suficiente tinta.
- ¿Qué pasa si no se cumplen?



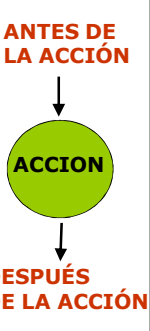
Si no se cumple la precondición

- Si la fotocopiadora no está encendida, no se puede fotocopiar (no se puede realizar la acción).
- Si el "toner" no contiene suficiente tinta, se fotocopiará pero la copia será ilegible por muy clara (la acción no se realiza correctamente).



Resumiendo: Precondición

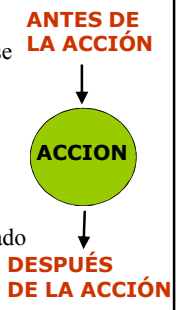
- Condiciones que **deben** cumplirse antes de ejecutar la acción correctamente. Si no se cumplen:
  - O bien la acción no puede ejecutarse.
  - O bien la acción no se ejecuta correctamente.



Mail: palasi@palasi.com Web: www.palasi.com

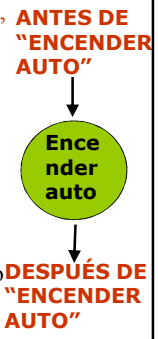
Poscondición

- Poscondición:
  - Condiciones que deben cumplirse después de ejecutar la acción **correctamente**.
- Si no se cumplen:
  - Es que la acción no se ha ejecutado correctamente.



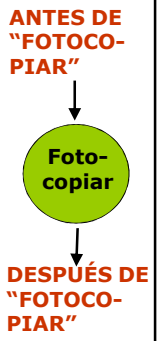
Veamos un ejemplo

- Poscondición de "encender auto" (condiciones que **deben** cumplirse después de ejecutar la acción correctamente):
  - El auto está encendido.
- ¿Qué pasa si no se cumplen?
  - Es señal de que la acción "encender auto" no se ha ejecutado correctamente.



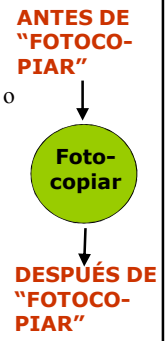
Veamos otro ejemplo

- Poscondición de "fotocopiar" (condiciones que **deben** cumplirse después de ejecutar la acción correctamente):
  - Hay una copia legible del original
- ¿Qué pasa si no se cumple? Es señal de que la acción "fotocopiar" no se ha ejecutado correctamente.



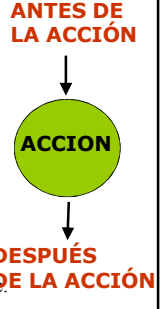
¿Por qué la acción se ejecuta incorrectamente?

- Hay dos motivos posibles:
  - No se cumple la precondición.
  - La acción está mal implementada o ejecutada.
- En el caso de la fotocopia:
  - A. La fotocopiadora no está encendida y no tiene toner.
  - B. La fotocopiadora tiene problemas de funcionamiento.



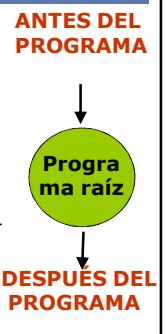
Resumiendo: Precondición y Poscondición

- Precondición:
  - Condiciones que deben cumplirse **antes** de ejecutar la acción correctamente.
  - Si no se cumplen, la acción no podrá ejecutarse correctamente.
- Poscondición:
  - Condiciones que deben cumplirse **después** de ejecutar la acción correctamente.
  - Si no se cumplen, es señal de que la acción no se ha ejecutado correctamente.



En el caso de un programa que obtenga la raíz cuadrada

- Precondición:
  - El número que se entra debe ser cero o positivo.
- Poscondición:
  - El resultado contiene la raíz cuadrada del número que se entró.



En el caso de un programa que escriba "Hola a todos"

- Precondición:
  - Este programa no tiene precondición, pues no hace falta ninguna condición para ejecutarlo.
- Poscondición:
  - Hay un mensaje en la pantalla que dice "Hola a todos"

Como vemos, la precondición puede no existir. En cambio, la poscondición existe siempre.

Un caso de uso tiene una precondición y una poscondición

- Un caso de uso especifica una secuencia de acciones que cambian la realidad.
- La precondición del caso de uso son las condiciones que deben cumplirse antes de ejecutar esa secuencia de acciones de forma correcta.
- La poscondición del caso de uso son las condiciones que deben cumplirse después de ejecutar esa secuencia de acciones de forma correcta.

Caso de uso "Otorga Crédito" (simplificado)

Nombre: Otorga Crédito.
Actor primario: Cliente.
Escenario principal:
1. El Cliente selecciona la opción de crédito.
2. El Sistema pide el monto del crédito y el número de libreta para depositar.
3. El Cliente entra el monto del crédito y el número de libreta.
4. El Sistema registra el crédito, deposita el monto en la libreta, presenta que el crédito ha sido concedido e imprime un certificado para el cliente.

¿Qué precondiciones tiene este caso de uso?

Si se fijan, nunca se pide el nombre del cliente

- Se supone que dado el número de libreta, conocemos quién es el cliente.
- Sin embargo, para otorgar un crédito, sólo dejaremos pedirlo al propietario de la libreta.
- ¿Cómo sabemos que quien está operando el sistema es realmente el dueño de la libreta? (y no un ladrón, por ejemplo)
- Para ello debemos conocer la identidad del cliente que opera el sistema y debemos saber que el cliente es quien dice ser.

Saber la identidad del cliente y saber que es quién dice ser

- Condición que se requiere para poder ejecutar el caso de uso "Otorga Crédito".
- Esta sería la precondición.
- Precondición: El Cliente ha sido identificado y autenticado.
- Esto se puede hacer de muchas maneras: con clave y contraseña, con tarjetas, con dispositivos biométricos.

La poscondición de un caso de uso

- Son las condiciones que se cumplen **después** de que éste haya acabado correctamente.
- Se supone que el caso de uso acaba correctamente si no falla, si no se cancela o aborta a medio camino, es decir si llega hasta su final.

Poscondición de "Otorgar crédito"

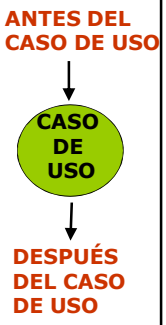
- Poscondición es la condición que se cumple después de que el caso de uso "Otorgar Crédito" acabe correctamente.
- Poscondición: Si el Cliente estaba autorizado al crédito,
  - el crédito está registrado
  - en la libreta del Cliente está depositado el monto del crédito
  - existe un certificado del crédito en papel para el cliente

Dicho de otra manera

- La poscondición:
  - Los efectos del caso de uso cuando éste acaba correctamente.
  - Cuando la ejecución del caso de uso acaba correctamente, ¿qué nos queda de consecuencia?
- Si el caso de uso es correcto, la poscondición debe cumplir los intereses de los interesados (pero sólo los requerimientos funcionales).

Resumiendo: Precondición y poscondición de un caso de uso

- Precondición:
  - Condiciones que deben cumplirse **antes** de ejecutar el caso de uso correctamente.
  - Si no se cumplen, el caso de uso no podrá ejecutarse correctamente.
- Poscondición:
  - Condiciones que deben cumplirse **después** de ejecutar el caso de uso correctamente.
  - Si no se cumple, es señal de que el caso de uso no ha acabado correctamente.



Ejercicio

- Obtengan la precondición y poscondición del caso de uso "Registrar notas".

Parte 2: Análisis orientado a objetos con UML

- 2.1. Introducción al análisis orientado a objetos.
- 2.2. Introducción a los casos de uso.
- 2.3. Completando casos de uso.
- 2.4. Precondiciones y postcondiciones.
- 2.5. Un método para detectar los casos de uso.
- 2.6. Diagramas de casos de uso.
- 2.7. Diagramas de secuencia del sistema.
- 2.8. Modelo de dominio: diagrama de clases conceptuales.

Recordemos la precondición del caso de uso "Otorga Crédito"

- Precondición: El Cliente ha sido identificado y autenticado.
- Esto quiere decir que esta condición debe cumplirse antes de ejecutar el caso de uso "Otorga Crédito".
- ¿Cómo nos aseguramos que se cumpla?

Mail: palasi@palasi.com Web: www.palasi.com

La identificación y autenticación se producen al inicio del programa

- Al inicio del programa, el cliente debe ser identificado y autenticado.
- La manera en que es identificado y autenticado no me interesa:
  - Clave y contraseña.
  - Tarjeta.
  - Huellas digitales.
  - Iris, etc.
- Lo único que me interesa es que, al ingresar al programa, el cliente es identificado y autenticado.

¿Cómo se especifica esto?

- Esto son requerimientos funcionales y, por lo tanto, deben especificarse en casos de uso.
- En nuestro caso, tendremos un caso de uso que resuelva el objetivo de ingresar al sistema.
- Este caso existirá en todos nuestros programas y normalmente se le llama el caso de uso "Inicio".

Ejemplo de un caso de uso "Inicio" (1)

Nombre: Inicio.
Actor primario: Cliente.
Interesados e intereses:
Cliente. Quiere ingresar al sistema y ser identificado de la forma más sencilla.
Banco: Quiere evitar entradas no autorizadas al sistema.
Superintendencia del Sistema Financiero: Quiere evitar la ejecución de operaciones fraudulentas por el método de suplantar identidad.
Ministerio de Hacienda: Quiere tener un registro de las operaciones bancarias de cada contribuyente.
Precondición: (No hay precondición)
Poscondición: El Cliente está autorizado y autenticado

Ejemplo de un caso de uso "Inicio" (2)

Escenario principal:
1. El Cliente accede al sistema.
2. El Sistema pide la identificación del Cliente.
3. El Cliente entra su identificación.
4. El Sistema presenta un mensaje indicando que el cliente ha sido identificado y autenticado.

Escenarios alternativos:
*a. En cualquier momento el Sistema falla.
   1. El Administrador reinicia el Sistema.
   2. El Sistema recupera su estado.
4a. El Cliente no ha podido ser identificado
   1. El Sistema señala error y rechaza la entrada.
   2. El Sistema pide una nueva identificación.
4b. El Cliente no ha podido ser autenticado.

Ejemplo de un caso de uso "Inicio" (3)

Requerimientos especiales:
Debe ser posible la identificación biométrica.
Debe ser posible la identificación por tarjetas.
El proceso de identificación debe tardar menos de dos segundos.
Frecuencia de uso: Continua

Ahora tenemos dos casos de uso en nuestro sistema

- El caso de uso "Inicio".
- El caso de uso "Otorga Crédito".
- Esto es normal: un sistema que no sea "de juguete" siempre está formado por varios casos de uso (normalmente muchos).
- Esto plantea la pregunta.

¿Cómo hallamos los casos de uso de un sistema?

- El modelado de caso de uso de un sistema consta de dos partes:
- 1. Identificar cuáles son los casos de uso del sistema.
- 2. Escribir los casos de uso con el formato que hemos explicado.
- La segunda parte ya hemos visto como se realiza. Pero, ¿y la primera parte?

¿Cómo identificamos los casos de uso de un sistema?

- Tenemos un sistema que deseamos especificar.
  - ¿Cómo hacemos para partir esta especificación en casos de uso?
  - ¿Cómo hacemos para identificar cada caso de uso y diferenciarlo de otros?
- Hay varios métodos pero nosotros sólo explicaremos uno de ellos.

Método para hallar los casos de uso del sistema

- 1. Definir los límites del sistema.
- 2. Encontrar los actores y sus objetivos.
- 3. Identificar los casos de usos.

Método para hallar los casos de uso del sistema

- 1. Definir los límites del sistema.
- 2. Encontrar los actores y sus objetivos.
- 3. Identificar los casos de usos.

1. Definir los límites del sistema

- Es decir, decidir qué funciones debe tener el sistema.
- Es decir, definir qué es **parte** del sistema y qué es **externo** al sistema.
- Como usamos un **método iterativo**, se comienza el modelado de casos de uso con una idea aproximada de los límites del sistema y cada vez se obtiene una idea más exacta.

1. Definir los límites del sistema

- Esto parece obvio pero muchos problemas se derivan de no definir con toda exactitud cuáles son los límites del sistema.
- Los límites del sistema tienen un enorme impacto sobre los requerimientos funcionales (y, a veces, no funcionales).
- Por ejemplo, los límites del sistema determinan cuáles serán los actores y los casos de uso.

1. Definir los límites del sistema

- Ejemplos:
  - En nuestro sistema de créditos, ¿es el sistema quien decide qué cliente está autorizado para un crédito o es alguien externo (algún gerente) quien lo decide?
  - En un sistema contable, ¿cómo se validan las tarjetas de crédito? ¿Esta función se incluye en el sistema o es realizada por un sistema externo?
  - En un sistema contable, ¿cómo se calculan los impuestos? ¿Se usa un sistema externo proporcionado por el Gobierno, o lo hace el

Método para hallar los casos de uso del sistema

- 1. Definir los límites del sistema.
- 2. Encontrar los actores y sus objetivos.
- 3. Identificar los casos de usos.

Encontrar los actores y los objetivos

- Para encontrar los actores, nos preguntamos "¿Cuáles son las personas y programas que interactuarán directamente con nuestro sistema?.
- Para encontrar los objetivos, nos preguntamos "¿Qué quieren hacer estos actores con el sistema?"
- Como ven, el proceso de encontrar actores y objetivos es simultáneo.
- El resultado es una tabla de actor-objetivos.

La tabla actor-objetivos (parte)

Actor	Objetivos
Cliente	Obtener crédito en línea Pagar por transferencia bancaria en línea Obtener historia sobre los últimos créditos y pagos.
Gerente General	Obtener información estadística sobre créditos
Gerente de Cobros	Obtener información sobre los morosos. Autorizar a un cliente a un monto de crédito
Operador de banco	Entrar los datos de cada cliente. Registrar los pagos personales. Obtener crédito para un cliente personal.

La tabla actor-objetivos

Actor	Objetivos
Cliente	Obtener crédito en línea Pagar por transferencia bancaria en línea Obtener historia sobre los últimos créditos y pagos.
Gerente General	Obtener información estadística sobre créditos
	Registrar los pagos personales. Obtener crédito para un cliente personal.

Como ven, contiene los actores y lo que desea hacer cada uno con el sistema (los objetivos).

Algunas preguntas para detectar los actores

- Como hemos visto, la pregunta correcta es "¿Cuáles son las personas y programas que interactuarán directamente con nuestro sistema?.
- Pero, a veces, preguntas más específicas (casos particulares de la primera pregunta) pueden ser útiles también.

Algunas preguntas específicas para detectar los actores

- ¿Quién o qué usa el sistema?
- ¿Hay tareas que se ejecutan cada cierto tiempo? (Si es así, el actor "Tiempo").
- ¿Quién instala el sistema?
- ¿Quién inicia y apaga el sistema?
- ¿Quién mantiene al sistema?
- ¿Qué otros sistemas interactúan con el sistema?
- ¿Quién obtiene información del sistema?
- ¿Quién provee información al sistema?

Método para hallar los casos de uso del sistema

- 1. Definir los límites del sistema.
- 2. Encontrar los actores y sus objetivos.
- 3. Identificar los casos de usos.

Recordemos de que habíamos definido casos de uso

- Un caso de uso es una historia de cómo se usa un sistema para conseguir un objetivo.
- Queda claro que, para cada objetivo, habrá un caso de uso diferente.
- Para hallar los casos de uso, me bastará con hallar los objetivos.

Cada uno de los objetivos de la tabla es un caso de uso

Actor	Objetivos
Cliente	Obtener crédito en línea Pagar por transferencia bancaria en línea Obtener historia sobre los últimos créditos y pagos.
Gerente General	Obtener información estadística sobre créditos
	Registrar los pagos personales. Obtener crédito para un cliente personal.

Cada uno de los objetivos de la tabla actor-objetivos nos dará un caso de uso.

Sólo debemos dar un nombre a cada caso de uso

Objetivos	Casos de uso
Obtener crédito en línea	Otorgar Crédito
Pagar por transferencia bancaria en línea	Pagar por Transferencia
Obtener historia sobre los últimos créditos y pagos.	Obtener Historia

- Recordemos que el nombre del caso de uso debe ser sencillo y suele estar compuesto por verbo y nombre.
- Ya tenemos los casos de uso.

Una excepción a la regla

- Normalmente cada caso de uso sirve a un único objetivo.
- Sin embargo hay casos de uso que sirven a varios objetivos.
- Se trata de los casos de uso que especifican las operaciones de crear, recuperar, actualizar y borrar de elementos de una colección.
- Crear, recuperar, actualizar y borrar empleados.
- Crear, recuperar, actualizar y borrar facturas.
- Son las acciones típicas que hay en un formulario de mantenimiento.

Mail: palasi@palasi.com Web: www.palasi.com

Estas tareas se denominan CRUD

- **C**reate **R**etrieve **U**pdate **D**elete. En español se podían llamar **CRAB** (**C**rear **R**ecuperar **A**ctualizar **B**orrar).
- Cada una de ellas podría ser especificada en un caso de uso diferente.
- Pero por comodidad se especifican en un solo caso de uso para todas (cuyo nombre suele comenzar por "Gestiona").
- Este caso de uso sirve a varios objetivos y es una excepción a la regla general de que cada caso de uso es para un solo objetivo..

¿Cómo saber que lo hicimos bien?

- Los casos de uso deberían especificar una tarea:
  - Realizada por un actor.
  - En un lugar.
  - De una vez.
  - Que añade valor de negocio.
  - Que deja los datos en un estado consistente.
- Esto no es una regla estricta (por ejemplo, pueden haber casos de uso con dos actores), pero si el caso de uso se aleja mucho de esto es una mala señal.
- Esto no puede servir cómo una regla de comprobación.

Por ejemplo

- Los siguientes no son casos de uso:
  - Negociar un contrato.
  - Imprimir un documento.
  - Calcular el IVA.
- Ya que:
  - Negociar un contrato significa una tarea que puede dudar días, con varios actores y diferentes lugares.
  - Imprimir un documento y calcular el IVA no añaden valor al negocio.

Y sin embargo

- A veces es conveniente definir subcasos de uso para tratar tareas que se repiten en varios casos de uso y que no queremos escribir una y otra vez.
- Estos subcasos de uso no tienen porqué estar asociados a un objetivo ni seguir la regla que acabamos de ver.
- Sin embargo, no estudiaremos estos subcasos de uso.

Ejercicios

- Hagan la tabla actor-objetivos del sistema de facturación que se les entregará.
- Identifiquen a partir de ella los casos de uso.

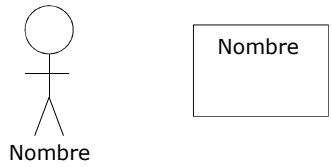
Parte 2: Análisis orientado a objetos con UML

Mail: palasi@palasi.com Web: www.palasi.com

Diagramas de casos de uso

- Son diagramas que nos sirven para representar gráficamente los diferentes actores del sistema juntamente con sus casos de uso.
- Es una forma gráfica de ver los diferentes casos de uso que hay en un sistema.

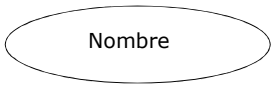
Recordemos: un actor se representa con la siguiente notación



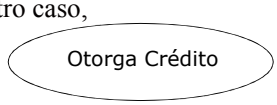
- Se puede utilizar cualquiera de las dos notaciones.
- Es común utilizar la primera notación para actores humanos y la segunda para otros sistemas, pero esto no es un estándar.
- A veces, se utiliza sólo la primera figura y se escribe **«Sistema»** para denotar que el actor no es humano.

Un caso de uso se representa como una elipse

- En el centro de la elipse se escribe el nombre del caso de uso. Así:

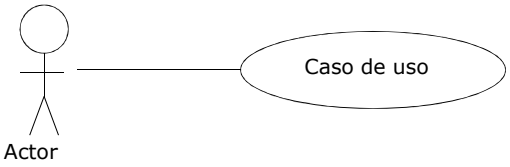


- En nuestro caso,



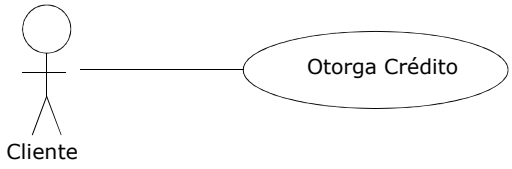
Si un actor está relacionado con un caso de crédito

- Se representa como una línea entre los dos.



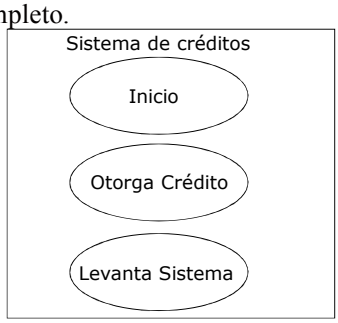
Por ejemplo,

- En nuestro caso de uso "Otorga Crédito".

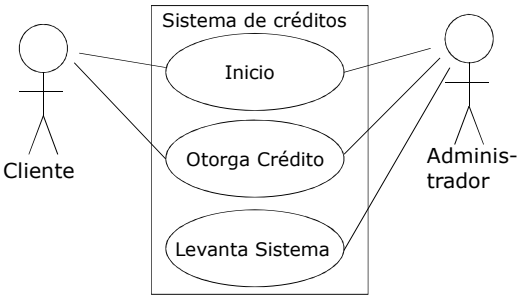


Todos los casos de uso se enmarcan con un rectángulo

- Este rectángulo representa el sistema completo.



Ejemplo: un diagrama de casos de uso



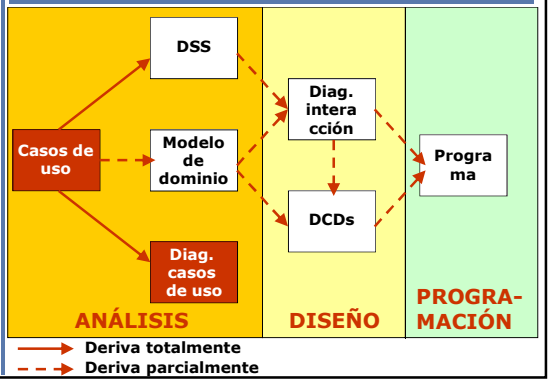
Nota importante

- Un diagrama de casos de uso expresa la misma información que una tabla actor-objetivos pero de forma mucho más visual.
- Los diagramas de casos de uso son útiles para tener un contexto visual del sistema:
  - saber cuáles son los casos de uso principales.
  - conocer el uso que del sistema hacen los actores.
- Sin embargo, lo importante de los casos de uso es **escribirlos**, no hacer diagramas.
- Si pasamos mucho tiempo haciendo diagramas esto es señal de que no hemos enfocado

Como conclusión

- Se recomienda dibujar un sencillo diagrama de casos de uso en conjunción a una tabla actor-objetivos.
- No se recomienda invertir mucho tiempo en la realización y perfeccionamiento de diagramas de casos de uso.

Relación de los diagramas de casos de uso con otros artefactos



Ejercicio

- Dibujen un diagrama de casos de uso del sistema de facturación que se les ha enseñado.

Parte 2: Análisis orientado a objetos con UML

Mail: palasi@palasi.com Web: www.palasi.com

Diagramas de secuencia del sistema (DSS)

- Son diagramas que muestran la interacción entre un actor y el sistema.
- Es decir, las diferentes comunicaciones que se establecen entre actores y sistemas.
- Los diferentes datos que el actor envía al sistema y el sistema devuelve al actor.
- Normalmente representan un escenario de un caso de uso.
- Más frecuentemente es el escenario principal, pero puede ser un escenario alternativo especialmente

Representando a un actor en un DSS

- Se representa con el símbolo de actor (ya sea el hombrecito o el cuadro) y el cuadro.
- Debajo de él, está su nombre subrayado precedido de dos puntos (:)
- Debajo de él, hay una línea vertical discontinua que refleja el paso del tiempo.

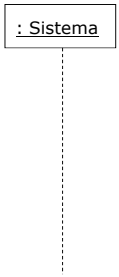
Pregunta: ¿Qué significa que esté subrayado y con los dos puntos delante?

Significa que es una instancia

- Como vimos anteriormente, un subrayado indica que se trata de un objeto, no de una clase.
- En este caso, la "clase" sería Cliente y quien interactúa es una instancia de cliente (un cliente en particular).
- Lo podríamos escribir así. cliente1:Cliente
- Sin embargo, el nombre de la instancia no nos interesa y lo escribimos así. :Cliente, que se lee "instancia de Cliente".

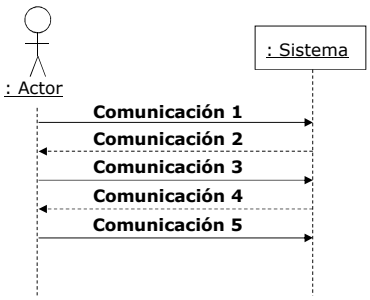
Representando al sistema en un DSS

- Se representa con un cuadro.
- Dentro del cuadro está el nombre del sistema subrayado y precedido de dos puntos (:)
- También se puede utilizar Sistema por simplicidad.



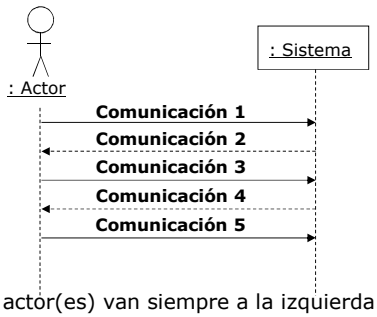
El subrayado expresa que es una instancia (una ejecución) del sistema

Estructura de un DSS sencillo



Un DSS indica las comunicaciones entre un actor y el sistema. Cada comunicación se representa por una flecha.

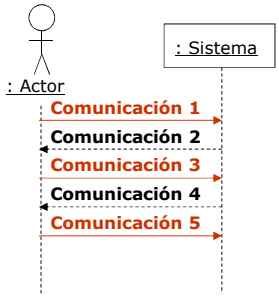
Estructura de un DSS sencillo



El(los) actor(es) van siempre a la izquierda. El tiempo va hacia abajo. Las comunicaciones de arriba se realizan primero.

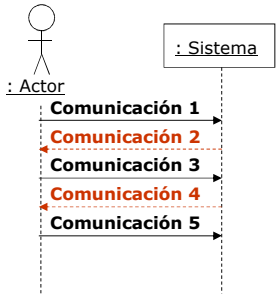
Hay dos tipos de comunicaciones. Primer tipo.

- **Son las peticiones que realiza un actor al sistema.**
- Se llaman "eventos del sistema" generados por los actores.
- Son las flechas continuas que van de izquierda a derecha (en rojo en la figura).



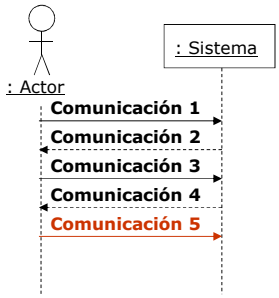
Hay dos tipos de comunicaciones. Segundo tipo.

- **Son las respuestas que realiza el sistema a las peticiones del actor.**
- Cada una está asociada a una petición del actor.
- Suelen retornar datos al actor.
- Son las flechas discontinuas que van de izquierda a derecha (en rojo en la figura).



El segundo tipo es opcional

- **Si la petición del actor no devuelve ningún dato, no hace falta dibujar la flecha discontinua hacia la izquierda**
- Esto se ve en la comunicación 5 del DSS que hay a la derecha.

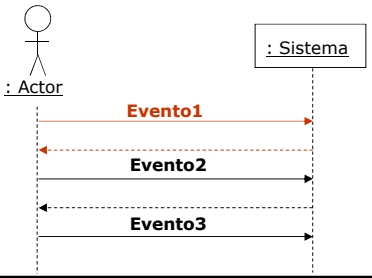


Estos dos tipos de comunicaciones vienen por pares

- El actor le pide una información al sistema y (a veces) el sistema le responde esta información (a veces el sistema no responde).
- A este par (o una sola comunicación si el sistema no responde) se le llama **evento del sistema**

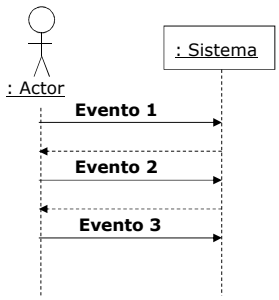
Un DSS no es más que una secuencia de eventos del sistema

- Fíjense que en el evento 3 el sistema no responde nada.

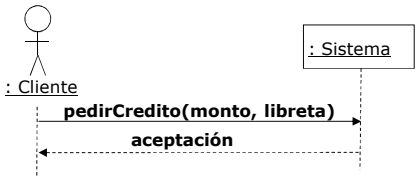


Importante

- Los eventos del sistema deben ser pensados como peticiones (mensajes, llamadas a métodos) al sistema que devuelven un resultado.
- Las líneas discontinuas indican los resultados que devuelven esas peticiones (estos métodos).

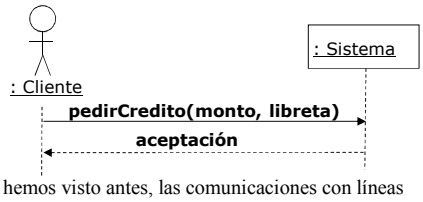


Ejemplo de DSS relacionado con el ejemplo de otorgar crédito



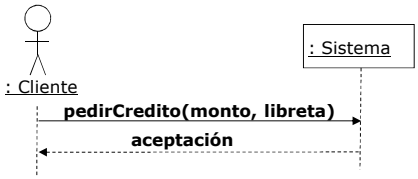
- El cliente pide un crédito y para ello envía al sistema el monto del crédito y la libreta como datos.
- El sistema devuelve si ha aceptado o no el crédito.

Ejemplo de DSS :Notas



- Como hemos visto antes, las comunicaciones con líneas continuas deben ser pensadas como llamadas a métodos de sistema. **Por lo tanto se expresan en una sintaxis de métodos (con parámetros).**
- Las comunicaciones con líneas discontinuas deben ser pensadas como los resultados de esta llamada a método. **Son datos (en este caso, la aceptación).**

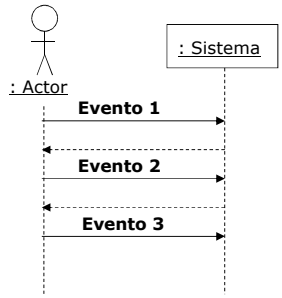
Una idea intuitiva: Veámoslo como llamadas a métodos.



- Cuando diseñamos un DSS, debemos pensar siempre en términos de llamadas a métodos (con parámetros y resultados) del actor al sistema.
- **Los comportamientos que no puedan ser modelados como llamadas a métodos así como los comportamientos internos del sistema no nos interesan.**

Un hecho importante

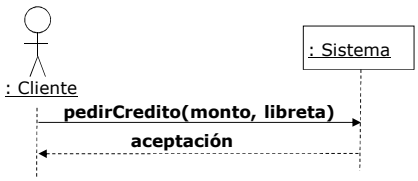
- **Los DSS representan un escenario de un caso de uso.**
- Puede ser el escenario principal (lo más común) o alternativo.
- **Para obtener el DSS, nos basamos en el caso de uso.**



¿Cuántos DSS hay que dibujar?

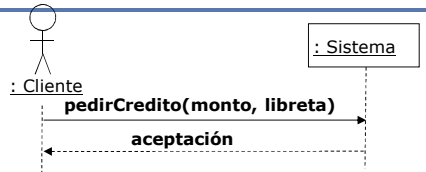
- Normalmente, se dibujan los DSS correspondientes a los escenarios principales de los casos de uso.
- Si hay algún escenario alternativo complejo o importante también se incluye.
- Los DSS se incluyen como un anexo a los casos de uso.

Ejemplo de DSS



- Este DSS se corresponde con el escenario principal del caso de uso.
- Veamos como se deriva esto del escenario principal del caso de uso.

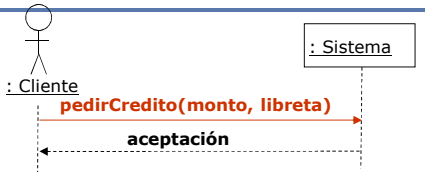
Derivando el DSS del escenario



Escenario principal:

1. El Cliente selecciona la opción de crédito.
2. El Sistema pide el monto del crédito y el número de libreta para depositar.
3. El Cliente entra el monto del crédito y el número de libreta.
4. El Sistema registra el crédito, deposita el monto en la libreta y presenta que el crédito ha sido

Derivando el DSS del escenario



Escenario principal:

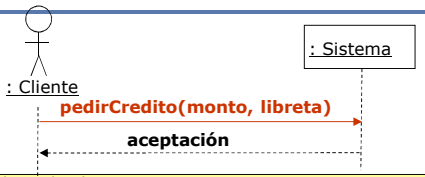
1. El Cliente selecciona la opción de crédito.
2. El Sistema pide el monto del crédito y el número de libreta para depositar.
3. El Cliente entra el monto del crédito y el número

Como vemos, el DSS no es una trascripción automática de un caso de uso.

Reglas para crear un DSS a partir de un caso de uso

- Todas las comunicaciones deben modelarse en forma de "métodos" que ejecuta el actor sobre el sistema junto con sus correspondientes parámetros y resultado.
- No nos interesa en detalles de interfaz gráfica como que el sistema pide unos ciertos datos o que el sistema muestra un cierto mensaje.
- No nos interesa qué hace el sistema en su interior, sólo la comunicación con los actores.
- Lo importante son
  - Las acciones que una actor pide al sistema

La 1ª acción que pedimos al sistema es un crédito

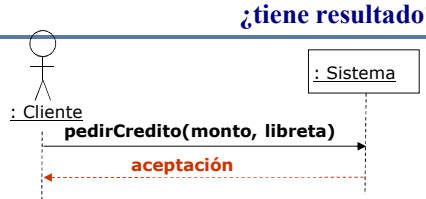


Escenario principal:

1. El Cliente selecciona la opción de crédito.
2. El Sistema pide el monto del crédito y el número de libreta para depositar.
3. El Cliente entra el monto del crédito y el número

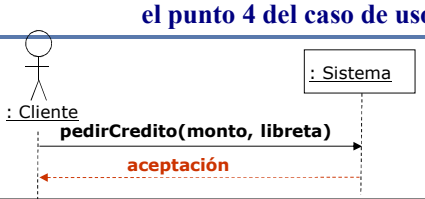
Es por eso que estos tres pasos se resumen en la 1ª flecha. No nos interesan los aspectos de interfaz gráfica.

Esta acción que hemos pedido, ¿tiene resultado?



- Si que lo tiene, devuelve un booleano que indica si el crédito ha sido aceptado.
- Esto es lo que se indica con la segunda flecha.

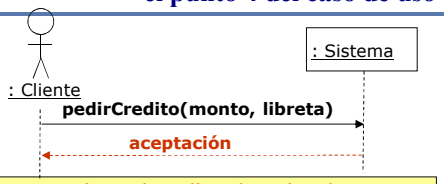
Esta segunda flecha refleja el punto 4 del caso de uso



4. El Sistema registra el crédito, deposita el monto en la libreta y presenta que el crédito ha sido concedido

El registro del crédito y el depósito son operaciones internas del sistema, que aquí no nos interesan.

Esta segunda flecha refleja el punto 4 del caso de uso



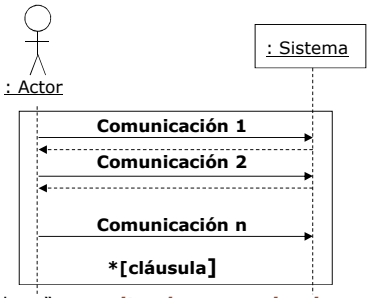
4. El Sistema registra el crédito, deposita el monto en la libreta y presenta que el crédito ha sido concedido

Que se informe al actor que el crédito ha sido concedido nos interesa, pero no el mensaje (que es tema de interfaz de usuario) sino el simple dato booleano.

Veamos ahora otro ejemplo

- Supongamos que queremos modelar con un DSS el escenario principal del caso de uso "Otorga Créditos".
- En éste había una iteración que nos permitía obtener varios créditos a la vez.
- Cómo expresamos la iteración (repetición de tareas en un DSS).

Cómo expresar la iteración en un DSS

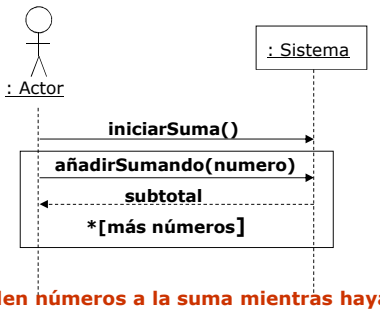


Esto se lee: "*se repiten las comunicaciones de la 1 a la N mientras se cumple la cláusula*"

Cómo expresar la iteración en un DSS

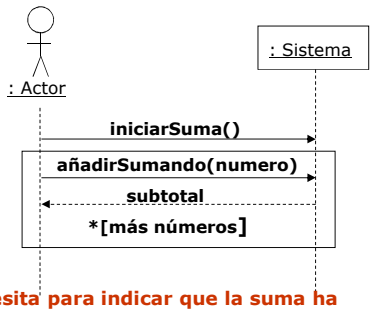
El cuadrado y el asterisco indica que es una iteración. La cláusula entre corchetes indica la condición que se cumple mientras se repite.

Ejemplo: sumar todos los números que desee el usuario



Se añaden números a la suma mientras haya más números.

Fijémonos en el iniciarSuma



Se necesita para indicar que la suma ha empezado y hacer las inicializaciones oportunas (por ejemplo, que el subtotal está a cero).

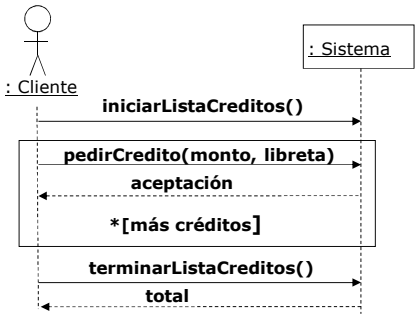
Ahora podemos traducir el caso de uso "Otorga Créditos" a DSS

Nombre: Otorga Créditos.
Actor primario: Cliente.
Escenario principal:
1. El Cliente selecciona la opción de crédito.
2. El Sistema pide el monto del crédito y el número de libreta para depositar.
3. El Cliente entra el monto del crédito y el número de libreta.
4. El Sistema registra el crédito, deposita el monto en la libreta y presenta que el crédito ha sido concedido

El Cliente repite pasos 2-4 hasta que acaba.

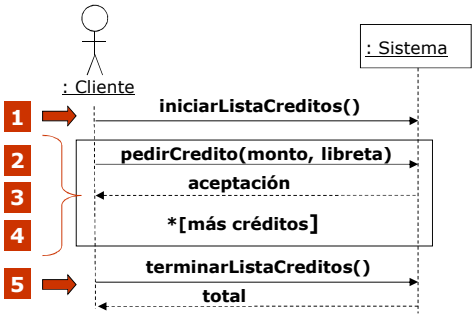
5. El Sistema presenta un total de créditos

Aplicando las reglas que vimos antes se obtiene



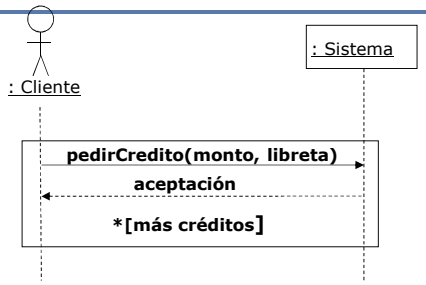
- Se necesita un **iniciarListaCreditos** para poner el subtotal a cero.

Relacionándolo con los casos de uso



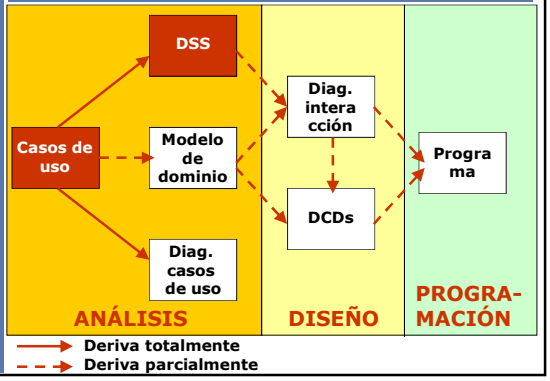
- Se han expresado en rojo los pasos del caso de uso correspondientes a este DSS.

¿Qué pasaría si no necesitáramos un total?



- Cada crédito se trataría por separado y no haría falta inicializar la lista, ni terminarla.

Relación de los DSS con otros artefactos



Ejercicio

- En el ejemplo de facturación, escribir el caso de uso "Ingresar Factura" y dibujar el DSS correspondiente al escenario principal del mismo.

Parte 2: Análisis orientado a objetos con UML

- 2.1. Introducción al análisis orientado a objetos.
- 2.2. Introducción a los casos de uso.
- 2.3. Completando casos de uso.
- 2.4. Precondiciones y postcondiciones.
- 2.5. Un método para detectar los casos de uso.
- 2.6. Diagramas de casos de uso.
- 2.7. Diagramas de secuencia del sistema.
- 2.8. Modelo de dominio: diagrama de clases conceptuales.

La realidad no es algo desordenado e informe

- La realidad tiene estructura. Tiene unas normas. No es al azar.
- Está compuesta por una serie de objetos que
  - tienen características determinadas.
  - se relacionan entre ellos de maneras determinadas.
- A esto le llamamos estructura.
- Por ejemplo:
  - Un auto está compuesto por unas partes específicas.
  - Un auto tiene unas propiedades y acciones.
  - Un auto es un tipo de vehículo.
  - Un auto puede correr sobre una carretera pero una carretera no puede correr sobre un auto.
  - No se puede hacer mayonesa con un auto pero sí que se puede llevar mayonesa dentro de un auto.

La estructura de la realidad nos importa en nuestro programa

- Ejemplo: Programa para un taller de mantenimiento de autos.
- Para programar, será importante conocer que el auto tiene diferentes partes y que estas partes son: motor, carrocería, sistema eléctrico, etc.
- En general, la estructura de la realidad nos importa a la hora de programar.
- De hecho, no nos importa la estructura de toda la realidad sino de la parte que gestiona nuestro programa.

A esto le hemos llamado "dominio"

- Hemos visto que el dominio es la parte de la realidad de la que trata nuestro sistema.
- En un sistema contable, el dominio es la contabilidad. En un sistema de gestión universitaria, el dominio es la Universidad.
- La realidad (y, por tanto, el dominio) tiene una estructura.
- El modelo de dominio es una forma de explorar la estructura del dominio de nuestra aplicación.

Modelo de dominio

- Es la representación gráfica de la estructura del dominio de la aplicación.
- ¿Qué forma usaremos para representarla?
- Hemos dicho que la orientación a objetos organiza los objetos de software de la misma manera que los objetos están organizados en la realidad.
- Por ello, parece una buena elección utilizar los conceptos de orientación a objetos para modelar la realidad.

Modelo de dominio

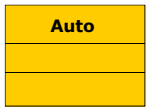
- Es la representación gráfica de la estructura del dominio de la aplicación.
- Para expresarlo, usaremos el concepto de clase conceptual.
- Una clase conceptual (o "clase de análisis") es un conjunto de **objetos del mundo real** que son similares.
- No hay que confundir con las clases de un lenguaje orientado a objeto, que son clases de programación.

Mail: palasi@palasi.com Web: www.palasi.com

Clase conceptual

- Conjunto de **objetos del mundo real** que son similares.

- Se representan con notación de orientación a objetos.

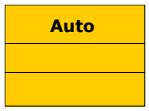


Clases de software

- Construcciones de un lenguaje de programación O-O (a veces, se corresponden con las clases conceptuales).

```
class Auto {
   String marca;
   String modelo;
   int potencia;
   void arrancar(){
   }
   ...
}
```

- Se representan también con notación de orientación a objetos.



Aunque se representan igual, es necesario distinguirlas

- Como el número 0 y la letra O.
- Las clases conceptuales representan la realidad.
- Las clases de software representan el programa.

Clase conceptual

Clase de software

```
class Auto {
   String marca;
   String modelo;
   int potencia;
   void arrancar(){
   }
   ...
}
```

El modelo de dominio sólo tiene clases conceptuales

- Ya que el modelo de dominio trata sobre la realidad, no sobre el programa.

Clase conceptual

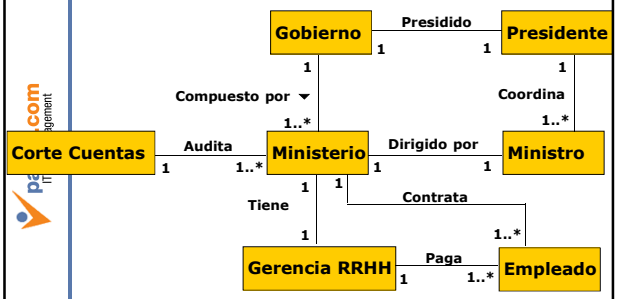
Clase de software

```
class Auto {
   String marca;
   String modelo;
   int potencia;
   void arrancar(){
   }
   ...
}
```

Modelo de dominio

- Es la representación gráfica de la estructura del dominio de la aplicación.
- "Es la representación visual de clases conceptuales en un dominio de interés." Martin Fowler
- Es decir, lo que tendremos será un diagrama de clases conceptuales que modela el dominio de nuestra aplicación.

Modelo de dominio. Diagrama de clases conceptuales.



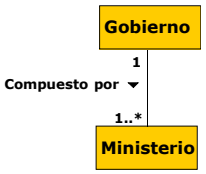
- Compuesto por **clases conceptuales** (cuadrados) y **relaciones** entre ellas (líneas), llamadas **asociaciones**.

Mail: palasi@palasi.com Web: www.palasi.com

Veamos una parte del diagrama de clases conceptuales

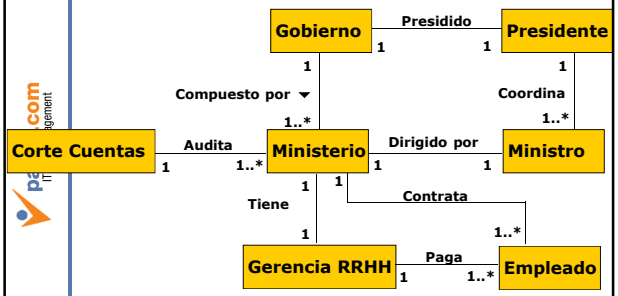
- Gobierno y Ministerio son clases y están relacionadas entre sí por una asociación (relación entre clases).
- La asociación se muestra como una línea que une las dos clases, etiquetada con un nombre. En este caso, "Compuesto por".
- Esto se lee como "el Gobierno está compuesto por

Algunos detalles sobre la asociación



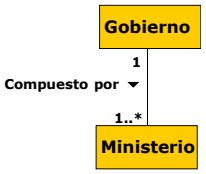
- La asociación se lee en el sentido que indica la flecha triangular.
- Así, se lee "Gobierno está compuesto por Ministerios" y no "Ministerio está compuesto por Gobiernos".
- Si no se pone la flecha se supone que la asociación se lee de arriba a abajo o de izquierda a derecha (aunque esto no es estándar)

Asociaciones sin flecha se leen de izquierda a derecha o de arriba a abajo



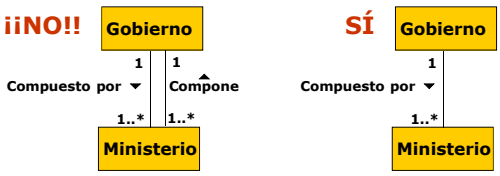
- Como ejercicio, lean todas las asociaciones que aparecen en este diagrama.

Sin embargo, la flecha no indica direccionalidad



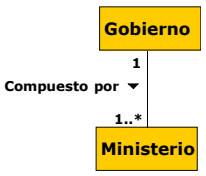
- **Todas las asociaciones son bidireccionales.**
- La asociación es bidireccional, va de Gobierno a Ministerio y de Ministerio a Gobierno.
- La flecha es sólo para saber cómo se lee el nombre, pero no indica direccionalidad de la asociación.

Todas las asociaciones son bidireccionales



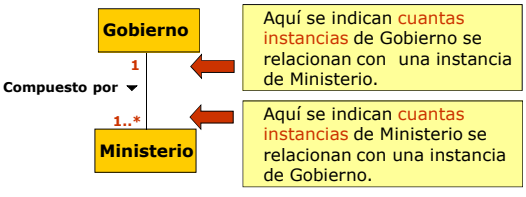
- **La figura de la izquierda es incorrecta.**
- No hace falta dos asociaciones en los dos sentidos, pues la asociación es siempre bidireccional.
- Repetimos que la flecha es sólo para saber cómo se lee el nombre.

Los números que aparecen a cada extremo de la asociación



- Recordemos que cada extremo de la asociación es una clase y, por lo tanto, puede tener varias instancias.
- Así, puede haber varios Ministerios y varios Gobiernos (aunque en un país suele haber sólo uno).
- Los numeritos nos indican cuantos Ministerios están relacionados con cada Gobierno.

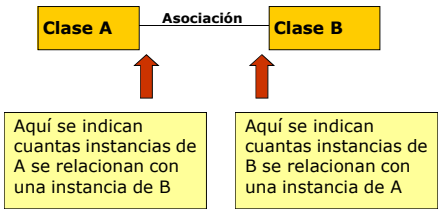
En este ejemplo



1 significa "uno solo"
1..* significa "uno o más" (varios)

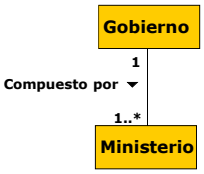
- Un Gobierno se relaciona con **varios** Ministerios.
- Un Ministerio se relaciona con **un solo** Gobierno.

En general



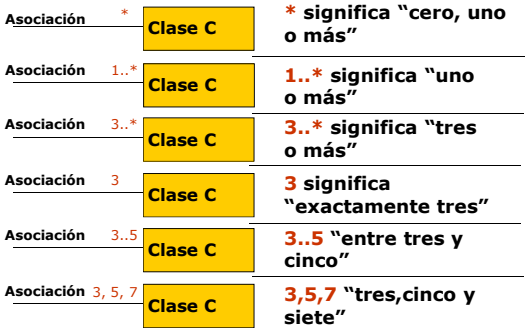
- A esto se le llama "multiplicidad".
- La multiplicidad se indica con los números pequeños a cada extremo de la asociación

A cada extremo de la asociación se le llama "un rol"



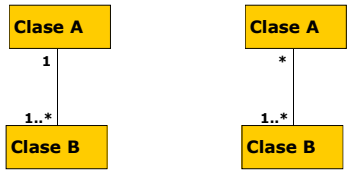
- Cada asociación tiene dos roles.
- Cada rol puede tener una multiplicidad.

Cómo se indica la multiplicidad de los roles



Asociaciones "varios a varios"

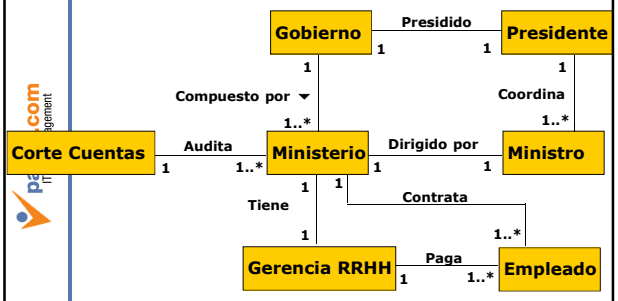
- Se llama así a las asociaciones cuyas dos multiplicidades permiten la posibilidad de varias instancias.



NO, el rol de Gobierno no permite varias instancias. Sólo una.

SI, los dos roles permiten varias instancias.

Resumiendo



- Un modelo de dominio está formado por **clases conceptuales** (cuadrados) y **asociaciones** entre ellas (líneas).

Mail: palasi@palasi.com Web: www.palasi.com

Repetimos

- Todas estas clases conceptuales y asociaciones representan la realidad, no un programa.
- Que una clase o asociación se encuentre en el modelo de dominio de nuestro programa, no quiere decir que acabe en nuestro programa.

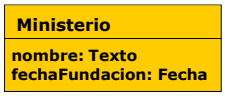
Atributos en clases conceptuales (1)

- Hasta ahora hemos dibujado las clases sin ninguna información adicional.

- Sin embargo, en el modelo de dominio se pueden añadir atributos a las clases. Esto puede representarse así.

Atributos en clases conceptuales (2)

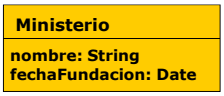
- También pueden indicarse el tipo de los atributos, si se desea.



- Sin embargo, es importante saber que los atributos en el modelo de dominio no son variables, ni sus tipos son tipos de datos.
- Se trata de clases de la vida real, con sus características y el tipo de valores que pueden

Atributos en clases conceptuales (3)

- El tipo de los atributos lo podemos expresar en inglés, por comodidad.



- Sin embargo, es importante repetir que los atributos no son variables, son características de clases conceptuales (de la vida real).
- Los tipos que presentan no son tipos de datos de programación sino los tipos de valores que hay en la vida

¿Y los métodos?

- Las clases conceptuales también pueden tener acciones (del mundo real) que podrían modelarse como métodos.
- Sin embargo, esto no suele ser incluido en el modelo de dominio. Lo único que está incluido en este modelo es:
  - Las clases conceptuales **(lo más importante)**.
  - Las asociaciones.
  - Los atributos.

Lo más importante no son los atributos

- Lo importante del modelo de dominio es encontrar las clases conceptuales.
- Encontrar los atributos es secundario. Por ello:
  - Es frecuente que aparezcan clases sin atributos.
  - Sólo deben indicarse los atributos que sean significativos para el dominio que estamos interesados.
- En caso de duda, es mejor no incluir un atributo: siempre se puede hacer en una iteración siguiente.

Por ejemplo, la clase conceptual Auto

- Puede tener muchos atributos.

- Si queremos modelar un sistema de parqueo, el único atributo que consideraremos es dueño

Ejercicio

- Dibujen un modelo de dominio que refleje un examen.
- Como la palabra "examen" es ambigua, consideramos un examen como el mismo para todos los alumnos ("el examen de Programación", "el examen de Estructuras de Datos") y no "el examen de Fulanito".
- No olviden de incluir las preguntas, la materia, la carrera, los alumnos y el profesor.
- Nota: no modelen la nota todavía.

Parte 2: Análisis orientado a objetos con UML

- 2.1. Introducción al análisis orientado a objetos.
- 2.2. Introducción a los casos de uso.
- 2.3. Completando casos de uso.
- 2.4. Precondiciones y postcondiciones.
- 2.5. Un método para detectar los casos de uso.
- 2.6. Diagramas de casos de uso.
- 2.7. Diagramas de secuencia del sistema.
- 2.8. Modelo de dominio: diagrama de clases conceptuales.

Ya sabemos cómo es el modelo de dominio

- Al menos en sus aspectos más significativos.
- Más adelante lo completaremos con más detalles.
- Ahora queremos saber cómo obtenemos un modelo de dominio.
- Para ello, presentaremos un método de creación.

Método para obtener un modelo de dominio

- 1. Obtener las clases conceptuales.
- 2. Añadir las asociaciones entre las clases.
- 3. Añadir atributos a las clases.
- 4. Refinar las asociaciones.

Mientras explicamos este método

- Lo vamos a aplicar al ejemplo del crédito que habíamos desarrollado desde el apartado anterior.
- De esta manera, no nos limitaremos a la teoría, sino que veremos los resultados a la práctica.

Método para obtener un modelo de dominio

- 1. Obtener las clases conceptuales.
- 2. Añadir las asociaciones entre las clases.
- 3. Añadir atributos a las clases.
- 4. Refinar las asociaciones.

1. Obtener las clases conceptuales

- Este es el paso más importante, ya que las asociaciones y atributos dependen de las clases.
- Es un paso difícil y requiere análisis y experiencia.
- Sin embargo es usual que no quede perfecto a la primera, por lo cual se recomienda un modelo iterativo.

1. Obtener las clases conceptuales

- 1.1. Obtener las clases candidatas.
  - 1.1.1. Usar una lista de categorías para clases conceptuales.
  - 1.1.2. Usar la identificación de nombres.
  - 1.1.3. Usar análisis CRC.
  - 1.1.4. Incluir otras clases que detectemos.
- 1.2. Seleccionar las clases conceptuales a partir de las clases candidatas.
  - 1.2.1. Hacer una lista de las clases candidatas.
  - 1.2.2. Descartar clases sinónimas y redundantes.
  - 1.2.3. Descartar clases candidatas que son atributos.
  - 1.2.4. Descartar clases candidatas que no nos

1. Obtener las clases conceptuales

- 1.1. Obtener las clases candidatas.
  - 1.1.1. Usar una lista de categorías para clases conceptuales.
  - 1.1.2. Usar la identificación de nombres.
  - 1.1.3. Usar análisis CRC.
  - 1.1.4. Incluir otras clases que detectemos.
- 1.2. Seleccionar las clases conceptuales a partir de las clases candidatas.
  - 1.2.1. Hacer una lista de las clases candidatas.
  - 1.2.2. Descartar clases sinónimas y redundantes.
  - 1.2.3. Descartar clases candidatas que son atributos.
  - 1.2.4. Descartar clases candidatas que no nos

1.1. Obtener las clases candidatas

- Se trata de obtener nombres de todo aquello que podría ser una clase.
- En este punto, nos interesa obtener la mayor cantidad de clases candidatas posibles.
- Es saludable realizar este paso con un espíritu de "lluvia de ideas" (*brainstorming*)
- Después, ya analizaremos qué clases nos quedamos o no.

1.1. Obtener las clases candidatas

- Podemos utilizar uno, varios o todos los métodos siguientes:
- 1.1.1. Usar una lista de categorías para clases conceptuales.
- 1.1.2. Usar la identificación de nombres.
- 1.1.3. Usar análisis CRC.
- 1.1.4. Incluir otras clases que detectemos.

1. Obtener las clases conceptuales

- 1.1. Obtener las clases candidatas.
  - 1.1.1. Usar una lista de categorías para clases conceptuales.
  - 1.1.2. Usar la identificación de nombres.
  - 1.1.3. Usar análisis CRC.
  - 1.1.4. Incluir otras clases que detectemos.
- 1.2. Seleccionar las clases conceptuales a partir de las clases candidatas.
  - 1.2.1. Hacer una lista de las clases candidatas.
  - 1.2.2. Descartar clases sinónimas y redundantes.
  - 1.2.3. Descartar clases candidatas que son atributos.
  - 1.2.4. Descartar clases candidatas que no nos

1.1.1. Usar una lista de categorías para clases conceptuales

- Se utiliza una lista como la que explicamos a continuación.
- Esta lista contiene categorías de clases conceptuales.
- Se repasa categoría por categoría y se ve qué posibles "cosas" de nuestro dominio se pueden incluir en estas categorías.
- Así se obtiene una lista de clases candidatas.

Lista de categorías para clases conceptuales (1)

Categoría	Ejemplos
Objetos físicos	Auto Calculadora
Especificaciones, diseños o descripciones de cosas.	DescripcionProducto
Lugares	Tienda
Transacciones	Venta Pago Reserva
Líneas de detalle de transacciones	LineaDetalleCompra LineaDetalleFactura
Cargos o roles de gente	Profesor Vendedor

Lista de categorías para clases conceptuales (2)

Categoría	Ejemplos
Cosas que contienen otras	Tienda Auto
Cosas contenidas en otras	Motor Pasajero
Otros sistemas externos	ServicioAutorizaciónTarjetas
Nombres abstractos de conceptos	Configuración
Organizaciones	Departamento GerenciaRecursosHumanos
Acontecimientos	Venta Pago Reunión

Lista de categorías para clases conceptuales (3)

Categoría	Ejemplos
Reglas y políticas	PoliticaDevolución
Catálogos	CatálogoProductos
Documentos	LibroIVA

Se repasa la lista y se buscan las "cosas" del dominio que corresponden a cada una de estas categorías. Estas serán nuestras clases candidatas.

Se busca la mayor cantidad posible.

Después ya tendremos tiempo de escoger las adecuadas.

Apliquémoslo a nuestro ejemplo del crédito (1)

Categoría	Ejemplos
Objetos físicos	Cliente
Especificaciones, diseños o descripciones de cosas.	
Lugares	Banco
Transacciones	Crédito
Líneas de detalle de transacciones	
Cargos o roles de gente	Cliente

Apliquémoslo a nuestro ejemplo del crédito (2)

Categoría	Ejemplos
Cosas que contienen otras	Banco
Cosas contenidas en otras	Cuenta
Otros sistemas externos	
Nombres abstractos de conceptos	
Organizaciones	Banco
Acontecimientos	AutorizaciónCréditos

Apliquémoslo a nuestro ejemplo del crédito (3)

Categoría	Ejemplos
Reglas y políticas	PolíticaCreditos
Documentos	Libreta

Clases candidatas resultantes:

- Cliente
- Banco
- Crédito
- Cuenta
- AutorizaciónCréditos
- PolíticaCreditos
- Libreta

1. Obtener las clases conceptuales

- 1.1. Obtener las clases candidatas.
  - 1.1.1. Usar una lista de categorías para clases conceptuales.
  - 1.1.2. Usar la identificación de nombres.
  - 1.1.3. Usar análisis CRC.
  - 1.1.4. Incluir otras clases que detectemos.
- 1.2. Seleccionar las clases conceptuales a partir de las clases candidatas.
  - 1.2.1. Hacer una lista de las clases candidatas.
  - 1.2.2. Descartar clases sinónimas y redundantes.
  - 1.2.3. Descartar clases candidatas que son atributos.
  - 1.2.4. Descartar clases candidatas que no nos

1.1.2. Identificación de nombres

- Se comienza recogiendo **la documentación**, es decir, la información en forma de texto que tengamos hasta ahora.
  - Los casos de uso.
  - Otros artefactos que no hemos visto: Especificación Suplementaria, Glosario, la Visión, el Documento de Arquitectura del Software.
  - Cualquier documento que tengamos sobre el dominio de la aplicación.

1.1.2. Identificación de nombres

- **En toda esta información se intentan detectar nombres.**
- Se entiende nombre en el sentido amplio ("sintagma nominal"). Por ejemplo: "numero de tarjeta de crédito" es un nombre.
- **Cada nombre es una clase candidata.**
- **Nota: no se incluyen los nombres de los actores ni del sistema.**

Sin embargo

- Este método no es automático.
- El lenguaje natural es un lenguaje ambiguo.
- A veces, los nombres tienen varios significados.
- A veces, hay verbos que podrían ser nombres.

Para evitar esto último, se proponen dos pasos

- 1. Se localizan todos los nombres y se añaden en nuestra lista como clases candidatas.
- 2. Se vuelven a escribir las frases, intentando transformar los verbos (excepto los de entrada/salida) en nombres y convirtiendo esto en clases candidatas.

Ejemplo. 1er. Paso

- Examinamos el escenario principal del caso de uso "Otorga Crédito". Nombres en rojo.

Nombre: Otorga Crédito.
Actor primario: Cliente.
Escenario principal:
1. El **Cliente** selecciona la opción de **crédito**.
2. El Sistema pide el **monto del crédito** y el número de **libreta** para depositar.
3. El Cliente entra el monto del crédito y el **número de libreta**.
4. El Sistema registra el crédito, deposita el monto en la libreta y presenta que el crédito ha sido

Ejemplo. 2o. Paso

Nombre: Otorga Crédito.
Actor primario: Cliente.
Escenario principal:
1. El Cliente selecciona la opción de crédito.
2. El Sistema pide el monto del crédito y el número de libreta para **depositar**.
3. El Cliente entra el monto del crédito y el número de libreta.
4. El Sistema **registra** el crédito, deposita el monto en la libreta y presenta que el crédito ha sido **concedido**.

No consideramos los verbos que tratan con la entrada/salida: selecciona, pide, entra.

Ejemplo. 2o. Paso

2. El Sistema pide el monto del crédito y el número de libreta para **depositar**.

Deposita ➡ Realiza un depósito

4. El Sistema **registra** el crédito, deposita el monto en la libreta y presenta que el crédito ha sido **concedido**.

Registra el crédito ➡ Realiza registro del crédito
Concedido ➡ Obtenido una concesión del crédito

Como conclusión del proceso de identificación de nombres

- Tenemos las siguientes clases candidatas:
  - Cliente.
  - Crédito.
  - Monto del crédito.
  - Libreta.
  - Número de libreta.
  - Depósito.
  - Registro del crédito.
  - Concesión del crédito.

1. Obtener las clases conceptuales

- 1.1. Obtener las clases candidatas.
  - 1.1.1. Usar una lista de categorías para clases conceptuales.
  - 1.1.2. Usar la identificación de nombres.
  - 1.1.3. Usar análisis CRC.
  - 1.1.4. Incluir otras clases que detectemos.
- 1.2. Seleccionar las clases conceptuales a partir de las clases candidatas.
  - 1.2.1. Hacer una lista de las clases candidatas.
  - 1.2.2. Descartar clases sinónimas y redundantes.
  - 1.2.3. Descartar clases candidatas que son atributos.
  - 1.2.4. Descartar clases candidatas que no nos

Análisis CRC

- CRC (acrónimo de Clases, Responsabilidades y Colaboradores) es una técnica de "lluvia de ideas" ("brainstorming") para encontrar clases candidatas.
- Como valor añadido, también nos permite dar una idea de cuáles serán los métodos de cada clase (lo cual no trataremos hasta el diseño).
- Es una técnica que nos permite involucrar a un equipo de gente en la obtención de clases.

El análisis CRC se basa en tarjetas

- Aunque es más práctico hacerlo con notas adhesivas tipo "Post-it".
- Cada tarjeta o nota adhesiva se divide en tres apartados de la siguiente forma:

Clase:	
Responsabilidades:	**Colaboradores:**

Cada tarjeta o nota adhesiva representa una clase diferente

- En el apartado superior se indica el nombre de la clase.
- En el apartado inferior izquierda, las responsabilidades de la clase (aquello de la cual es responsable).
- En el apartado inferior derecha, los colaboradores de la clase (las clases que colaboran con esta para realizar cualquier funcionalidad del sistema

Clase:	
Responsabilidades:	**Colaboradores:**

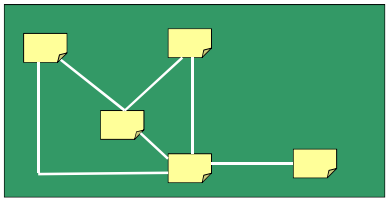
Un ejemplo: la clase Libreta

- La clase Libreta es responsable de permitir ingresar Depósito y de mantener el saldo de la Libreta.
- La clase Libreta es ayudada por la clase Banco para operar.

Clase: Libreta	
Responsabilidades:	**Colaboradores:**
IngresarDeposito MantenerSaldo	Banco

El apartado de colaboradores es una manera de indicar las relaciones

- Las relaciones entre clases.
- Una manera que nos parece mejor es pegar las notas sobre una pizarra y dibujar líneas entre ellas.



CRC es una técnica de "lluvia de ideas" (1)

- Una "lluvia de ideas" ("brainstorming") es una reunión de diversa gente cuyo objetivo es obtener la mayor cantidad de ideas de un tema (en nuestro caso, las ideas serán las clases candidatas).
- El objetivo es evitar la censura y la autocensura:
  - Censura: "Eso que has dicho no sirve"
  - Autocensura: "No propondré esta idea pues los otros pueden considerarme tonto".
- Por culpa de la censura y la autocensura muchas ideas valiosas nunca salen a la luz.

Mail: palasi@palasi.com Web: www.palasi.com

CRC es una técnica de "lluvia de ideas" (2)

- Una "lluvia de ideas" consiste en una reunión de gente dirigida por un facilitador.
- En una "lluvia de ideas" se estimula a que los participantes digan todas las ideas que se les ocurran, aunque sean ridículas o imposibles.
- El facilitador va anotando las ideas sin hacer comentarios.
- Todas las ideas son buenas ideas. Las ideas no son analizadas, criticadas, ni comentadas, sólo se anotan.

Cómo realizar una sesión CRC. Preparación de la sesión.

- 1. Se deciden los participantes a la sesión. Analistas, programadores y gente que conozca el dominio de nuestro programa.
- 2. Se reúnen los participantes y facilitador en un salón tranquilo con una pizarra.
- 3. Se reparten notas adhesivas (o tarjetas) a todos los participantes divididas según el esquema CRC.
- 4. El facilitador explica que todas las ideas son buenas y que no se deben debatir ni comentar las ideas.

Cómo realizar una sesión CRC. Ejecución de la sesión.

- 5. El facilitador pide a los participantes que nombren las "cosas" que hay en el dominio del programa (cliente, producto).
- 6. Cada cosa se escribe en una nota adhesiva que se pega a la pizarra.
- 7. El facilitador pide a los participantes que decidan las responsabilidades que tienen esas "cosas". Esto se escribe en el apartado de responsabilidades.
- 8. El facilitador pide a los participantes que busquen clases que puedan trabajar juntas. Esto se representa como líneas entre las notas (o bien en el

Cómo resultado de la sesión

- 1. Se tiene una lista de clases candidatas (como con las otras técnicas).
- 2. Se tiene información sobre los posibles métodos y relaciones de las clases. Esto no lo utilizaremos por ahora, pero nos será muy útil para más adelante.

Nota importante

- Hemos visto tres técnicas para identificar clases candidatas:
  - 1.1.1. Usar una lista de categorías para clases conceptuales.
  - 1.1.2. Usar la identificación de nombres.
  - 1.1.3. Usar análisis CRC.
- Estas técnicas no son obligatorias. Se puede aplicar:
  - Sólo una.
  - Dos de ellas.
  - Las tres.
- El objetivo es encontrar clases candidatas y las técnicas sólo son un instrumento, no un fin en sí

Ejercicio

- Obtengan las clases candidatas del sistema que producía cotizaciones de computadora y cuyo caso de uso hemos especificado.

Mail: palasi@palasi.com Web: www.palasi.com

1. Obtener las clases conceptuales

- 1.1. Obtener las clases candidatas.
  - 1.1.1. Usar una lista de categorías para clases conceptuales.
  - 1.1.2. Usar la identificación de nombres.
  - 1.1.3. Usar análisis CRC.
  - 1.1.4. Incluir otras clases que detectemos.
- 1.2. Seleccionar las clases conceptuales a partir de las clases candidatas.
  - 1.2.1. Hacer una lista de las clases candidatas.
  - 1.2.2. Descartar clases sinónimas y redundantes.
  - 1.2.3. Descartar clases candidatas que son atributos.
  - 1.2.4. Descartar clases candidatas que no nos

1.1.4. Incluir otras clases que detectemos

- A veces hay otras clases candidatas que hemos detectado de forma diferente a las técnicas que acabamos de ver.
- Las podemos detectar por nuestro conocimiento del dominio, por la documentación, porque se nos ocurren o por cualquier otra razón.
- Estas clases que hemos detectado también las incluimos como clases candidatas.

1. Obtener las clases conceptuales

- 1.1. Obtener las clases candidatas.
  - 1.1.1. Usar una lista de categorías para clases conceptuales.
  - 1.1.2. Usar la identificación de nombres.
  - 1.1.3. Usar análisis CRC.
  - 1.1.4. Incluir otras clases que detectemos.
- 1.2. Seleccionar las clases conceptuales a partir de las clases candidatas.
  - 1.2.1. Hacer una lista de las clases candidatas.
  - 1.2.2. Descartar clases sinónimas y redundantes.
  - 1.2.3. Descartar clases candidatas que son atributos.
  - 1.2.4. Descartar clases candidatas que no nos

Como resultado de las técnicas anteriores

- Acabaremos con una lista de clases candidatas.
- Estas clases no acabaran todas siendo clases conceptuales.
- De hecho, las clases candidatas pueden ser:
  - Excluidas del modelo de dominio (como mínimo en la presente iteración).
  - Incluidas en el modelo de dominio como clases conceptuales.
  - Incluidas en el modelo de dominio como atributos de clases.

Método para obtener las clases conceptuales a partir de las candidatas

- 1.2.1. Hacer una lista de las clases candidatas obtenidas en el paso anterior.
- 1.2.2. Descartar clases sinónimas y redundantes.
- 1.2.3. Descartar clases candidatas que son atributos.
- 1.2.4. Considerar clases que pueden modelarse como métodos.
- 1.2.5. Descartar clases candidatas que no nos interesan (como mínimo, en esta iteración).
- El resultado será la primera lista de clases conceptuales (esto se irá refinando en posteriores

1. Obtener las clases conceptuales

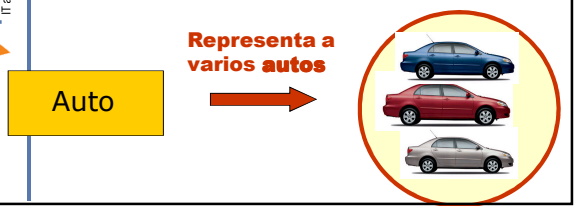
- 1.1. Obtener las clases candidatas.
  - 1.1.1. Usar una lista de categorías para clases conceptuales.
  - 1.1.2. Usar la identificación de nombres.
  - 1.1.3. Usar análisis CRC.
  - 1.1.4. Incluir otras clases que detectemos.
- 1.2. Seleccionar las clases conceptuales a partir de las clases candidatas.
  - 1.2.1. Hacer una lista de las clases candidatas.
  - 1.2.2. Descartar clases sinónimas y redundantes.
  - 1.2.3. Descartar clases candidatas que son atributos.
  - 1.2.4. Descartar clases candidatas que no nos

12.1. Hacer una lista de las clases candidatas obtenidas

- Se combinan todas las listas obtenidas por las técnicas aplicadas (categorías, nombres y/o CRC) en una sola lista de clases candidatas.
- Esta lista será la que servirá como punto de partida de las clases diferentes.

Recordemos: el nombre de la clase siempre va en singular

- Aunque realmente representa un conjunto plural de cosas.
- Así, tenemos la clase "Auto" que representa un conjunto de autos.



En nuestro caso, tenemos la siguiente lista

- Eliminamos clases repetidas.
- Fíjense que todos los nombres de las clases son en singular.

•Cliente	•Libreta
•Banco	•Monto del crédito
•Crédito	•Número de libreta
•Cuenta	•Depósito
•AutorizaciónCréditos	•Registro del crédito
•PolíticaCreditos	•Concesión crédito

1. Obtener las clases conceptuales

- 1.1. Obtener las clases candidatas.
  - 1.1.1. Usar una lista de categorías para clases conceptuales.
  - 1.1.2. Usar la identificación de nombres.
  - 1.1.3. Usar análisis CRC.
  - 1.1.4. Incluir otras clases que detectemos.
- 1.2. Seleccionar las clases conceptuales a partir de las clases candidatas.
  - 1.2.1. Hacer una lista de las clases candidatas.
  - 1.2.2. Descartar clases sinónimas y redundantes.
  - 1.2.3. Descartar clases candidatas que son atributos.
  - 1.2.4. Descartar clases candidatas que no nos

1.2.2. Descartar clases sinónimas y redundantes

•Cliente	•Libreta
•Banco	•Monto del crédito
•Crédito	•Número de libreta
•Cuenta	•Depósito
•AutorizaciónCréditos	•Registro del crédito
•PolíticaCreditos	•Concesión crédito

- las que son superfluas.
- Por ejemplo, Libreta y Cuenta hacen referencia al mismo concepto: una cuenta bancaria. Eliminamos Libreta.
- Por ejemplo, AutorizaciónCréditos y Concesión crédito nos indican las dos si el crédito ha sido autorizado.

Descartando clases sinónimas, nos queda la lista

•Cliente	•Monto del crédito
•Banco	•Número de libreta
•Crédito	•Depósito
•Cuenta	•Registro del crédito
•PolíticaCreditos	•Concesión crédito

- Veamos las clases superfluas. Registro del crédito es superflua, pues la información del crédito se registra en la clase Crédito.
- También desaparecerá de la lista.

1. Obtener las clases conceptuales

- 1.1. Obtener las clases candidatas.
  - 1.1.1. Usar una lista de categorías para clases conceptuales.
  - 1.1.2. Usar la identificación de nombres.
  - 1.1.3. Usar análisis CRC.
  - 1.1.4. Incluir otras clases que detectemos.
- 1.2. Seleccionar las clases conceptuales a partir de las clases candidatas.
  - 1.2.1. Hacer una lista de las clases candidatas.
  - 1.2.2. Descartar clases sinónimas y redundantes.
  - 1.2.3. Descartar clases candidatas que son atributos.
  - 1.2.4. Descartar clases candidatas que no nos

1.2.3. Descartar clases candidatas que son atributos

- Las clases candidatas pueden ser:
  - Clases conceptuales.
  - Atributos que no son clase.
- (Por supuesto, las clases pueden ser atributos de otras clases, pero esto ahora no nos importa)
- ¿Cómo distinguimos entre las dos?
- Ofrecemos un método para saber si una clase candidata es atributo o no.

Regla de oro para diferenciar entre clases y atributos no clase

- A. Son atributos no clase si son de un "tipo de datos" primitivo: número, texto, fecha, hora, sí/no, color…
- B. Son atributos no clase si intentamos asociarle propiedades (atributos) y no se puede.
- Estos dos criterios son equivalentes.

Ejemplo. Apliquemos el criterio A

- Supongamos las siguientes clases candidatas de un programa universitario de realización de examen:
  - Examen.
  - Nota (del examen).
  - Hora (del examen).
  - Aula.
- Aplicando el criterio A, vemos que "nota" y "hora" son de tipos primitivos (número y hora, respectivamente). Los otros no.
- Por lo tanto, "nota" y "hora" son atributos no clase. Los otros son clases.

Ejemplo. Apliquemos el criterio B

- Apliquemos el criterio B.
- A Examen se le pueden asociar propiedades: nota, hora, materia, dificultad, aula, etc.
- A Nota no se le pueden asociar propiedades: es un número y nada más.
- Lo mismo pasa con Hora: no se le pueden asociar propiedades.
- En cambio a Aula sí que se le pueden asociar propiedades: número, ubicación.
- Nota y Hora son atributos no clase y Examen y Aula son clases. Es lo mismo que nos ha dado el

Apliquemos el criterio A a la lista de clases

•**Cliente**	•**Monto del crédito**
•**Banco**	•**Número de libreta**
•**Crédito**	•**Depósito**
•**Cuenta**	•**Concesión crédito**
•**PolíticaCreditos**	

- Serían atributos no clase: monto del crédito (número), número de libreta (texto) y concesión del crédito (sí/no).
- Todo lo otro serían clases.

Mail: palasi@palasi.com Web: www.palasi.com

Apliquemos el criterio B a la lista de clases

•Cliente	•Monto del crédito
•Banco	•Número de libreta
•Crédito	•Depósito
•Cuenta	•Concesión crédito
•PolíticaCreditos	

- Serían atributos no clase: monto del crédito, número de libreta y concesión del crédito. A ellos no se les puede añadir propiedades.
- A los otros sí: Cliente (nombre, teléfono), Banco (nombre, dirección), Crédito (monto, fecha que se pidió), Cuenta (número, saldo), PolíticaCréditos(fecha de validez), Depósito (monto, fecha).

Los dos criterios deben dar el mismo resultado

•Cliente	•Monto del crédito
•Banco	•Número de libreta
•Crédito	•Concesión crédito
•Cuenta	
•PolíticaCreditos	
•Depósito	

CLASES ATRIBUTOS NO CLASE

- ¿De qué clases serán los atributos de la derecha?
- "Número de libreta" será de "Cuenta".
- "Monto del crédito" y "Concesión del crédito" de "Crédito".

Por ahora tenemos las siguientes clases

- El atributo "numero de libreta", ahora se llama "numero" y está en "Cuenta".
- El atributo "monto del crédito", ahora se llama "monto" y está en "Crédito".
- El atributo "concesión del crédito", ahora se llama "autorizado" y está en "Crédito".

1. Obtener las clases conceptuales

- 1.1. Obtener las clases candidatas.
  - 1.1.1. Usar una lista de categorías para clases conceptuales.
  - 1.1.2. Usar la identificación de nombres.
  - 1.1.3. Usar análisis CRC.
  - 1.1.4. Incluir otras clases que detectemos.
- 1.2. Seleccionar las clases conceptuales a partir de las clases candidatas.
  - 1.2.1. Hacer una lista de las clases candidatas.
  - 1.2.2. Descartar clases sinónimas y redundantes.
  - 1.2.3. Descartar clases candidatas que son atributos.
  - 1.2.4. Descartar clases candidatas que no nos

1.2.4. Descartar clases candidatas que no nos interesan

- A veces, hay clases candidatas que han aparecido como resultado del análisis, pero que, a pesar de ser parte del dominio, no nos interesan para la tarea que queremos implementar.
- Por ejemplo, supongamos que en el análisis anterior nos hubiera aparecido la clase "Superintendencia del Sistema Financiero".
- Como no nos interesa la regulación de las entidades financieras en este programa, lo podríamos descartar.

En nuestro caso

- Todas estas clases parecen pertinentes a la hora de otorgar un crédito, por lo que las vamos a dejar.
- Siempre tenemos tiempo de cambiar esta lista, pues usaremos un modelo iterativo.

Mail: palasi@palasi.com Web: www.palasi.com

Con esto, hemos acabado el método para obtener clases conceptuales

- 1.1. Obtener las clases candidatas.
  - 1.1.1. Usar una lista de categorías para clases conceptuales.
  - 1.1.2. Usar la identificación de nombres.
  - 1.1.3. Usar análisis CRC.
  - 1.1.4. Incluir otras clases que detectemos.
- 1.2. Seleccionar las clases conceptuales a partir de las

Este era el paso más importante.
Lo que nos queda no es tan importante

  - 1.2.2. Descartar clases sinónimas y redundantes.
  - 1.2.3. Descartar clases candidatas que son atributos.

Ejercicio

- Obtengan las clases conceptuales del sistema que produce cotizaciones de computadora. Básense en las clases candidatas que hemos obtenido anteriormente.

Método para obtener un modelo de dominio

- 1. Obtener las clases conceptuales.
- 2. Añadir las asociaciones entre las clases.
- 3. Añadir atributos a las clases.
- 4. Refinar las asociaciones.

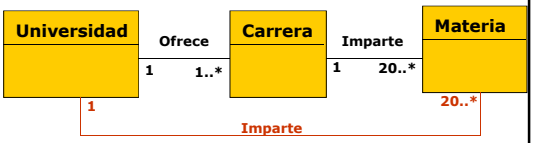
Añadir asociaciones es DEMASIADO fácil

- Si se escogen dos clases conceptuales al azar, es muy probable que hallemos una relación entre ellas que nos permita una asociación.

"Pez" y "Bicicleta" no tienen nada que ver, pero se puede definir una asociación entre ellos.

Además, hay asociaciones que pueden darse de manera transitiva

- Es decir, asociaciones que se derivan de otras.



Estas asociaciones transitivas no aportan nada nuevo. Sólo hacen que complicar nuestro modelo de dominio.

Si suponemos que entre cada dos clases puede haber una asociación

- Si hay N clases conceptuales, el número de asociaciones posibles es de N*(N-1).
- Este número es muy grande y crece exponencialmente.

Número de clases	Número de asociaciones
5	20
10	90
20	380
50	2450

Como consecuencia, si añadimos las asociaciones sin ningún orden

- Es posible que acabemos con un modelo de dominio demasiado complejo, que nos dificulte el análisis y diseño en vez de facilitárnoslo.
- Además, el objetivo del modelo de dominio es identificar clases. Las asociaciones son algo secundario.
- Por ello, deberemos tener algún criterio para limitar el número de asociaciones.

Método para añadir asociaciones al modelo de dominio

- 1. Incluir asociaciones.
  - 1.1. Incluir las asociaciones derivadas de la lista de asociaciones comunes.
  - 1.2. Incluir las asociaciones importantes para nuestro dominio.
  - 1.3. Determinar si entre dos clases hay diferentes asociaciones.
- 2. Descartar asociaciones.
  - 2.1. Descartar las asociaciones cuyo conocimiento no debe guardarse.
  - 2.2. Descartar las asociaciones "transitivas" (que pueden derivarse de otras)

Método para añadir asociaciones al modelo de dominio

- 1. Incluir asociaciones.
  - 1.1. Incluir las asociaciones derivadas de la lista de asociaciones comunes.
  - 1.2. Incluir las asociaciones importantes para nuestro dominio.
  - 1.3. Determinar si entre dos clases hay diferentes asociaciones.
- 2. Descartar asociaciones.
  - 2.1. Descartar las asociaciones cuyo conocimiento no debe guardarse.
  - 2.2. Descartar las asociaciones "transitivas" (que pueden derivarse de otras)

La lista de asociaciones comunes (1)

Categoría	Ejemplos
A es una propiedad de B	Aula-Examen Profesor-Materia
A es una parte de B	Motor-Auto Departamento-Universidad
A está contenido en B	Pasajero-Auto Producto-Catalogo
A es un miembro de B	Profesor-Universidad
A es una línea de detalle de B	LíneaDetalleFactura-Factura
A es una descripción de B	DescripciónVuelo-Vuelo

- Las tres primeras deberían incluirse "siempre" en el modelo de dominio

La lista de asociaciones comunes (2)

Categoría	Ejemplos
A usa o gestiona B	Vendedor-Caja Conductor-Auto
A comunica con B	Cliente-Vendedor
A es contiguo a B	LineaDetalleFactura-LineaDetalleFactura
A posee B	Dueño-Auto
A está relacionado con una transacción B	Cliente-Pago Pago-Venta
A es un evento relacionado con B	Venta-Cliente

En nuestro ejemplo del crédito (1)

Categoría	Ejemplos
A es una propiedad de B	Cliente-Cuenta Cuenta-Depósito Banco-Cuenta Cliente-Depósito
A es una parte de B	Cuenta-Banco
A está contenido en B	Depósito-Cuenta Crédito-Cuenta
A es un miembro de B	Cliente-Banco
A es una línea de detalle de B	
A es una descripción de B	

En nuestro ejemplo del crédito (2)

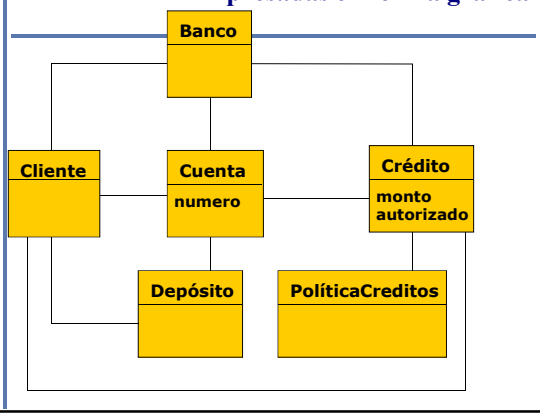
Categoría	Ejemplos
A usa o gestiona B	Cliente-Banco Banco-Crédito Banco-Cuenta Crédito-PolíticaCreditos
A comunica con B	Cliente-Banco
A es contiguo a B	
A posee B	Cliente-Cuenta
A está relacionado con una transacción B	Cliente-Crédito Cliente-Depósito
A es un evento relacionado con B	

Recopilando, tenemos las siguientes asociaciones

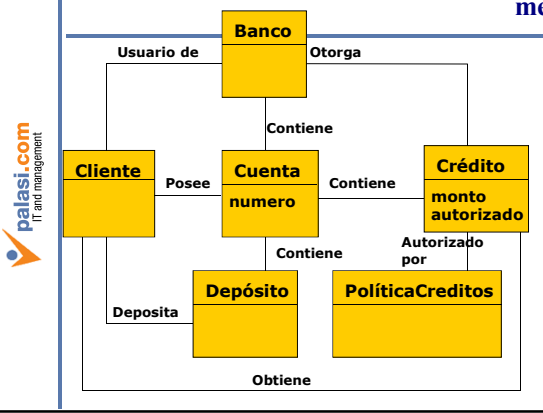
- Descartamos las repetidas. Recuerden además de que las asociaciones son bidireccionales y que es lo mismo la asociación "Cliente-Banco" que "Banco-Cliente".

- Cliente-Cuenta
- Cuenta-Depósito
- Banco-Cuenta
- Cliente-Depósito
- Crédito-Cuenta
- Cliente-Banco
- Banco-Crédito
- Crédito-PolíticaCreditos
- Cliente-Crédito

Expresadas en forma gráfica



Le ponemos nombre para identificarlas mejor



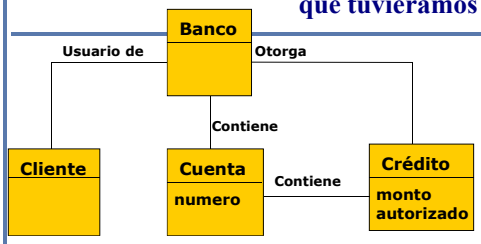
Método para añadir asociaciones al modelo de dominio

- 1. Incluir asociaciones.
  - 1.1. Incluir las asociaciones derivadas de la lista de asociaciones comunes.
  - 1.2. Incluir las asociaciones importantes para nuestro dominio.
  - 1.3. Determinar si entre dos clases hay diferentes asociaciones.
- 2. Descartar asociaciones.
  - 2.1. Descartar las asociaciones cuyo conocimiento no debe guardarse.
  - 2.2. Descartar las asociaciones "transitivas" (que pueden derivarse de otras)

1.2. Incluir las asociaciones importantes para nuestro dominio

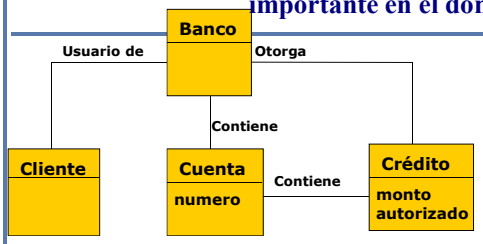
- A veces, hay asociaciones que son importantes en nuestro dominio y que no hemos obtenido en el paso anterior.
- Estas asociaciones suelen ser obvias y fáciles de detectar cuando dibujamos el diagrama.
- Es este el momento en que las podemos incluir.

Ejemplo: supongamos que tuviéramos esto



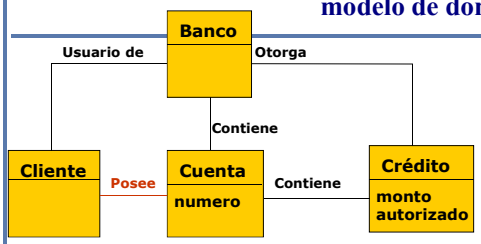
- Supongamos que en el paso anterior hubiéramos obtenido este diagrama.
- Lo hemos simplificado para que se vea con más claridad lo que deseamos decir.

Hay una asociación que es obvia e importante en el dominio



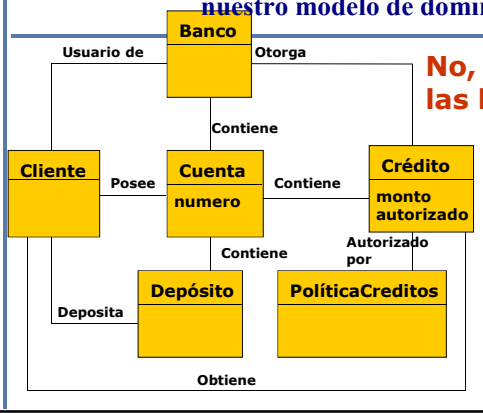
- Es la asociación entre cliente y cuenta.
- Naturalmente, cada cuenta pertenece a un cliente y esto no sólo es obvio, sino que además es importante y básico en nuestro dominio.

Esto nos haría incluir esta asociación en el modelo de dominio



- En nuestro caso, ya la teníamos, pero si no estuviera habría que incluirla.

¿Hay otras asociaciones importantes en nuestro modelo de dominio?



En el diagrama que tenemos

- No existen asociaciones importantes en nuestro dominio que no hayan sido incluidas.
- Por lo tanto, no modificaremos nuestro diagrama.
- Pero en otros casos pueden existir asociaciones importantes no incluidas en nuestro diagrama.
- Habría que incluirlas en este punto para completar el modelo.

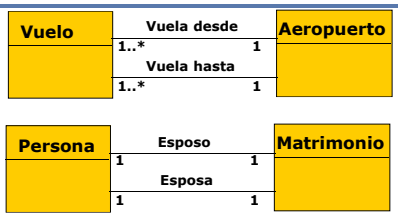
Método para añadir asociaciones al modelo de dominio

- 1. Incluir asociaciones.
  - 1.1. Incluir las asociaciones derivadas de la lista de asociaciones comunes.
  - 1.2. Incluir las asociaciones importantes para nuestro dominio.
  - 1.3. Determinar si entre dos clases hay diferentes asociaciones.
- 2. Descartar asociaciones.
  - 2.1. Descartar las asociaciones cuyo conocimiento no debe guardarse.
  - 2.2. Descartar las asociaciones "transitivas" (que pueden derivarse de otras)

1.3. Determinar si entre dos clases hay diferentes asociaciones

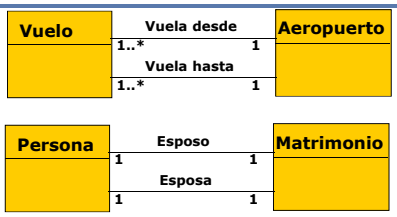
- El método que hemos aplicado hasta ahora sólo detecta una única asociación entre dos clases.
- Sin embargo, a veces hay más de una asociación entre dos clases.

A veces hay más de una asociación entre dos clases



- Esto pasa cuando una instancia de una clase A **se puede relacionar de varias maneras diferentes** con una instancia de la clase B.

A veces hay más de una asociación entre dos clases

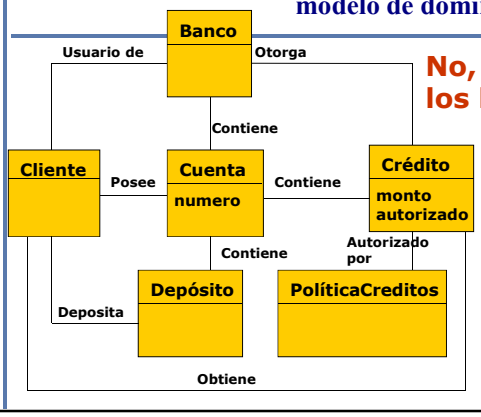


- Así un aeropuerto se puede relacionar de dos formas con un vuelo. Las dos formas son "aeropuerto de origen" o "aeropuerto de destino".
- Una persona se puede relacionar de dos maneras diferentes con un matrimonio. Puede ser esposo o esposa.

El método aplicado hasta ahora no detecta estos casos

- Por eso, es ahora el momento de pensar si hay casos de estos en nuestro modelo de dominio.

¿Hay casos así en nuestro modelo de dominio?



No, no los hay

Método para añadir asociaciones al modelo de dominio

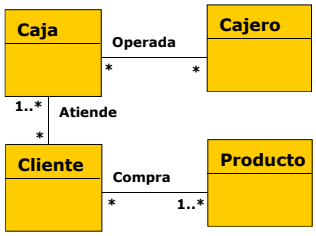
- 1. Incluir asociaciones.
  - 1.1. Incluir las asociaciones derivadas de la lista de asociaciones comunes.
  - 1.2. Incluir las asociaciones importantes para nuestro dominio.
  - 1.3. Determinar si entre dos clases hay diferentes asociaciones.
- 2. Descartar asociaciones.
  - 2.1. Descartar las asociaciones cuyo conocimiento no debe guardarse.
  - 2.2. Descartar las asociaciones "transitivas" (que pueden derivarse de otras)

2.1. Descartar las asociaciones cuyo conocimiento no debe guardarse

- De las asociaciones que tenemos en nuestro diagrama, hay algunas cuyo conocimiento debe conservarse y otras que no debe conservarse.
- Debemos descartar estas últimas y quedarnos con las primeras.
- Para cada asociación debemos preguntarnos:
  - **¿El conocimiento de esta asociación debe guardarse o no?**
- La respuesta a esta pregunta dependerá de las propiedades de nuestro sistema.

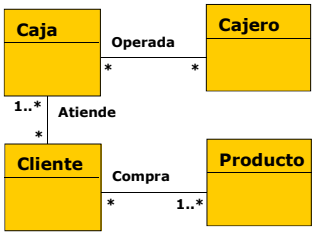
Ejemplo de asociaciones cuyo conocimiento no debe guardarse

- Supongamos que modelamos un sistema de facturación de un supermercado (simplificado).
- Un cliente compra unos productos en una caja operada por un cajero.



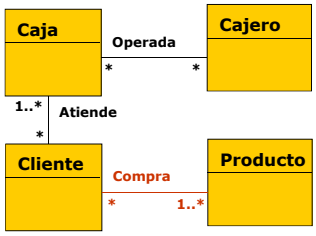
Para todas las asociaciones, preguntamos la misma pregunta

- **¿El conocimiento de esta asociación debe guardarse o no?**
- La respuesta a esta pregunta dependerá de las propiedades de nuestro sistema.



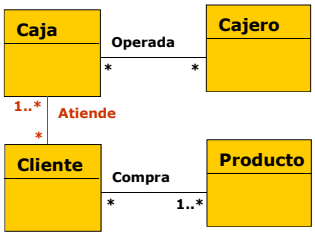
¿El conocimiento de la asociación "Compra" debe guardarse?

- En la mayoría de sistemas sí, pues debemos guardar las facturas que genera el cliente en el supermercado.



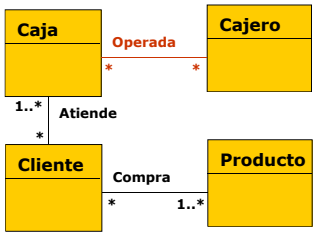
¿El conocimiento de la asociación "Atiende" debe guardarse?

- En la mayoría de sistemas no, pues no nos interesa qué cliente acude a qué caja en cada ocasión.



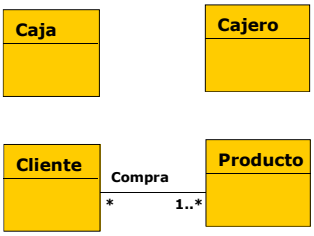
¿El conocimiento de la asociación "Operada" debe guardarse?

- Depende del sistema, si el sistema controla a los cajeros para cualquier responsabilidad de pérdida de dinero, sí. En caso contrario, no. Supondremos que no.

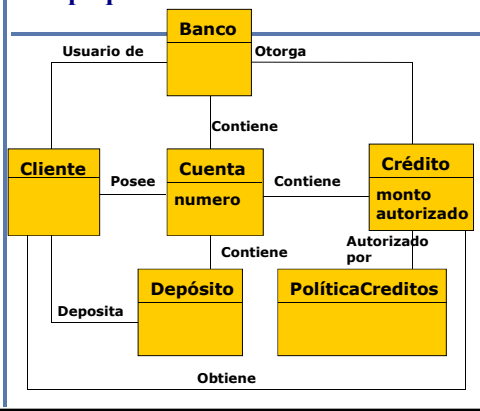


Sólo mantendremos las asociaciones cuyo conocimiento debe guardarse

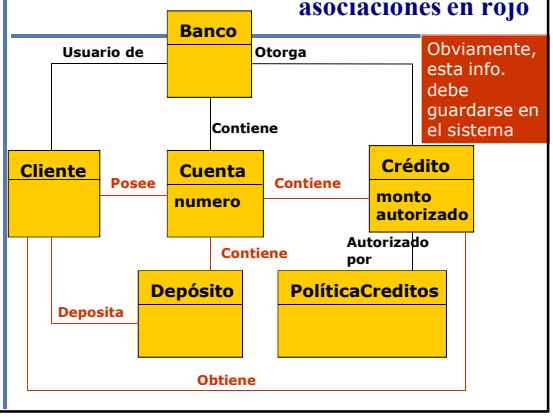
- En nuestro caso, sólo mantendremos la asociación "Compra"



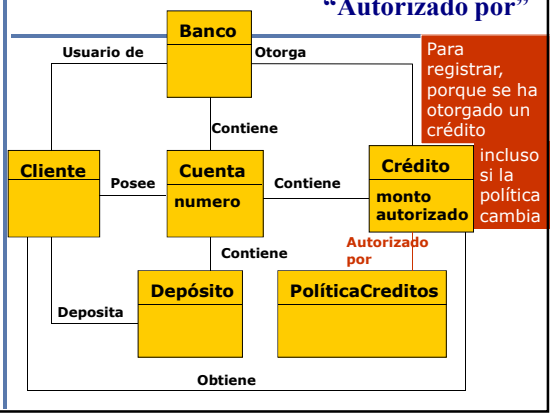
Apliquemos a nuestro modelo de dominio



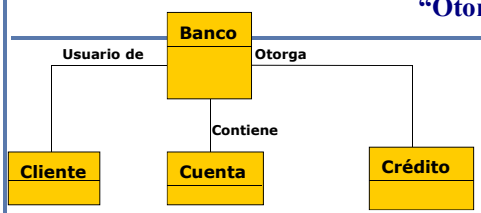
Interesa conservar el conocimiento de las asociaciones en rojo



Nos interesa conservar el conocimiento de "Autorizado por"

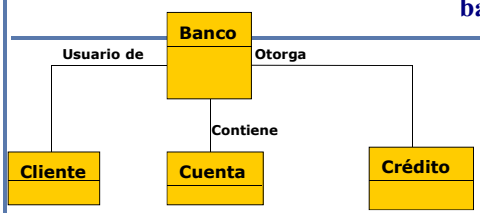


Veamos la asociación "Otorga"



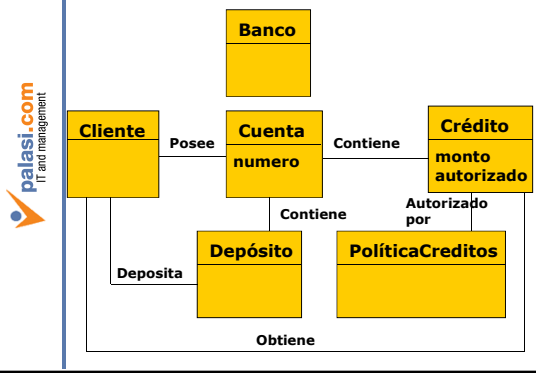
- ¿Debe conservarse su conocimiento? Como siempre dependerá de las propiedades del sistema.
- Si el sistema manejara varios bancos, nos interesaría saber cuál de ellos ha otorgado un determinado crédito.
- Si el sistema sólo maneja un banco, se supone que es el quien otorga el crédito. No nos interesa guardar esta

Suponemos que el sistema sólo maneja un banco



- Es la situación más usual.
- El conocimiento de la asociación "Otorga" no necesitamos guardarlo. Eliminaremos esta asociación.
- Lo mismo podemos decir de las asociaciones "Usuario de" y "Contiene". Como sólo hay un banco, no necesitamos conservar su conocimiento y las eliminamos.

Nuestro modelo de dominio queda

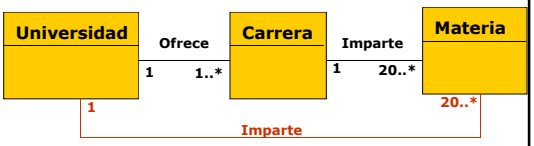


Método para añadir asociaciones al modelo de dominio

- 1. Incluir asociaciones.
  - 1.1. Incluir las asociaciones derivadas de la lista de asociaciones comunes.
  - 1.2. Incluir las asociaciones importantes para nuestro dominio.
  - 1.3. Determinar si entre dos clases hay diferentes asociaciones.
- 2. Descartar asociaciones.
  - 2.1. Descartar las asociaciones cuyo conocimiento no debe guardarse.
  - 2.2. Descartar las asociaciones "transitivas" (que pueden derivarse de otras)

Recordemos que asociaciones que pueden darse de manera transitiva

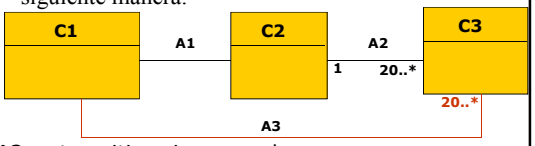
- Es decir, asociaciones que se derivan de otras.



Estas asociaciones transitivas no aportan nada nuevo. Sólo hacen que complicar nuestro modelo de dominio. Por eso, hay que eliminarlas

Definición formal de una asociación transitiva

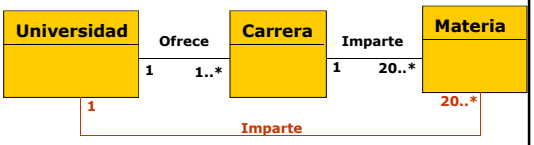
- Supongamos que tenemos 3 clases conceptuales C1, C2 y C3, relacionadas por A1, A2 y A3 de la siguiente manera.



A3 es transitiva si se cumple que:
1 instancia de C1 está relacionada por A3 con una de C3 si y solo si:
C1 está relacionada por A1 con 1 instancia de C2.
Esta instancia de C2 está relacionada por A2 con la instancia de C3.

Esto parece muy difícil pero es sencillo

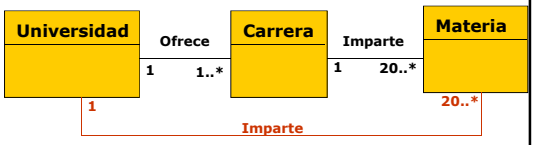
- Se ve mejor con un ejemplo.



La asociación en rojo es transitiva porque:
Una universidad imparte una materia si y solo si:
- La universidad ofrece una carrera.
- La carrera ofrece esa materia

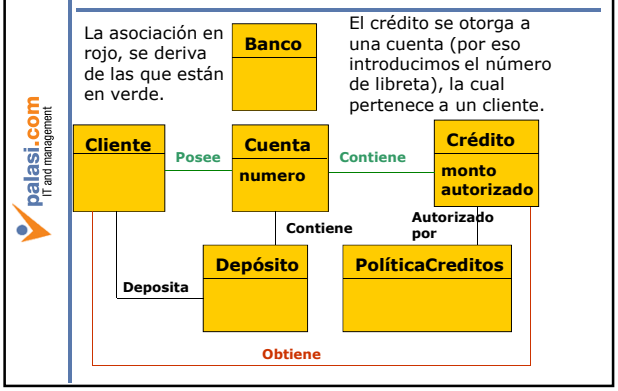
Sin embargo, la mayoría de las veces no es necesario esto

- Las asociaciones transitivas "saltan" a la vista.

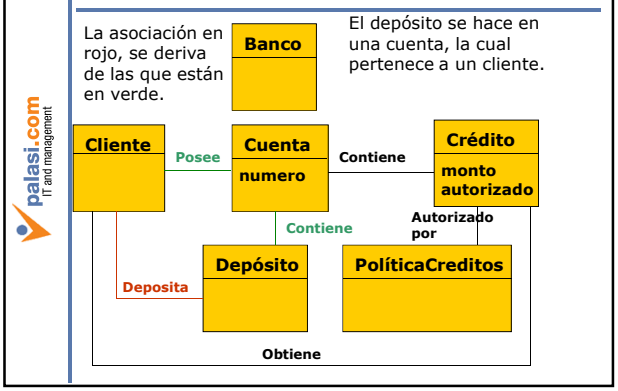


Aquí se ve que la asociación en rojo, se deriva de las dos que están en negro.

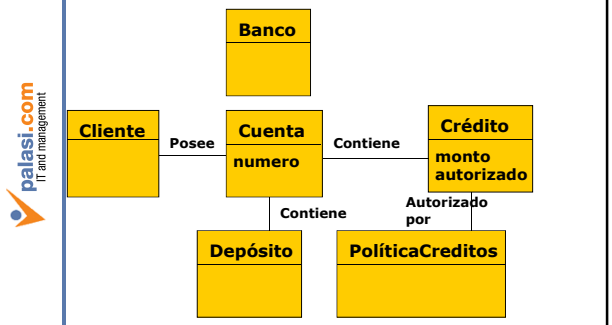
Aplicando a nuestro modelo de dominio



Aplicando a nuestro modelo de dominio



Nuestro modelo de dominio queda



Nota importante

- Aunque aquí estas dos tareas:
  - Descartar asociaciones cuyo conocimiento no debe guardarse.
  - Descartar asociaciones transitivas.

  se explican una detrás de la otra.
- Lo normal es que se realicen simultáneamente.
- Ya que cuando examinamos las asociaciones vemos ejemplos de los dos tipos y se aprovechan para descartar en ese mismo momento.

Método para añadir asociaciones al modelo de dominio

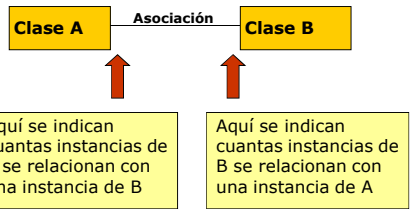
- 1. Incluir asociaciones.
  - 1.1. Incluir las asociaciones derivadas de la lista de asociaciones comunes.
  - 1.2. Incluir las asociaciones importantes para nuestro dominio.
  - 1.3. Determinar si entre dos clases hay diferentes asociaciones.
- 2. Descartar asociaciones.
  - 2.1. Descartar las asociaciones cuyo conocimiento no debe guardarse.
  - 2.2. Descartar las asociaciones "transitivas" (que pueden derivarse de otras)

Con el método que hemos acabado de explicar

- Ya tenemos las asociaciones del modelo del dominio.
- Ahora sólo nos hace falta especificar las multiplicidades de los roles.
- Recordemos que un rol es el extremo de una asociación.
- La multiplicidad son los numeritos que aparecen en un rol y que indican cuantas instancias de una clase se relacionan con instancias de la otra.

Mail: palasi@palasi.com Web: www.palasi.com

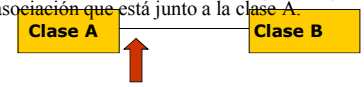
Multiplicidad



- A esto se le llama "multiplicidad".
- La multiplicidad se indica con los números pequeños a cada extremo de la asociación

Obteniendo la multiplicidad

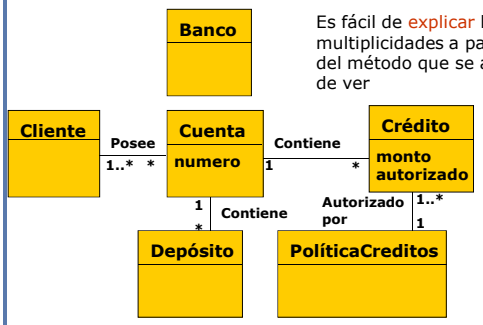
- Sencillo. Sólo se aplica la definición de multiplicidad.
- Para cada clase A que se relaciona por asociación con B, se pregunta:
  - **¿Cuántas instancias de A se relacionan con una instancia de B?**
  - Esto se traduce a numeritos.
  - La traducción se indica en el rol (extremo) de la asociación que está junto a la clase A.



Recordemos: Traduciendo a numeritos

Asociación *	Clase C	* significa "cero, uno o más"
Asociación 1..*	Clase C	1..* significa "uno o más"
Asociación 3..*	Clase C	3..* significa "tres o más"
Asociación 3	Clase C	3 significa "exactamente tres"
Asociación 3..5	Clase C	3..5 "entre tres y cinco"
Asociación 3, 5, 7	Clase C	3,5,7 "tres,cinco y siete"

Aplicando esto, nuestro modelo de dominio queda



Es fácil de explicar las multiplicidades a partir del método que se acaba de ver

Con esto hemos acabado de ver las asociaciones del modelo de dominio

- Sin embargo, es importante repetir que:
- Lo importante del modelo de dominio es obtener las clases. Obtener las asociaciones entre clases no es tan importante.
- Si pasamos mucho tiempo detectando relaciones, esto es una señal de que no tenemos bien establecidas las prioridades.

Ejercicio

- Añadan asociaciones al modelo de dominio del sistema que produce cotizaciones de computadora.

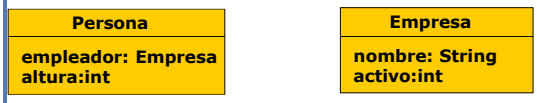
Mail: palasi@palasi.com Web: www.palasi.com

Método para obtener un modelo de dominio

- 1. Obtener las clases conceptuales.
- 2. Añadir las asociaciones entre las clases.
- 3. Añadir atributos a las clases.
- 4. Refinar las asociaciones.

3. Añadir atributos a las clases

- Hay dos tipos de atributos:
  - Atributos que son clases.
  - Atributos que no son clases.



Como vemos, Persona tiene un atributo "empleador" que es de clase Empresa. Sin embargo, el atributo "altura" no es una clase pues sólo es un valor numérico (un entero).

3. Añadir atributos a las clases

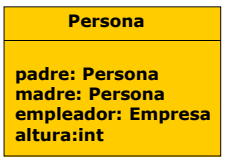
- Sólo los atributos que no son clases, se modelan como atributos en el modelo de dominio.
- Los atributos que son clase, se modelan como una asociación.

Vemos cómo "empleador" ha pasado a ser una asociación. "altura" es un atributo no clase y, por eso, se modela como atributo.

¿Cómo saber cuáles son atributos no clase y cuáles son atributos clase?

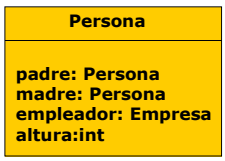
- Recordemos la regla de oro:
- A. Son atributos no clase si son de un "tipo de datos" primitivo: número, texto, fecha, hora, sí/no, color …
- B. Son atributos no clase si intentamos asociarle propiedades (atributos) y no se puede.
- Estos dos criterios son equivalentes.

Ejemplo



- "altura" es atributo no clase porque es de un tipo primitivo (entero).
- "padre", "madre", "empleador" no son de tipo primitivo, por lo tanto son atributo clase.

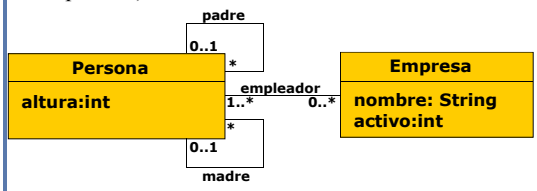
De forma equivalente



- "altura" es atributo no clase porque no se le pueden añadir atributos (propiedades) a una altura.
- A "padre", "madre", "empleador" se le pueden añadir propiedades (atributos).

El modelo de dominio quedaría

- Los atributos no clase (altura) se modelan como atributos. Los atributos clase (padre, madre, empleador) se modelan como asociaciones.



Fíjense que una clase puede tener una asociación con ella misma. Esto no presenta ningún problema.

Como conclusión

- Este paso del método no debería llamarse.
- Añadir atributos a las clases.

sino

- Añadir atributos no clases a las clases.

Añadir atributos no clases a las clases

- Una clase conceptual es de la vida real y por lo tanto puede tener infinidad de atributos. Así la clase "Auto": color, año de fabricación, alto ancho, color de las ventanas, potencia de los faros, etc.
- La mayoría de estos atributos de la vida real no nos interesarán para nuestra aplicación.
- Además, el objetivo del modelo de dominio no es obtener atributos, sino obtener las clases conceptuales.
- Por ello, hemos de limitar el número de atributos a

¿Qué quiere decir "importantes"?

Incluiremos en nuestro modelo de dominio:

1. los atributos que detectamos cuando identificamos las clases conceptuales.
2. los atributos para los que **la documentación sugiere o implica** una necesidad de recordar información.
3. los atributos que no incluye la documentación pero que conocemos y son obviamente necesarios.

- La documentación puede estar compuesta por requerimientos (por ejemplo, casos de uso) o por cualquier documento que hayamos conseguido sobre el dominio del sistema.
- Fíjense en las palabras "sugiere" e "implica" la

¿Qué quiere decir esto?

3. los atributos que no incluye la documentación pero que conocemos y son obviamente necesarios.

- Imaginemos que estamos modelando una libreta de banco y la documentación no dice que tiene un número.
- Pero esto lo conocemos nosotros y es tan obvio y necesario que no se puede excluir.
- Por eso, lo incluimos como atributo en

Apliquemos a nuestro caso

- Los atributos que íbamos arrastrando ya los conocemos.
- Examinemos el caso de uso para ver si nos sugiere otros atributos importantes.
- Para ello examinaremos el caso de uso con el diagrama de clases conceptuales al lado (aquí no caben)

Escenario principal:

1. El Cliente selecciona la opción de crédito.
2. El Sistema pide el monto del crédito y el número de libreta para depositar.
3. El Cliente entra el monto del crédito y el número de libreta.
4. El Sistema registra el crédito, deposita el monto en la libreta y presenta que el crédito ha sido

"Deposita el monto de la libreta"

- Parece lógico que un dato importante de un depósito es su monto.
- Esto lo incluiremos como un atributo.

Escenario principal:

1. El Cliente selecciona la opción de crédito.
2. El Sistema pide el monto del crédito y el número de libreta para depositar.
3. El Cliente entra el monto del crédito y el número de libreta.
4. El Sistema registra el crédito, deposita el monto en la libreta y presenta que el crédito ha sido

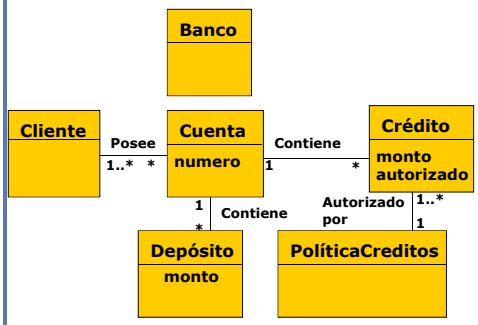
En este caso, no nos sugieren más atributos

- El caso de uso es muy sencillo. En un caso más real, añadiríamos más atributos.
- De todas maneras, recordemos que usamos un modelo iterativo: esto lo iremos perfeccionando.

Escenario principal:

1. El Cliente selecciona la opción de crédito.
2. El Sistema pide el monto del crédito y el número de libreta para depositar.
3. El Cliente entra el monto del crédito y el número de libreta.
4. El Sistema registra el crédito, deposita el monto en la libreta y presenta que el crédito ha sido

Aplicando esto, nuestro modelo de dominio queda



Ejercicio

- Añadan atributos al modelo de dominio del sistema que produce cotizaciones de computadora.

Método para obtener un modelo de dominio

- 1. Obtener las clases conceptuales.
- 2. Añadir las asociaciones entre las clases.
- 3. Añadir atributos a las clases.
- 4. Refinar las asociaciones.

Refinando las asociaciones del modelo de dominio

- Hasta ahora, todas las asociaciones las habíamos considerado de la misma forma y las habíamos representado con la misma notación.
- Esto es una opción correcta al dibujar nuestro modelo de dominio.
- Pero hay clases de asociaciones que son especiales y UML les da una notación especial.
- Se puede refinar el modelo de dominio, distinguiendo estas clases de asociaciones específicas y representándolas con su notación específica.

Mail: palasi@palasi.com Web: www.palasi.com

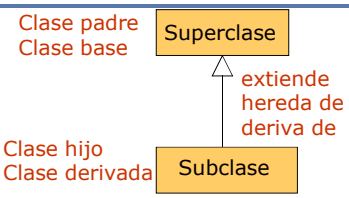
Clases de asociaciones específicas

- 1. Herencia.
- 2. Agregación.
- 3. Asociaciones con clase incorporada.

Clases de asociaciones específicas

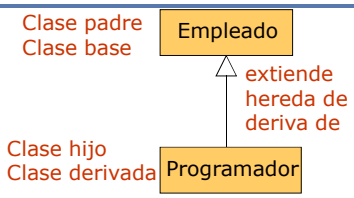
- 1. Herencia.
- 2. Agregación.
- 3. Asociaciones con clase incorporada.

Ya conocemos la asociación de herencia



- Es una asociación entre dos clases, llamadas subclase y superclase.
- Todas las instancias de la subclase son también instancias de la superclase.
- Se le llama asociación "ES-UN" y se representa por la notación que se muestra.

Ejemplo

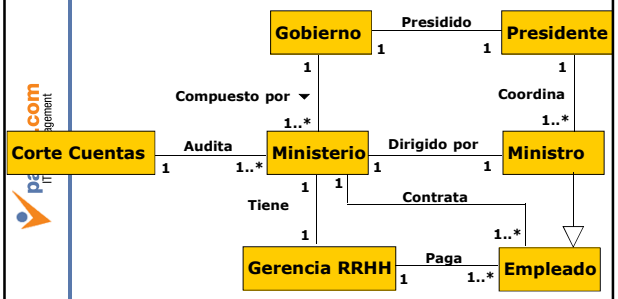


- "Programador" ES-UN "Empleado"
- Todas las instancias de programador son también instancias de empleado.

Podemos refinar el modelo de dominio

- Detectando todas las relaciones de herencia e indicándolas con su notación.

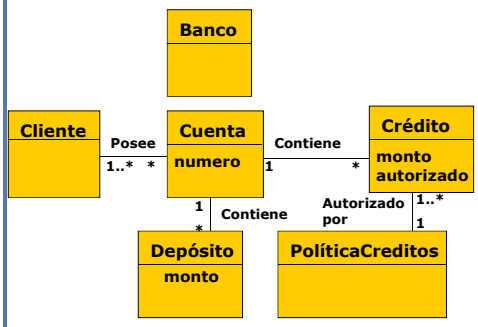
Habíamos visto este modelo de dominio



- Lo enriquecemos con la relación de herencia entre "Ministro" y "Empleado"

Mail: palasi@palasi.com Web: www.palasi.com

En el ejemplo que vamos siguiendo no hay ninguna relación de herencia

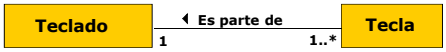


Clases de asociaciones específicas

- 1. Herencia.
- 2. Agregación.
- 3. Asociaciones con clase incorporada.

Agregación

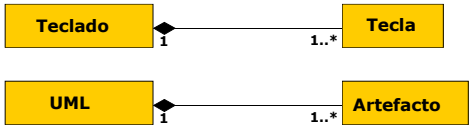
- Es un tipo de asociación que modela relaciones entre el todo (llamado "el compuesto") y las partes.
- Sería la asociación "Es parte de".



- Más formalmente: A y B tienen una relación de agregación si **todas las instancias de B son parte de instancias de A**

UML tiene una notación especial para la agregación

- Se representan como una asociación pero con rombo al lado de la clase que es el compuesto.
- Normalmente no se les pone un nombre (pues se supone que el nombre es "Es parte de"), pero podría incluirse si así se quiere.

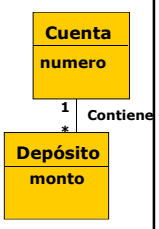


Una pregunta para detectar agregaciones

- Si queremos saber si hay una agregación entre dos clases A y B, nos podemos preguntar.
- **¿Un B es parte de un A?**

A veces la respuesta no es tan obvia

- A veces, no sabemos si una asociación es una agregación o no.
- En nuestro caso: ¿El depósito es parte de una cuenta?
- Por una parte, podemos ver los depósitos como líneas de detalle de la cuenta y, por lo tanto, parte de ella. Esto es obvio si pensamos en la cuenta como una libreta y en los depósitos como líneas.
- Si lo vemos de forma abstracta, no parece tan natural pensar en un depósito como en "parte" de una cuenta.



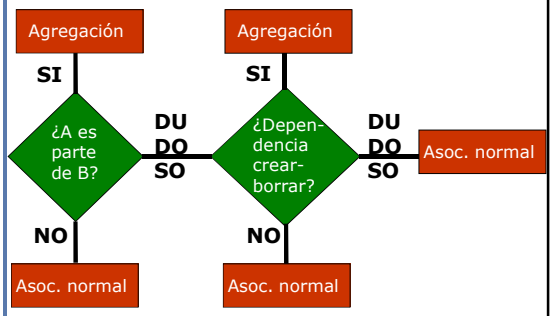
Criterios para cuando la respuesta no es tan obvia

- Se puede establecer alguna relación de tipo todo-partes (aunque no sea tan obvia) y además hay una dependencia de crear-borrar, es decir,:
  - Cuando se borra el todo, se deben borrar las partes. O bien, no se puede borrar el todo, si antes no se han borrado las partes. Así: Factura-Líneas de detalle de factura.
  - No se puede crear la parte, si el todo no está creado. Así: Factura-Líneas de detalle de

Si todavía así, tenemos dudas

- **No indicamos la agregación.**
- La modelamos como una asociación normal.
- No se debe emplear demasiado tiempo identificando agregaciones: este no es el objetivo del modelo de dominio.
- Siempre podemos refinar el modelo después, de forma iterativa.
- Además, indicar la agregación tiene beneficios, pero no son de vital importancia por lo cual se puede prescindir.

Resumen

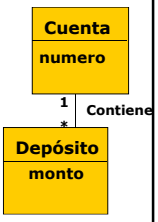


¿Cuáles son los beneficios de la agregación?

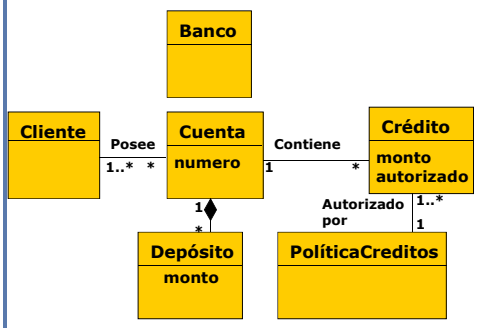
- Los beneficios no se ven en el análisis sino a la hora del diseño.
- La agregación nos permite identificar quien creará la parte (el compuesto).
- La agregación nos permite establecer relaciones de integridad referencial:
  - cuando borramos un compuesto, debemos borrar todas sus partes.
  - no podemos borrar el compuesto, si antes no hemos borrado sus partes.
- Esto es importante en cuanto a objetos y

En el caso anterior

- Se puede establecer un tipo de relación de tipo "todo-partes" entre el depósito y la cuenta (aunque no sea obvia).
- Además, si borramos la cuenta, deberemos borrar los depósitos.
- Por lo tanto, podemos modelar esta asociación como una agregación
- De todas maneras, si la dejamos como una asociación normal, tampoco es ningún problema.



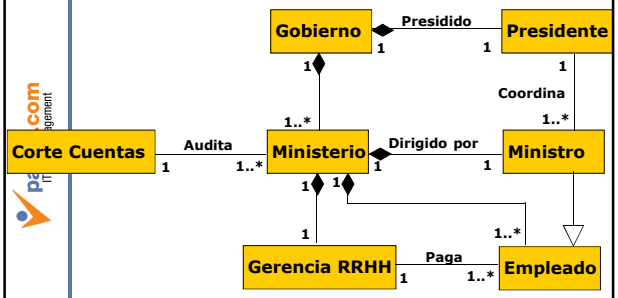
En las otras asociaciones es más difícil establecer relación todo-partes



Por eso, nuestro modelo de dominio queda así.

Mail: palasi@palasi.com Web: www.palasi.com

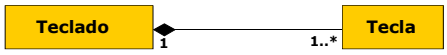
Otro ejemplo



- Fíjense en que las agregaciones pueden llevar nombres.

Dos tipos de agregación (1)

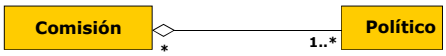
- **Agregación compuesta o composición.** La parte sólo puede ser parte de UN ÚNICO compuesto.
- Una tecla es parte de un único teclado. Un dedo es parte de una única mano.



- Se detecta fácilmente porque la multiplicidad en el extremo del compuesto es **1** o bien **0..1**

Dos tipos de agregación (2)

- **Agregación compartida.** La parte puede ser parte de VARIOS compuestos.
- Un político puede ser parte de varias comisiones.



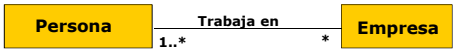
- Se detecta fácilmente porque la multiplicidad en el extremo del compuesto es diferente de **1** y de **0..1**
- Se representa en UML por un **rombo vacío**.

Clases de asociaciones específicas

- 1. Herencia.
- 2. Agregación.
- 3. Asociaciones con clase incorporada.

Los atributos que no se sabe donde van

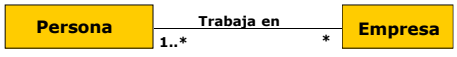
- Supongamos un sistema que modela los diferentes empleos que tiene una persona (por ejemplo, en una AFP).



- Como vemos, una persona puede trabajar en varias empresas.

¿Qué pasa si queremos modelar el salario?

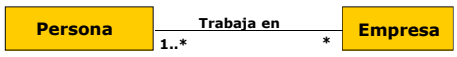
- El salario es un dato importante de nuestro sistema y queremos modelarlo.
- Claramente, es un atributo pues es de un tipo primitivo (número).
- ¿Dónde irá el atributo "salario"?



Mail: palasi@palasi.com Web: www.palasi.com

¿Dónde irá el atributo "salario"?

- En "Persona" no parece, pues una misma persona puede tener varios salarios diferentes si trabaja en varias empresas diferentes.
- En "Empresa" tampoco, pues una empresa tiene muchos salarios dependiendo de las personas que allí trabajan.

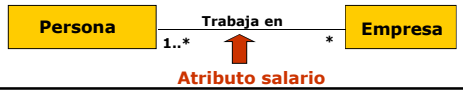


¿Dónde irá el atributo "salario"?

- Parece claro que salario es una propiedad no de "Persona" ni de "Empresa" sino de la relación que hay entre las dos.
- Cuando una Persona y una Empresa acuerdan una relación de trabajo, fijan un salario para esa relación.
- **Por tanto, el atributo "salario" es propiedad de la asociación "Trabaja en".**

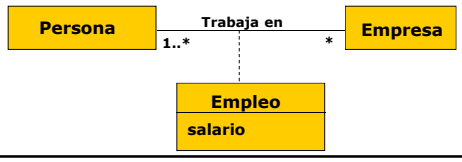
Un momento, por favor

- En UML las asociaciones no pueden tener atributos, sólo las clases.
- ¿Cómo hacemos entonces para relacionar el atributo "salario" a la asociación "Trabaja en"?
- UML permite relacionar clases a las asociaciones. A esto se le llama "clases de asociación".



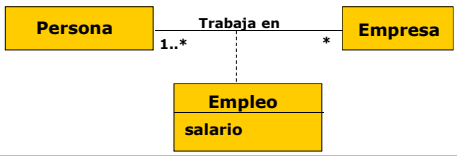
Clases de asociación

- Clases que se relacionan con las asociaciones y en las cuales podemos incluir los atributos de las asociaciones.
- Se unen a la asociación correspondiente por medio de una línea discontinua. Así:



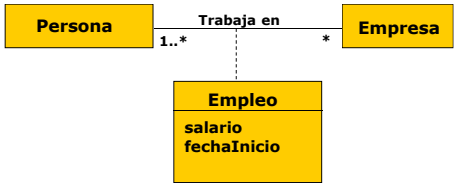
Ahora vemos donde se incluye el atributo "salario"

- En una clase relacionada con la asociación "Trabaja en".
- Esta clase llamada "Empleo" es la apropiada pues
  - Cada relación entre persona y empresa (cada empleo) tiene un salario diferente.



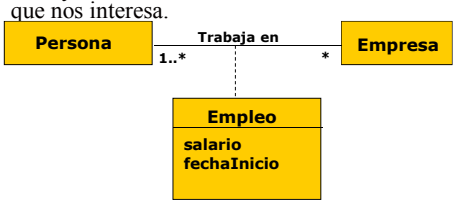
La clase de asociación "Empleo" puede tener los atributos que se quiera

- Por ejemplo, fecha de inicio de trabajo, cargo, superior, etc.
- Es una clase más de nuestro modelo de dominio.



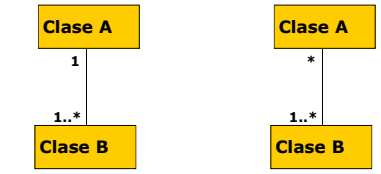
Cuándo incluir una clase de asociación

- Cuando se cumplan entras dos condiciones:
- 1. Existe una asociación "varios a varios" entre varias clases.
- 2. Hay información relacionada a esta asociación que nos interesa.



Recordemos: Asociaciones "varios a varios"

- Se llama así a las asociaciones cuyas dos multiplicidades permiten la posibilidad de varias instancias.

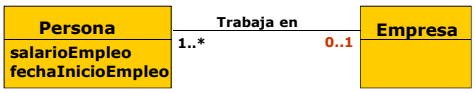


NO, el rol de Gobierno no permite varias instancias. Sólo una.

SI, los dos roles permiten varias instancias.

¿Y si la asociación no es "varios a varios"?

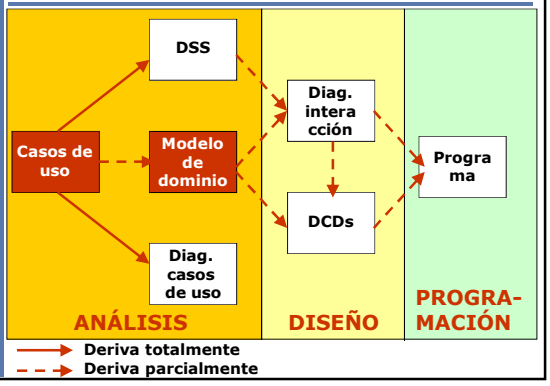
- Entonces el atributo se puede incluir en una de las dos clases y no necesitamos la clase de asociación.
- Supongamos que por ley, una persona sólo pudiera tener un empleo.



Con esto conseguiremos la primera versión del modelo de dominio

- Esta versión la podemos ir refinando iterativamente.
- Ahora bien, cuando el sistema es grande, es difícil realizar el modelo de dominio con este sistema: aparecen demasiadas clases candidatas y asociaciones que después debemos descartar.
- La solución se trata en **dividir el sistema en partes (por ejemplo, en casos de uso)** y hacer el modelo de dominio de cada una de ellas.
- Después las partes del modelo de dominio que hemos obtenido se juntan en una sola y **se resuelven las posibles inconsistencias.**

Relación del modelo de dominio con otros artefactos



Con esto hemos acabado la descripción del análisis O-O

- Dado todo lo que habemos dado, podría parecer que el análisis realizado en la primera iteración puede durar semanas (para un sistema de tamaño comercial).
- Sin embargo, una vez se dominan las técnicas básicas del modelado de casos de uso y del modelo de dominio, esto debería durar solamente unos pocos días.
- La idea no es conseguir unos requerimientos perfectos, sino una primera aproximación que refinaremos en posteriores iteraciones.

Ejercicio

- Vean si el modelo de dominio del sistema que produce cotizaciones de computadoras se debe refinar con herencia, agregación y/o clases de asociación. Si es así, incluyan estos elementos en él.

Ejercicio

- Obtengan el modelo de dominio del sistema de facturación que hemos visto anteriormente.

Temario del curso

- 1. Introducción al A & D O-O y a UML.
- 2. Análisis orientado a objetos con UML.
- 3. Diseño orientado a objetos con UML.
- 4. Conclusiones finales.

3. Diseño orientado a objetos con UML

- 3.1. Introducción al diseño orientado a objetos.
- 3.2. Diagramas de secuencia.
- 3.3. Diagramas de colaboración.
- 3.4. Patrones para asignar responsabilidades.
- 3.5. Método para obtener los diagramas de interacción.
- 3.6. Visibilidad.
- 3.7. Diagramas de clases de diseño.

3. Diseño orientado a objetos con UML

- 3.1. Introducción al diseño orientado a objetos.
- 3.2. Diagramas de secuencia.
- 3.3. Diagramas de colaboración.
- 3.4. Patrones para asignar responsabilidades.
- 3.5. Método para obtener los diagramas de interacción.
- 3.6. Visibilidad.
- 3.7. Diagramas de clases de diseño.

Diseño de un programa

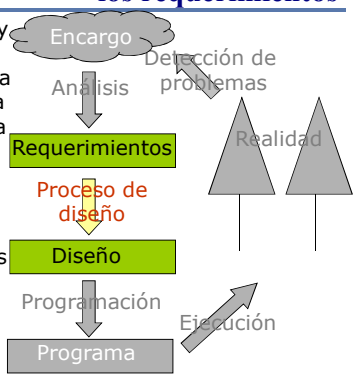
- Mientras los requerimientos explican QUÉ debe hacer un programa, el diseño refleja CÓMO lo debe hacer.
- Objetivos:
  - De los requerimientos: Hacer la cosa correcta.
  - Del diseño: Hacer la cosa correctamente.
- El diseño son los planos de un programa. Cómo está construido: qué clases tiene, cómo interactúan clases, qué capas, etc.

Mail: palasi@palasi.com Web: www.palasi.com

Para hacer un diseño, nos basamos en los requerimientos

Los casos de uso y el modelo de dominio nos dan la información con la cual empezamos a descubrir el diseño.

Se sigue un modelo iterativo: los requerimientos y el diseño se realizan en paralelo, influyéndose mutuamente.



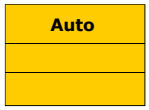
El diseño del programa constará de dos artefactos.

- De hecho, puede constar de más pero estos dos son los principales.
- Estos dos artefactos son:
  - **El diagrama de interacción.**
  - **El diagrama de clases de diseño.**
- Los dos diagramas representan clases de software (también llamadas clases de diseño o de programa) .

Recordemos: clase conceptual

- Conjunto de **objetos del mundo real** que son similares.

- Se representan con notación de orientación a objetos.

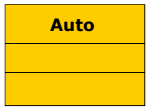


Recordemos: Clases de software

- Construcciones de un lenguaje de programación O-O.

```
class Auto {
   String marca;
   String modelo;
   int potencia;
   void arrancar(){
   }
   ...
}
```

- Se representan también con notación de orientación a objetos.



Aunque se representan igual, es necesario distinguirlas

- Como el número 0 y la letra O.
- Las clases conceptuales representan la realidad.
- Las clases de software representan el programa.

Clase de software

```
class Auto {
   String marca;
   String modelo;
   int potencia;
   void arrancar(){
   }
   ...
}
```

En el diseño sólo tratamos las clases de software

- Ya que el diseño trata sobre el programa, no sobre la realidad.
- Las clases conceptuales eran para el modelo de dominio.

Clase de software

```
class Auto {
   String marca;
   String modelo;
   int potencia;
   void arrancar(){
   }
   ...
}
```


Mail: palasi@palasi.com Web: www.palasi.com

Hay clases de software que modelan clases conceptuales

- Por ejemplo, la clase Auto del programa modela el conjunto de autos (clases conceptuales)
- **Lo modela pero no son lo mismo.**
- Las llamamos clases de dominio.

Clase conceptual

Clase de software

```
class Auto {
   String marca;
   String modelo;
   int potencia;
   void arrancar(){
   }
   ...
}
```

Hay clases de software que no modelan clases conceptuales

- Por ejemplo, las clases que se incluyen aquí.
- Se llamarían clases artificiales o clases que no son de dominio.

```
class SeguridadSistema {
   void autorizaAcceso(){
   }
   ...
}
```

```
class AccesoBD {
   void grabaRegistro(){
   }
   ...
}
```

```
class Formulario {
   String título;
   int color;
   void dibujar(){
   }
   ...
}
```

Las clases de software se dividen en dos tipos

- 1. Clases que modelan clases conceptuales (de la vida real). Las llamaremos clases de dominio. Ejemplo: clase Auto.
- 2. Clases que no modelan clases conceptuales. Las llamaremos clases que no son de dominio o clases artificiales. Ejemplo: clase Formulario.

Otra división de las clases de software

- 1. Clases persistentes. Son aquellas cuyas instancias debemos guardar en un soporte permanente para que la información se conserve entre ejecuciones . Ejemplo: clase Factura.
- 2. Clases no persistentes. Son aquellas cuyas instancias no debemos guardar, ya que su información sólo nos interesa en la presente ejecución. Ejemplo: clase Formulario.

Otra división de las clases

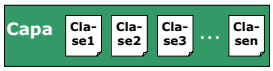
- En un programa, tenemos tres tareas:
- 1. Manejar la entrada y salida de información hacia el usuario. Esto se llama "presentación".
- 2. Hacer los cálculos y el procesamiento de datos del sistema. Indican las reglas del negocio que vamos a implementar. A esto le llamaremos "cálculo o negocio".
- 3. Guardar y recuperar los datos para que no se pierdan entre ejecuciones. A esto le llamaremos "acceso a datos" (suele implicar una BD

Podemos dividir las clases entre estas tres tareas

- Tres tipos de clases:
- 1. Clases de presentación. Son todas clases artificiales. Suelen ser no persistentes.
- 2. Clases de negocio. Algunas son clases de dominio y otras son clases artificiales. Pueden ser persistentes o no persistentes.
- 3. Clases de acceso a datos. Son todas clases artificiales. Suelen ser no persistentes.

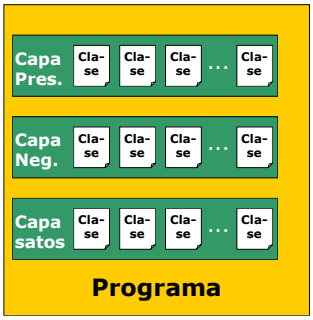
Mail: palasi@palasi.com Web: www.palasi.com

De hecho, estas clases se suelen agrupar en capas



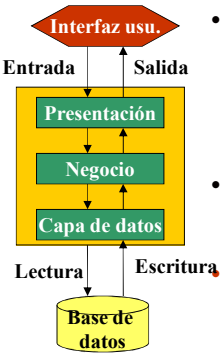
- Cada capa es un conjunto de clases de un mismo tipo. Hay tres capas en un programa.
- 1. Capa de presentación.
- 2. Capa de negocio.
- 3. Capa de acceso a datos.

Hay tres capas en un programa



- De forma abreviada

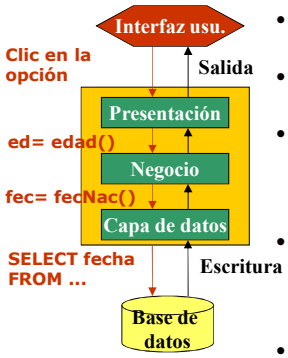
Las capas suelen ser organizadas según esta arquitectura



- A esto se le llama arquitectura en tres capas y se ha convertido en el estándar moderno de arquitectura.
- Cada capa sólo accede a la inferior.

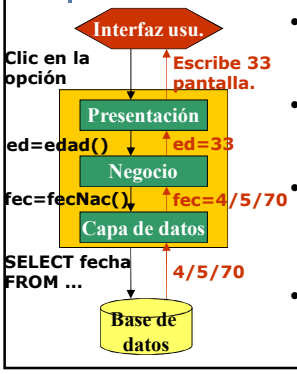
Es decir, las clases de una capa sólo llaman a métodos de clases de la capa inferior.

Ejemplo : calcular la edad de un empleado (1)



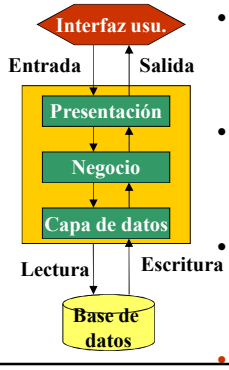
- 1. El usuario selecciona el empleado en la interfaz.
- 2. Esto hace que se llame a la capa de presentación.
- 3. La capa de pres. llama a un método de la de negocio que indica la edad de un empleado.
- 4. La capa de negocio pregunta la fecha de nacimiento a la capa de datos.
- 5. La capa de datos pregunta

Ejemplo : calcular la edad de un empleado (2)



- 6. La base de datos devuelve la fecha de nacimiento a la capa de datos.
- 7. La capa de datos devuelve la fecha de nacimiento a la capa de negocio.
- 8. La capa de negocio calcula la edad restando la fecha actual y la devuelve a la capa de presentación.
- 9. La capa de presentación lo muestra por pantalla.

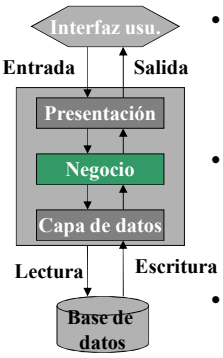
Ventajas de la arquitectura en tres capas



- Se puede cambiar de base de datos sin tener que reprogramarlo todo.
- Se puede cambiar de presentación (Web) sin tener que reprogramarlo todo.
- Es más fácil de programar y de mantener y más flexible a los cambios.
- Es el estándar actual

Mail: palasi@palasi.com Web: www.palasi.com

Sin embargo



- Aquí no vamos a ver el diseño de la capa de presentación, por falta de tiempo.
- Tampoco vamos a ver el diseño de la capa de datos (se suele comprar en vez de programar).
- Sólo estudiaremos la capa de negocio.

Esto quiere decir que sólo veremos clases de negocio

- 1. Clases de presentación. Son todas clases artificiales.
- 2. Clases de negocio. Algunas son clases de dominio y otras son clases artificiales.
- 3. Clases de acceso a datos. Son todas clases artificiales.

¿Cómo clasificamos las clases de negocio?

- Las podemos clasificar en las siguientes clases:
- **2.1. Clases de dominio.** Las clases de negocio que modelan clases de la vida real. Por ejemplo: Auto.
- **2.2. Clases controladoras.** Son las que coordinan las otras clases. Son como el director de orquesta.
- **2.3. Otras clases artificiales.** Sirven para hacer más flexible el programa para futuros cambios.

¿Cómo obtenemos estas clases?

- **2.1. Clases de dominio.** Muchas se encuentran en el modelo de dominio. Otras iremos descubriéndolas iterativamente o mediante el diseño.
- **2.2. Clases controladoras.** Las obtendremos a partir del diseño.
- **2.3. Otras clases artificiales.** Las obtendremos a partir del diseño.

3. Diseño orientado a objetos con UML

- 3.1. Introducción al diseño orientado a objetos.
- 3.2. Diagramas de secuencia.
- 3.3. Diagramas de colaboración.
- 3.4. Patrones para asignar responsabilidades.
- 3.5. Método para obtener los diagramas de interacción.
- 3.6. Visibilidad.
- 3.7. Diagramas de clases de diseño.

Recordemos: El diseño del programa constará de dos artefactos.

- De hecho, puede constar de más pero estos dos son los principales.
- Estos dos artefactos son:
  - **El diagrama de interacción.**
  - **El diagrama de clases de diseño.**

Diagramas de interacción

- Diagramas que sirven para indicar cómo interactúan los diferentes objetos de una aplicación para conseguir un objetivo determinado.
- Por ejemplo, un diagrama de interacción podría mostrar todos los objetos de la aplicación que colaboran para otorgar un crédito a un cliente.
- Los diagramas de interacción son dinámicos se centran en el comportamiento de los objetos (mensajes) más que en la estructura de las clases (atributos y métodos)

Hay dos versiones de diagramas de interacción

- Diagramas de secuencia.
- Diagramas de colaboración.
- Los dos tienen exactamente la misma información.
- De hecho, hay herramientas CASE que convierten a un tipo de diagrama en otro con solo seleccionar una opción de la herramienta.
- Entonces, ¿en qué se diferencian?

La diferencia está en qué información es más visible

- Los diagramas de secuencia y de colaboración **tienen la misma información.**
- Sin embargo, los diagramas de secuencia hacen más visible el orden temporal de los mensajes. El énfasis está en el tiempo.
- Los diagramas de colaboración hacen más visible la información de cómo colaboran los objetos para conseguir un objetivo de la aplicación. El énfasis está en la colaboración.

Diagrama de secuencia

- Es parecido a los diagramas de secuencia del sistema que hemos visto, pero más elaborados.
- De hecho, el diagrama de secuencia del sistema no es más que un caso particular de diagrama de secuencia.
- Un diagrama de secuencia está compuesto por:
  - Una serie de **objetos**.
  - Que se relacionan entre sí enviándose **mensajes**.

Un objeto en un diagrama de secuencia

- Se representa con un cuadro.
- Dentro del cuadro está el nombre del objeto y separado por dos puntos (:) el nombre de la clase.
- Debajo de él, hay una línea vertical discontinua que refleja el paso del tiempo.

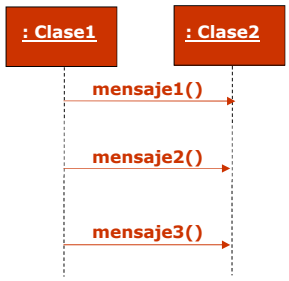
Si no nos interesa el nombre del objeto

- Podemos indicar solamente el nombre de la clase precedido por dos puntos.

Mail: palasi@palasi.com Web: www.palasi.com

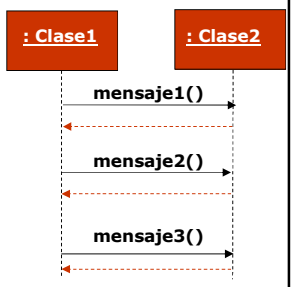
Un diagrama de secuencia está compuesto por objetos y mensajes

- Un mensaje entre dos objetos se indica como una flecha entre sus líneas verticales.
- La flecha está etiquetada con el nombre del mensaje.
- Los mensajes que aparecen arriba se ejecutan antes que



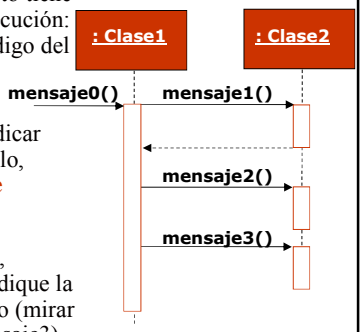
Puede indicarse cuando devuelve el control al primer objeto

- Cuando la ejecución del mensaje acaba, el segundo objeto retorna el control al primero.
- Algunos diseñadores lo indican con una flecha de retorno discontinua, aunque no es común.
- También se puede etiquetar esta flecha de retorno con el resultado del mensaje.



El tiempo entre el mensaje y el retorno

- El segundo objeto tiene el control de ejecución: se ejecuta el código del segundo objeto.
- Esto se pude indicar con un rectángulo, llamado **caja de activación.**
- Se suele indicar, aunque no se indique la flecha de retorno (mirar mensaje2 y mensaje3).



La etiqueta que lleva el mensaje

- Si el mensaje no devuelve resultado:

```
mensaje(param1:tipo1,..,paramn:tipon)
```

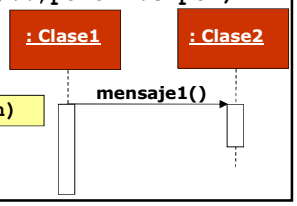
- Si el mensaje devuelve resultado:

```
resultado:=
  mensaje(param1:tipo1,..,paramn:tipon)
```

Sin embargo, los tipos se suelen omitir si no son importantes

```
mensaje(param1,..,paramn)
```

Esto es estándar en UML



Nota: aunque todas las sintaxis son válidas

- Si el mensaje devuelve resultado:

```
resultado:=
  mensaje(param1:tipo1,..,paramn:tipon)
```

- Si el mensaje no devuelve resultado pero tiene parámetros:

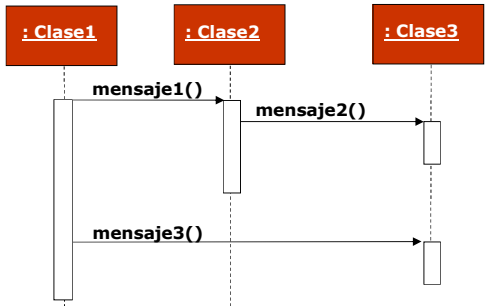
```
mensaje(param1:tipo1,..,paramn:tipon)
mensaje(param1,..,paramn)
```

- Si el mensaje no devuelve resultado ni tiene parámetros:

```
mensaje()
```

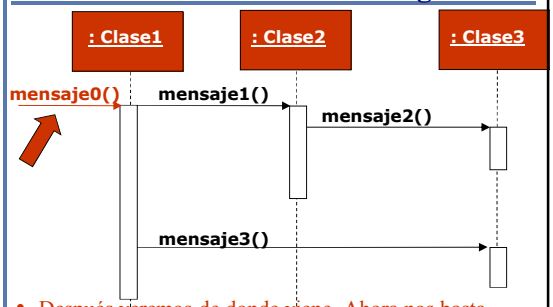
- En los ejemplos que veremos sólo usaremos esta última sintaxis (es decir supondremos que los mensajes no tienen parámetros ni resultados).
- Está es por brevedad, dado el poco espacio

Los diagramas suelen tener más de dos objetos



- Aquí sólo aparecen tres, por problemas de espacio.

Diagrama comienza con 1 mensaje que viene "de ningún sitio"



- Después veremos de donde viene. Ahora nos basta saber qué es el mensaje inicial.
- Sí se incluía en el diagrama de secuencia del sistema

A veces, un mensaje no llega a un objeto

- Llega a una lista de objetos, una colección de objetos, un array de objetos, etc.
- A esto se le llama en UML **multiobjeto** y se le representa así

- No quiere decir "un objeto de Clase1" sino "una **colección de objetos de Clase1**".

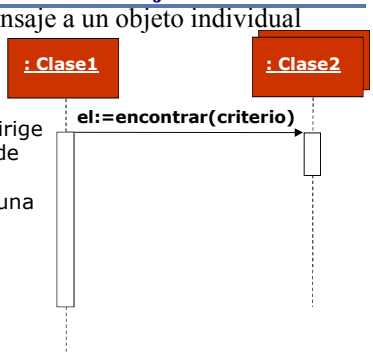
Los multiobjetos son colecciones de objetos

- A la hora de la programación se implementan como listas, arrays, colas, etc.
- A la hora del diseño son simplemente multiobjetos.

- Un multiobjeto tiene una serie de operaciones predefinidas: encontrar, añadir, eliminar, siguiente, tamaño, contiene.

Un mensaje a un multiobjeto es un mensaje a la colección

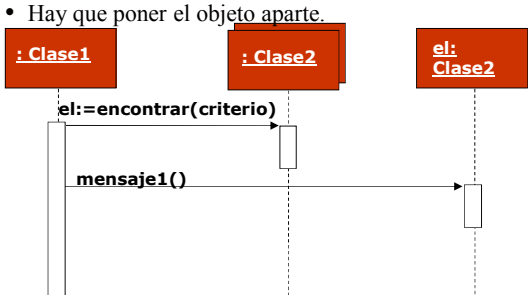
- No es un mensaje a un objeto individual



El mensaje encontrar se dirige a la colección de instancias de Clase2 y no a una instancia individual.

¿Cómo enviar un mensaje a un objeto dentro de un multiobjeto?

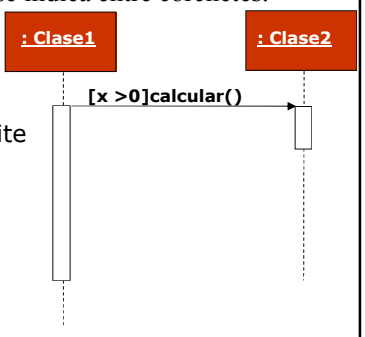
- Hay que poner el objeto aparte.



Primero se encuentra el objeto "el" enviando un mensaje a la colección. Después se envía el mensaje al objeto "el".

Sintaxis de un mensaje condicional

- La condición se indica entre corchetes.



Si x es mayor que 0, se emite el mensaje "calcular()".

Si no, no se hace nada.

Mail: palasi@palasi.com Web: www.palasi.com

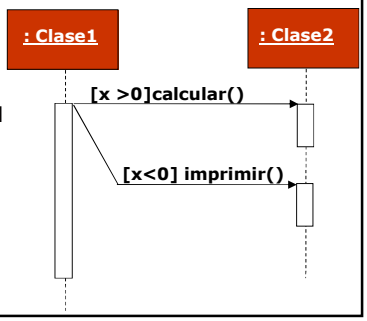
Mensajes condicionales mutuamente exclusivos

- Se indican con una línea con ángulo oblicuo que salga del primer punto.

Si x es mayor que 0, se emite el mensaje "calcular()".

Si no, se emite el mensaje imprimir().

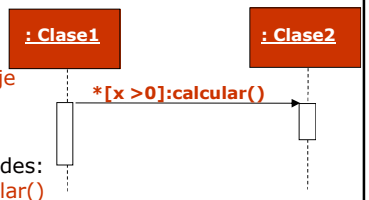
Las condiciones se excluyen mutuamente (no se cumplen a la vez).



Iteración: mensajes que se repiten

- Si un mensaje se repite varias veces, se indica de esta forma.

Esto se lee, mientras x>0, emitir el mensaje "calcular" (sería un WHILE).



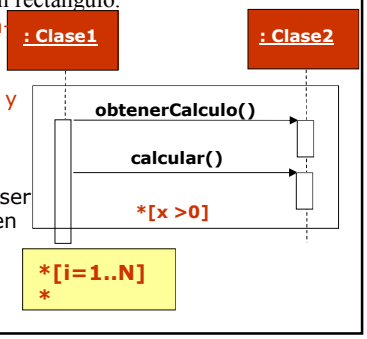
Otras posibilidades:
*[i=1..N]:calcular()
Emitir el mensaje calcular desde i igual a 1 hasta N (sería un FOR).
*:calcular()
Si no nos interesa indicar la condición de repetición, lo podemos expresar así.

Iteración: grupos de mensajes que se repiten

- Igual que antes, pero ahora los mensajes se encierran en un rectángulo.

Esto se lee, mientras x>0, repetir los mensajes "obtenerCalculo" y "calcular"

Como antes, la condición puede ser de tipo FOR o bien sólo escribir un asterisco (si no interesa la condición).

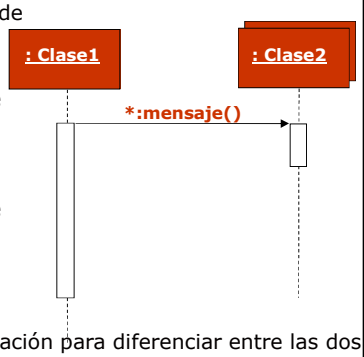


Iteración: mensajes que se repiten en un multiobjeto

El diagrama puede interpretarse de dos maneras:

1. El mensaje se envía muchas veces a la colección.

2. El mensaje se repite para cada uno de los elementos del multiobjeto.

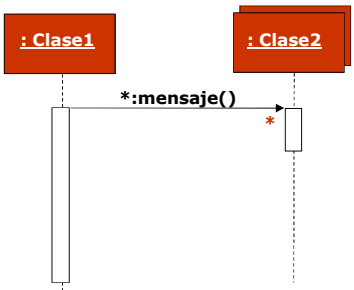


UML no tiene notación para diferenciar entre las dos.

Iteración: mensajes que se repiten en un multiobjeto

Sin embargo, se puede utilizar un asterisco como el que está en rojo para indicar

que el mensaje se repite para cada uno de los elementos del multiobjeto.



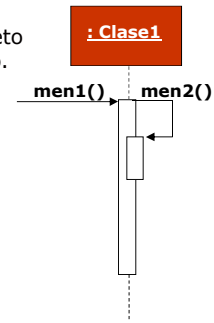
Esto no es estándar UML: es prestado de los diagramas de colaboración.

Mensajes que se envían al mismo objeto

En programación orientada a objetos, es común que un objeto mande un mensaje a él mismo.

Esto se representa así.

Como se ve, el objeto se envía el mensaje "men2()" a él mismo.

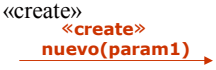


Mail: palasi@palasi.com Web: www.palasi.com

Creación de objetos

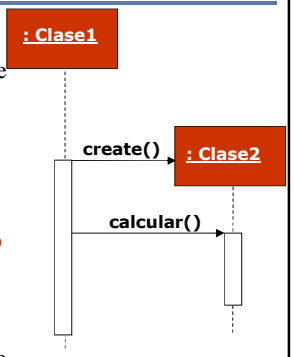
- Cualquier mensaje puede usarse para crear un objeto, pero la convención de UML es que el mensaje que crea un objeto se llama "create"

- El mensaje puede tener parámetros de inicialización y en algunos lenguajes se le llama constructor.
- Si se usa otro nombre menos obvio, siempre se puede indicar que es un mensaje de creación con el estereotipo «create»



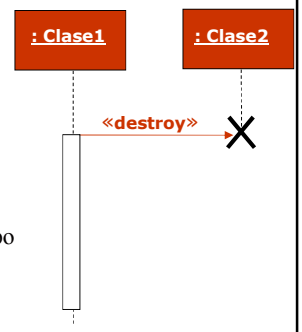
Creación de objetos

- Fíjense que el objeto que se acaba de crear se coloca a la altura del momento de su creación.
- El objeto es el que recibe el mensaje create(). Esto podría no ser obvio.
- Una vez creados (pero no antes) se les puede enviar mensajes como a



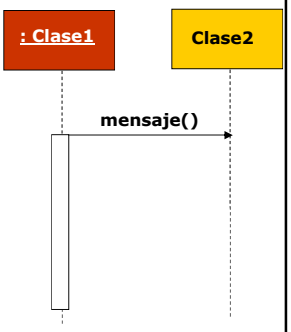
Destrucción de objetos

- **Algunos** lenguajes orientados a objetos (como C++) no tienen recolector de basura y los objetos deben destruirse explícitamente.
- Esto se indica con un mensaje con estereotipo «destroy» y una X que acaba con la línea vertical del objeto, indicando que ese



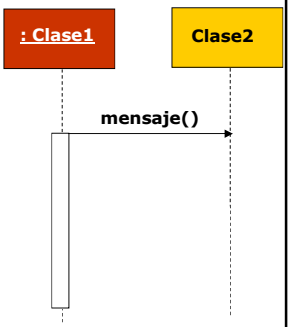
Mensajes a las clases

- Algunos lenguajes (Java, VB. NET) permiten que las clases tengan métodos asociados a ellas, en vez de asociados a objetos.
- Estos métodos se llaman "estáticos" o de clase e invocar estos métodos significa enviar un mensaje a la clase.

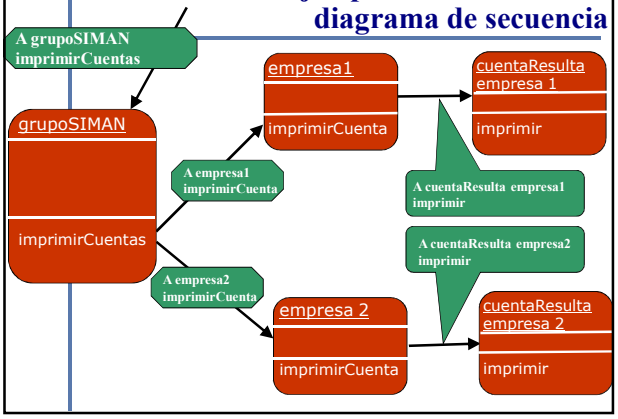


Mensajes a las clases

- Fíjense que a la derecha no hay un objeto sino una clase.
- Lo sabemos porque las clases no van subrayadas (y no tienen dos puntos).
- El mensaje se envía a la clase y no a un objeto.



Ejemplo: transformar en diagrama de secuencia

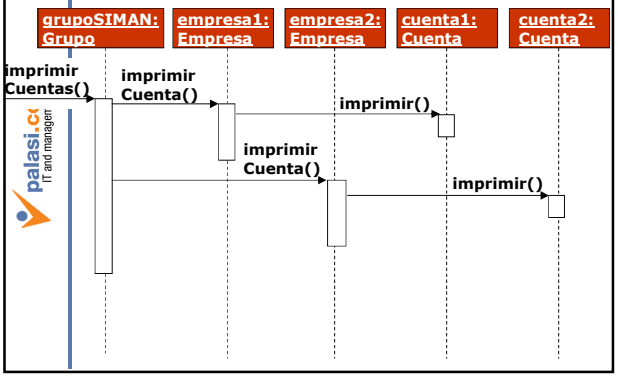


Mail: palasi@palasi.com Web: www.palasi.com

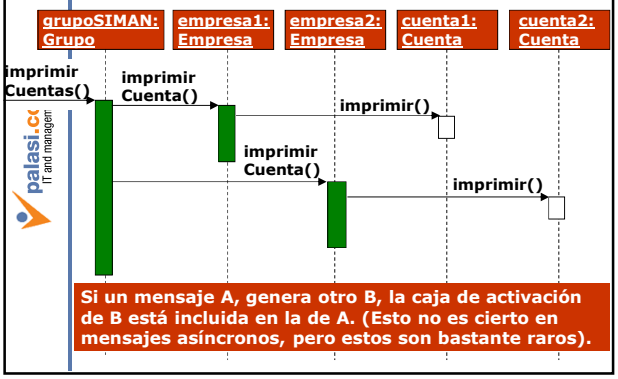
Algunas observaciones

- En este diagrama no especificábamos cuáles eran las clases de los objetos.
- Supondremos que
  - "grupoSIMAN" es de clase "Grupo",
  - "empresa1", "empresa2" son de clase Empresa.
  - "cuentaResultaEmpresa1" y "cuentaResultaEmpresa2" son de clase Cuenta.
- Por falta de espacio, "cuentaResultaEmpresa1" y "cuentaResultaEmpresa2 las abreviaremos como "cuenta1" y "cuenta2"

En diagrama de secuencia

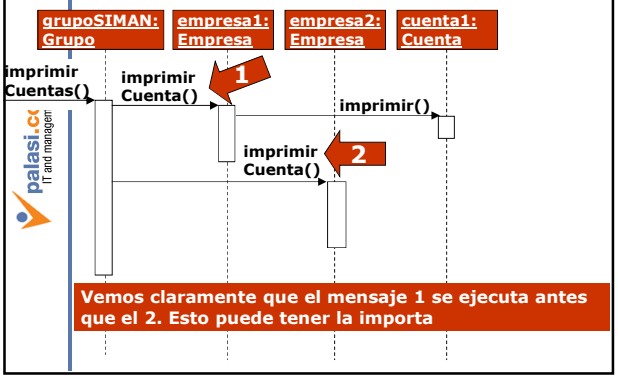


Algunas lecciones



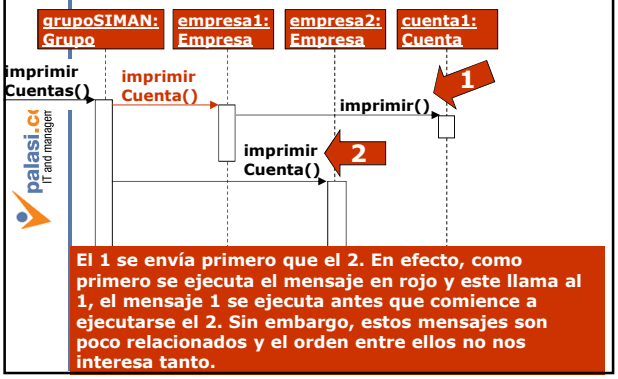
Si un mensaje A, genera otro B, la caja de activación de B está incluida en la de A. (Esto no es cierto en mensajes asíncronos, pero estos son bastante raros).

¿Cuál de los dos mensajes señalados se envía primero?



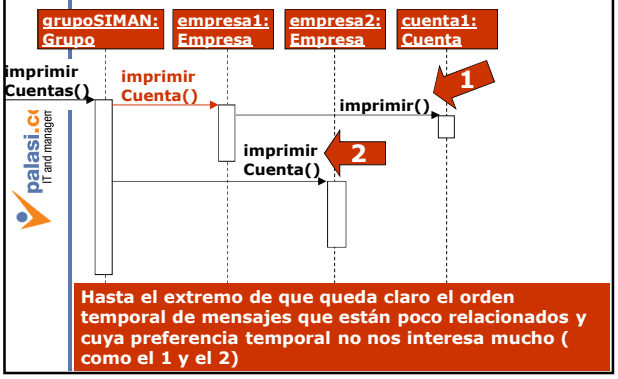
Vemos claramente que el mensaje 1 se ejecuta antes que el 2. Esto puede tener la importa

¿Cuál de los dos mensajes señalados se envía primero?



El 1 se envía primero que el 2. En efecto, como primero se ejecuta el mensaje en rojo y este llama al 1, el mensaje 1 se ejecuta antes que comience a ejecutarse el 2. Sin embargo, estos mensajes son poco relacionados y el orden entre ellos no nos interesa tanto.

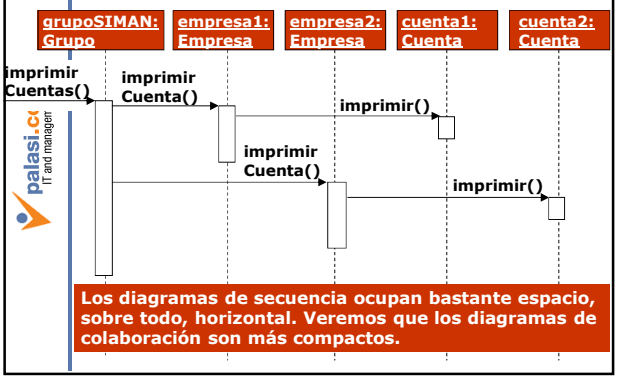
El diagrama de secuencia enfatiza el orden temporal



Hasta el extremo de que queda claro el orden temporal de mensajes que están poco relacionados y cuya preferencia temporal no nos interesa mucho (como el 1 y el 2)

Mail: palasi@palasi.com Web: www.palasi.com

Una lección final



Los diagramas de secuencia ocupan bastante espacio, sobre todo, horizontal. Veremos que los diagramas de colaboración son más compactos.

Ejercicio

- Transformen la solución del ejercicio que han hecho sobre la coordinación de un examen en un diagrama de secuencia.

3. Diseño orientado a objetos con UML

- 3.1. Introducción al diseño orientado a objetos.
- 3.2. Diagramas de secuencia.
- 3.3. Diagramas de colaboración.
- 3.4. Patrones para asignar responsabilidades.
- 3.5. Método para obtener los diagramas de interacción.
- 3.6. Visibilidad.
- 3.7. Diagramas de clases de diseño.

Diagramas de colaboración

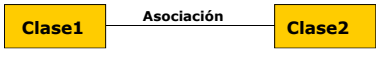
- Son otro tipo de diagramas de interacción.
- Tal como hemos dicho, la información es la misma que los diagramas de secuencia pero presentada de otra manera.
- Los dos diagramas se agrupan bajo el mismo nombre de diagramas de interacción.
- La elección de cuál de los dos tipos de diagrama escoger pertenece al diseñador, según los siguientes criterios.
  - Preferencias personales.
  - Qué información se desea resaltar.
  - Qué herramienta CASE se desea utilizar.

Diagramas de colaboración

- Recordemos: un diagrama de secuencia está compuesto por:
  - Una serie de **objetos**.
  - Que se relacionan entre sí enviándose **mensajes**.
- Un diagrama de colaboración está compuesto por:
  - Una serie de **objetos.**
  - Que se relacionan entre sí enviándose **mensajes**.
  - Los cuales viajan a través de **enlaces.**
- ¿Qué es esto de enlaces?

Un enlace (link)

- Recordemos que las clases estaban relacionadas por asociaciones.

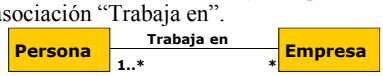


- Los objetos están relacionados por enlaces.

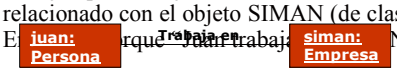
- Un enlace es una relación entre objetos.

Un ejemplo

- Entre las clases "Persona" y "Empresa" existe la asociación "Trabaja en".

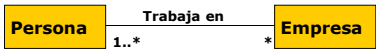


- Esto quiere decir que las instancias de "Persona" están relacionadas con las instancias de "Empresa".
- Por ejemplo, el objeto Juan (de clase Persona) está relacionado con el objeto SIMAN (de clase E[illegible]rque "Juan trabaja [illegible]N".

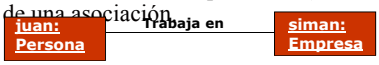


Es decir

- Igual que un objeto es una instancia (un caso particular) de una clase.



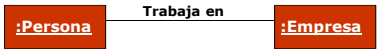
- Un enlace es un caso particular ("una instancia") de una asociación.



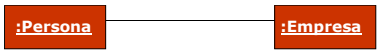
- Que el objeto Juan tenga un enlace con el objeto SIMAN es un caso particular de que la clase [illegible]

A veces, no nos interesa el nombre de las instancias

- No pondremos el nombre, pero siempre la clase:



- Esto representa el hecho de que una instancia X de Persona está relacionada con una instancia Y determinada de Empresa, pero no sabemos cómo el nombre de X o Y.



- A veces tampoco nos interesa el nombre del [illegible]

¿Qué pasa si una persona trabaja en varias empresas?

- Si una persona trabaja en varias empresas tiene un enlace con varias empresas, es decir con una colección de empresas. Esto puede indicarse así:

- Es decir, se utiliza un multiobjeto, para indicar que hay un enlaces con una colección de objetos.

Recordemos: diagramas de colaboración

- Un diagrama de colaboración está compuesto por:
  - Una serie de **objetos**.
  - Que se relacionan entre sí enviándose **mensajes**.
  - Los cuales viajan a través de **enlaces.**

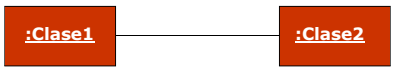
Un objeto en un diagrama de colaboración

- Se representa con un cuadro.
- Dentro del cuadro está el nombre del objeto y separado por dos puntos (:) el nombre de la clase.
- La mayoría de veces no nos interesa el nombre del objeto y sólo ponemos el nombre de la clase.

o bien

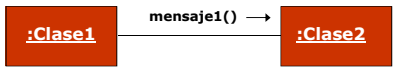
Un enlace en un diagrama de colaboración

- Se representa con una línea entre objetos.
- No se suele indicar ningún nombre en la línea.



Un mensaje en un diagrama de colaboración

- Para que haya un mensaje entre dos objetos debe haber un enlace entre los mismos.



- **El mensaje se coloca al lado de la línea del enlace con una flecha que indica la dirección del mensaje.**
- El mensaje tiene la notación estándar UML que ya hemos visto.

```
resultado:=
  mensaje(param1:tipo1,..,paramn:tip
```

Recordemos: aunque todas las sintaxis son válidas

- Si el mensaje devuelve resultado:

```
resultado:=
  mensaje(param1:tipo1,..,paramn:tipon)
```

- Si el mensaje no devuelve resultado pero tiene parámetros:

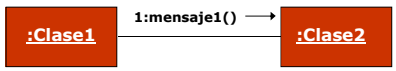
```
mensaje(param1:tipo1,..,paramn:tipon)
mensaje(param1,..,paramn)
```

- Si el mensaje no devuelve resultado ni tiene parámetros:

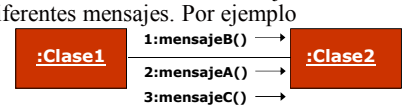
```
mensaje()
```

- En los ejemplos que veremos sólo usaremos esta última sintaxis (es decir supondremos que los mensajes no tienen parámetros ni resultados).
- Está es por brevedad, dado el poco espacio.

El mensaje suele llevar un número

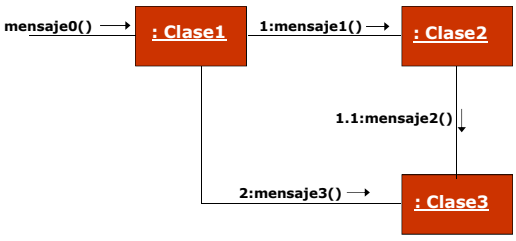


- El número precede el texto del mensaje y está separado por dos puntos (:).
- El número indica el orden de ejecución de los diferentes mensajes. Por ejemplo



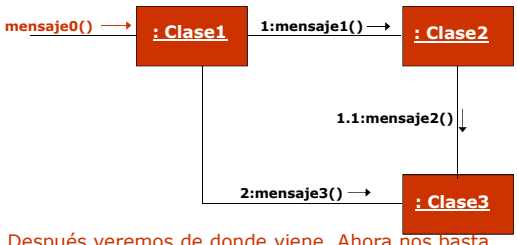
- En este caso, primero se enviaría mensajeB, luego se enviaría mensajeA, luego mensajeC.

Los diagramas suelen tener más de dos objetos



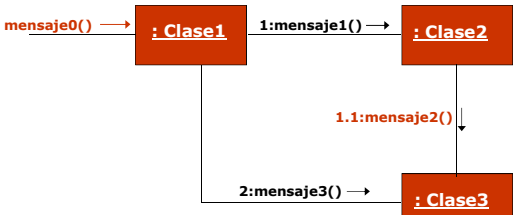
- Aquí sólo aparecen tres, por problemas de espacio.

Diagrama comienza con 1 mensaje que viene "de ningún sitio"



- Después veremos de donde viene. Ahora nos basta saber qué es el mensaje inicial.
- Estaba en el análisis (diagrama de secuencia del sistema), pero aquí no.

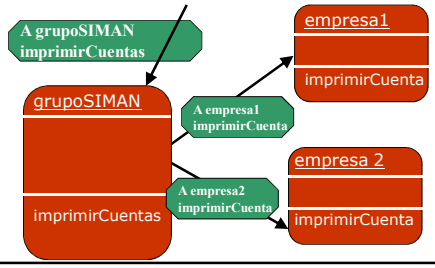
La numeración de los mensajes no es tan sencilla como pensamos



- El mensaje inicial no tiene número.
- El mensaje2 tiene un número complicado (**1.1**)
- ¿Cómo se asignan los números?

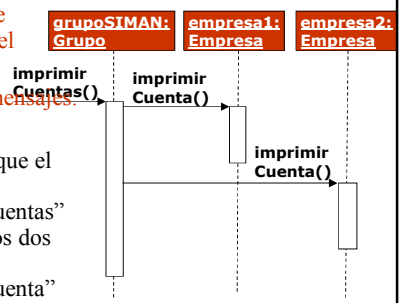
Mensajes que producen otros mensajes

- Si para ejecutar un mensaje, el objeto emite otra serie de mensajes
- Decimos que un mensaje produce otros.

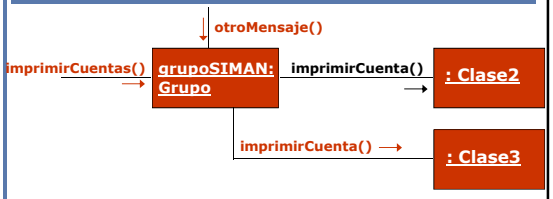


Esto se indica en el diagrama de secuencia de la siguiente manera

- De la caja de activación del mensaje, salen otros mensajes.
- Esto indica que el mensaje "imprimirCuentas" produce otros dos mensajes "imprimirCuenta" para ejecutarse.



¿Cómo se indica en un diagrama de colaboración?

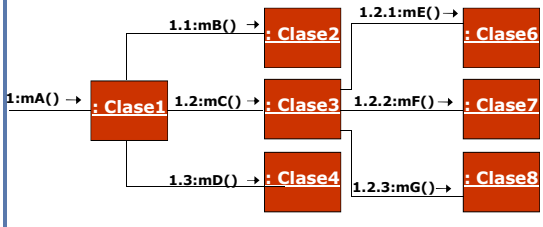


- ¿Cómo sé que el mensaje en rojo es producido por el mensaje imprimirCuentas y no por otroMensaje()?
- Esto es importante para entender este diagrama.
- Esto lo indicaré con la numeración.

Regla: Si un mensaje con número n produce otros

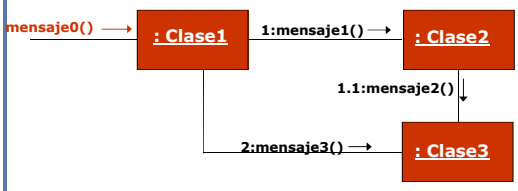
- Los mensajes producidos por el mensaje n se numerarán como
  - *n*.1
  - *n*.2
  - y así sucesivamente
- Así, si un mensaje numerado 1 produce otros estos serán: 1.1, 1.2, etc.
- Si un mensaje numerado 1.2 produce otros, estos serán 1.2.1, 1.2.2, 1.2.3, etc.

Así por ejemplo



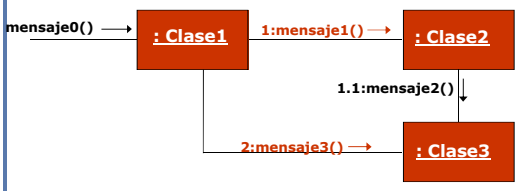
- El mensaje 1 produce los mensajes 1.1, 1.2 y 1.3.
- El mensaje 1.2 produce los mensajes 1.2.1, 1.2.2 y 1.2.3.

Diagrama comienza con 1 mensaje que viene "de ningún sitio"



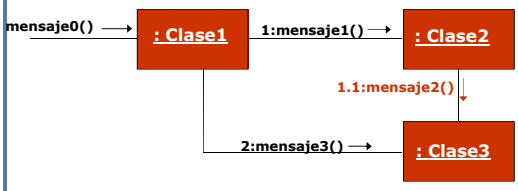
- Después veremos de donde viene. Ahora nos basta saber qué es el mensaje inicial.
- Este mensaje inicial nunca tiene número.

El mensaje inicial produce que se envíen nuevos mensajes



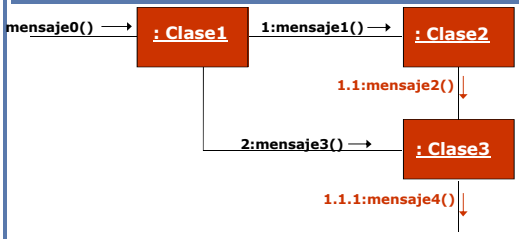
- Estos mensajes producidos por el mensaje inicial, se numeran con un número 1,2,3 (que indica el orden en que se emiten).

Estos mensajes producen otros mensajes y así sucesivamente



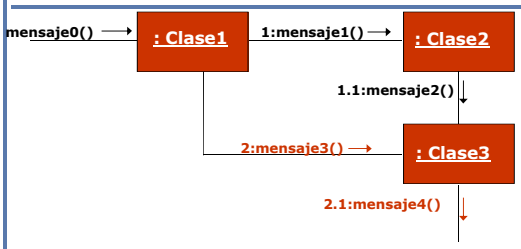
- El mensaje1 produce el mensaje2. Por eso, el mensaje2 tiene el número 1.1.

Veamos más allá



- Si el mensaje4 tiene una numeración de 1.1.1 quiere decir que ha sido producido por el mensaje2 (que tiene el 1.1)
- Es decir, el mensaje4 se ejecutará justo después del mensaje2.

Veamos más allá



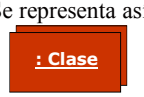
- Si el mensaje4 tiene una numeración de 2.1 quiere decir que ha sido producido por el mensaje3 (que tiene el 2)
- Es decir, el mensaje4 se ejecutará justo después del mensaje3.

Conclusión

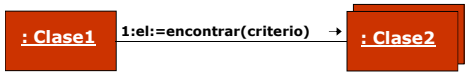
- La numeración de los mensajes es muy importante en los diagramas de colaboración.
- La numeración nos indica en qué orden se ejecutan los mensajes.
- En los diagramas de secuencia no hacían falta números, ya que el orden ya venía dado de arriba abajo.

En diagramas de colaboración también hay multiobjetos

- Recordemos que un multiobjeto es una colección de objetos. Se representa así.

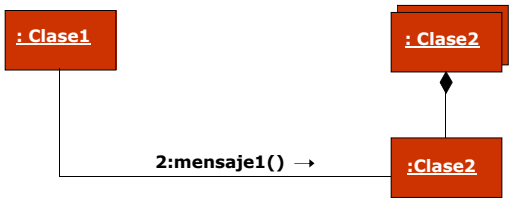


- Un mensaje a un multiobjeto es un mensaje a la colección, no a un objeto individual.



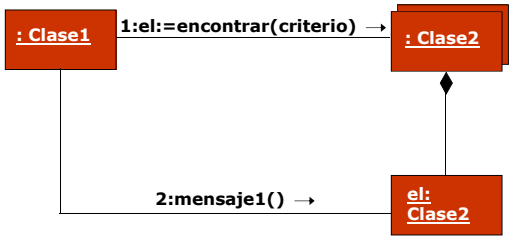
¿Cómo enviar un mensaje a un objeto dentro de un multiobjeto?

- Hay que poner el objeto aparte.
- Se puede poner el símbolo de agregación para indicar que el objeto está dentro del multiobjeto, **aunque esto es poco común en diagramas complicados**.



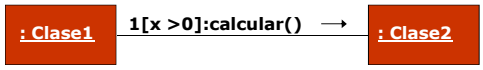
Normalmente

- Se busca el elemento antes de enviarle un mensaje.
- Fíjense que "el" es resultado de encontrar y también es el nombre del objeto.



Sintaxis de un mensaje condicional

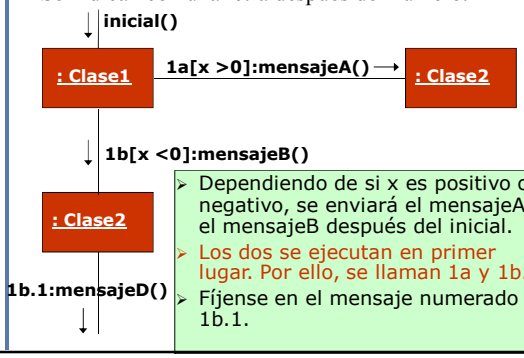
- La condición se indica entre corchetes antes de los dos puntos.



- Como ve, es bastante parecida a la sintaxis de un mensaje condicional en los diagramas de secuencia.

Mensajes condicionales mutuamente exclusivos

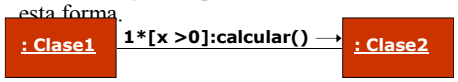
- Se indican con una letra después del número.



- Dependiendo de si x es positivo o negativo, se enviará el mensajeA o el mensajeB después del inicial.
- Los dos se ejecutan en primer lugar. Por ello, se llaman 1a y 1b.
- Fíjense en el mensaje numerado 1b.1.

Iteración: mensajes que se repiten

- Si un mensaje se repite varias veces, se indica de esta forma.



Esto se lee, mientras x>0, emitir el mensaje "calcular" (sería un WHILE).

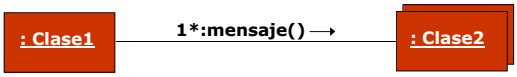
Otras posibilidades:
1*[i=1..N]:calcular()
Emitir el mensaje calcular desde i igual a 1 hasta N (sería un FOR).

1*:calcular()
Si no nos interesa indicar la condición de repetición, lo podemos expresar así.

palasi.com
IT and management

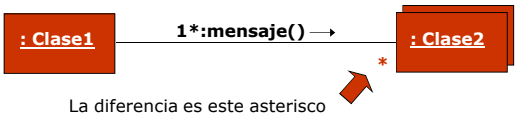
Iteración: mensajes que se repiten en un multiobjeto

Este diagrama significa que un mensaje se envía repetidamente (asterisco) a la colección.



Esto es lógico, pues hemos dicho que los mensajes de multiobjeto se envían a la colección.

Para indicar que 1 mismo mensaje se envía a cada uno de los objetos individuales del multiobjeto.



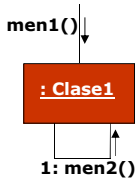
La diferencia es este asterisco

Mensajes que se envían al mismo objeto

En programación orientada a objetos, es común que un objeto mande un mensaje a él mismo.

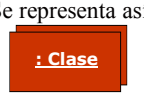
Esto se representa con un enlace del objeto a si mismo y con el mensaje en ese enlace.

Como se ve, el objeto se envía el mensaje "men2()" a él mismo.



Creación y destrucción de objetos

- Recordemos que un multiobjeto es una colección de objetos. Se representa así.

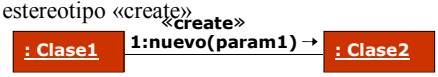


- Un mensaje a un multiobjeto es un mensaje a la colección, no a un objeto individual.

Creación de objetos

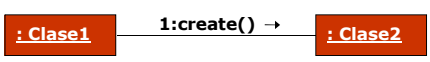
- Cualquier mensaje puede usarse para crear un objeto, pero la convención de UML es que el mensaje que crea un objeto se llama "create"

- El mensaje puede tener parámetros de inicialización y en algunos lenguajes se le llama constructor.
- Si se usa otro nombre menos obvio, siempre se puede indicar que es un mensaje de creación con el estereotipo «create»



Nota

- El objeto que se crea es el que recibe el método "create". Esto podría no parecer obvio para algunas personas pero la notación es así.

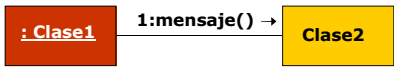


Destrucción de objetos

- **Algunos** lenguajes orientados a objetos (como C++) no tienen recolector de basura y los objetos deben destruirse explícitamente.
- Esto se indica con un mensaje con estereotipo «destroy».

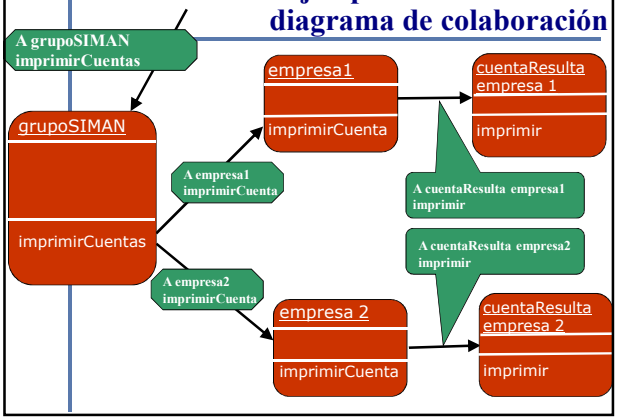
Mail: palasi@palasi.com Web: www.palasi.com

Mensajes a las clases



- Algunos lenguajes (Java, VB. NET) permiten métodos estáticos que son métodos asociados a las clases.
- Estos métodos se llaman "estáticos" o de clase e invocar estos métodos significa enviar un mensaje a la clase.
- Esto se indica en UML de esta forma. A la izquierda hay una clase, pues no está subrayada con los objetos.

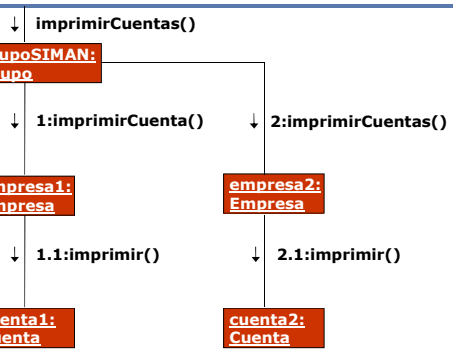
Ejemplo: transformar en diagrama de colaboración



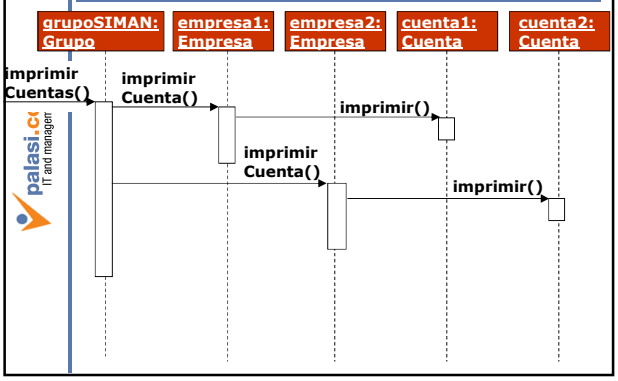
Algunas observaciones

- En este diagrama no especificábamos cuáles eran las clases de los objetos.
- Supondremos que
  - "grupoSIMAN" es de clase "Grupo",
  - "empresa1", "empresa2" son de clase Empresa.
  - "cuentaResultaEmpresa1" y "cuentaResultaEmpresa2" son de clase Cuenta.
- Por falta de espacio, "cuentaResultaEmpresa1" y "cuentaResultaEmpresa2 las abreviaremos como "cuenta1" y "cuenta2"

En diagrama de colaboración



Comparémoslo con el diagrama de secuencia



Comparación entre los diagramas

- El diagrama de colaboración es más compacto.
- El diagrama de secuencia necesita más espacio horizontal.
- El diagrama de secuencia deja más claro el orden temporal.
- El diagrama de colaboración deja más claro los enlaces.

Mail: palasi@palasi.com Web: www.palasi.com

Ejercicio

- Transformen la solución del ejercicio que han hecho sobre la coordinación de un examen en un diagrama de colaboración.

3. Diseño orientado a objetos con UML

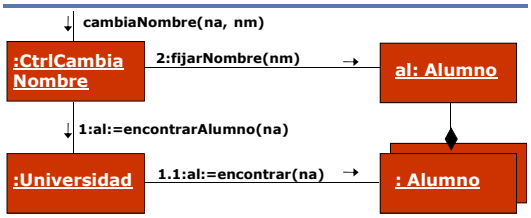
De dónde venimos y adónde vamos

- Ahora que hemos visto la notación de los diagramas de interacción, nos toca explicar un método que nos permite diseñarlos a partir del análisis.
- Sin embargo, este método está basado en el concepto de "patrones de diseño".
- Para poder entender el método, debemos comprender primero qué son los patrones de diseño.
- Los patrones de diseño sirven para asignar

Responsabilidad

- En lo que a nosotros respecta, definimos responsabilidad de la siguiente manera:
- Un objeto (o clase) tiene la responsabilidad de realizar una tarea si es **el que se encarga de ejecutar esta tarea.**
- Como vemos, esto es muy parecido a lo que significa responsabilidad en la vida real.
- Más específicamente, un objeto (o clase) tiene la **responsabilidad de realizar una tarea si tiene el código (el método)** para realizar esta tarea

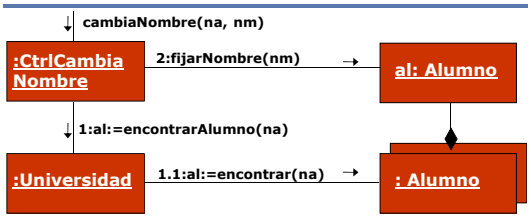
Ejemplo



La responsabilidad de la tarea de cambiar el nombre de un alumno es del objeto que hay en la esquina superior izquierda.

Es este objeto el que se encarga de llevar a cabo esta tarea y el que tiene un método para ejecutarla.

Ejemplo



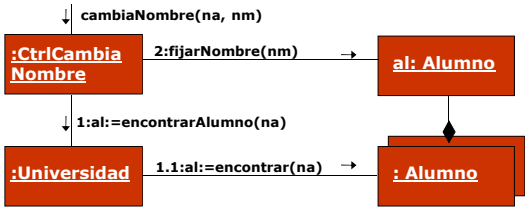
La responsabilidad de la tarea de encontrar un alumno es del objeto de la clase Universidad.

Es este objeto el que se encarga de llevar a cabo esta tarea y el que tiene un método para ejecutarla.

Lo más difícil en un diagrama de interacción

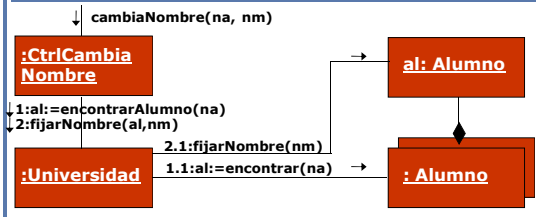
- **Lo más importante** (y lo más difícil) para diseñar diagramas de interacción es **asignar responsabilidades a los objetos.**
- Es decir, para cada tarea, decidir el objeto que la ejecutará (el objeto que será responsable de ella).
- Esto es más difícil de lo que parece a simple vista pues hay muchas opciones posibles.

Ejemplo de varias opciones



- ¿De quien es la responsabilidad de fijar el nombre de un alumno? Según aparece aquí, la responsabilidad es del objeto Alumno. Esto parece lógico y natural.

Sin embargo,



- Podíamos tener una asignación de responsabilidades así. El responsable de fijar el nombre de un alumno es la Universidad que, a su vez, delega en el alumno.
- Como ven, incluso en un ejemplo sencillo, puede haber dudas

¿Cómo hacemos para asignar responsabilidades a los objetos?

- Como hemos visto, incluso en un ejemplo sencillo pueden haber dudas. En un ejemplo normal puede haber multitud de opciones.
- ¿Cómo seleccionar la mejor opción?
- Es más, ¿qué entendemos como mejor opción?

La mejor opción

- La mejor opción es la que sea más verdaderamente orientada a objetos y la que sea más flexible.
- Dicho de otra manera, **la mejor opción es la que tiene una alta cohesión, un bajo acoplamiento y variaciones protegidas.**

Cohesión

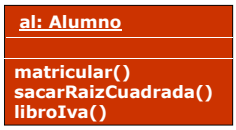
- Una clase o objeto con alta cohesión es aquella que todas sus responsabilidades (sus tareas) están muy relacionadas entre sí.

- Una clase o objeto con baja cohesión es aquella que tiene responsabilidades (tareas) muy poco relacionadas.

- **Es preferible la alta cohesión**

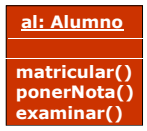
Es preferible la alta cohesión

- Una clase con baja cohesión es:
- Difícil de entender.
- Difícil de reutilizar.
- Difícil de mantener.
- Delicada. Constantemente afectada por el cambio.
- La baja cohesión es señal de mal diseño.



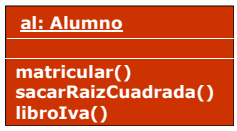
Acoplamiento

- Una clase o objeto con bajo acoplamiento es aquel que no depende ni está relacionado con otras clases.



Depende de Universidad, Examen, Aula.

- Una clase o objeto con alto acoplamiento es aquella que depende de muchas clases.



- **Es preferible el bajo acoplamiento.**

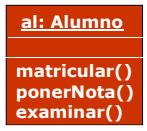
Depende de Universidad, Examen, Aula, Aritmética, Factura, Cliente, etc.

Es preferible el bajo acoplamiento

- El alto acoplamiento presenta los siguientes problemas:
  - Los cambios en las clases relacionadas producen cambios en la clase.
  - Difícil de entender por sí misma sin entender otras clases.
  - Difícil de reutilizar pues necesita la presencia de otras clases.

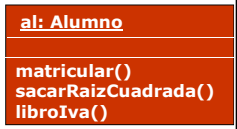
Como vemos la cohesión y el acoplamiento están relacionados

- Pero en el caso de que haya un conflicto es más importante la cohesión que el acoplamiento.



Alta cohesión y bajo acoplamiento

Baja cohesión y alto acoplamiento



Variaciones protegidas

- Es otro principio que es deseable. Produce flexibilidad del programa.
- Se trata de diseñar el programa de forma que sea fácil de modificar y mantener en el futuro.
- Por ello, a todos los puntos que preveamos que el sistema debe variar, debemos aplicar un diseño que nos permita que sea fácil modificarlo en el futuro.
- Decimos que el sistema está protegido contra las variaciones probables futuras. Abreviadamente "variaciones protegidas".

Recordemos: ¿Cómo asignar responsabilidades a los objetos?

- Como hemos visto, incluso en un ejemplo sencillo pueden haber dudas. En un ejemplo normal puede haber multitud de opciones.
- ¿Cómo seleccionar la mejor opción?
- Es más, ¿qué entendemos como mejor opción?
- **Ahora lo sabemos: la mejor opción es la que tiene alta cohesión, bajo acoplamiento y variaciones protegidas.**

Mail: palasi@palasi.com Web: www.palasi.com

Objetivo

- Nuestro objetivo es asignar responsabilidades de forma que se creen objetos (y clases) con alta cohesión, bajo acoplamiento y variaciones protegidas.

¿Cómo seleccionar la mejor opción?

- Nos queda la otra pregunta. En principio, parecería obvio. Consideramos todas las opciones y examinamos su cohesión y acoplamiento y si tienen variaciones protegidas.
- En la vida práctica, hay demasiadas opciones para considerarlas una por una.
- ¿Cómo seleccionar la mejor opción sin tener que examinar todas las opciones (lo cual no sería posible)?

¿Cómo asignar responsabilidades para obtener la mejor opción?

- ¿Cómo asignar responsabilidades para encontrar la mejor opción (alta cohesión, bajo acoplamiento y variaciones protegidas)?
- En principio, **no hay una fórmula mágica** que sirva para todos los casos.
- Lo que sí que hay es una serie de "recetas" que sirven para dar la mejor opción en cada caso particular.
- A estas "recetas", se les llama "patrones de diseño"

Patrones de diseño

- Soluciones a problemas comunes de la programación orientada a objetos. **Permiten asignar responsabilidades a las clases y objetos.**
- Cuando el programador encuentra un problema, se pregunta a que patrón de diseño pertenece y lo resuelve con la solución que tiene este patrón de diseño.
- Así las listas de patrones de diseño son como un libro de "recetas" con soluciones a problemas comunes.
- Los patrones de diseño se pueden ver desde el punto de vista de los objetos y desde el punto de vista de las clases. Lo veremos desde los 2 puntos

Nosotros veremos cinco patrones para asignar responsabilidades

- Los llamaremos por sus nombres en inglés.
- 1. "Controller"
- 2. "Information Expert"
- 3. "Creator"
- 4. "Multiobject"
- 5. "Pure Fabrication".

Nosotros veremos cinco patrones para asignar responsabilidades

- Los llamaremos por sus nombres en inglés.
- 1. "Controller"
- 2. "Information Expert"
- 3. "Creator"
- 4. "Multiobject"
- 5. "Pure Fabrication".

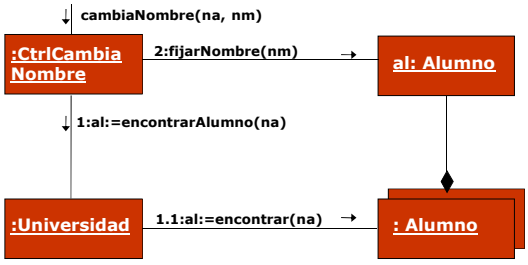
Mail: palasi@palasi.com Web: www.palasi.com

El patrón "Controller"

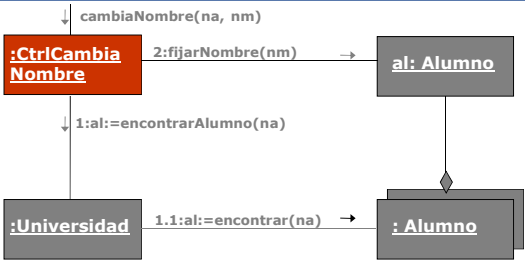
- Problema: "¿Qué objeto o clase tiene la responsabilidad de un evento del sistema?"
- Solución: "Hay un objeto, llamado controlador, que es responsable de todos los eventos del sistema.".

- Nota: Recordemos que un evento del sistema es el

Veamos un ejemplo



La clase controlador

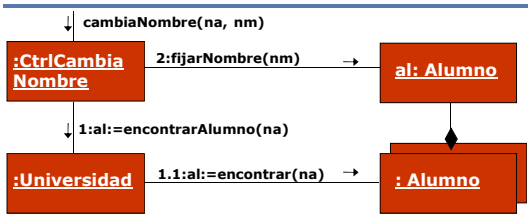


¿Qué es esta clase CtrlCambiaNombre?
No aparece en el dominio.

Es una clase controlador

- Las clases controlador son las que reciben todos los eventos del sistema y se encargan de ejecutarlos, según el patrón "Controller".
- Las clases controladores son clases artificiales (que no están en el dominio) y dirigen todas las otras clases. Son como los directores de orquesta.
- **Hay una clase controlador por cada caso de uso** (atención:no por cada diagrama).

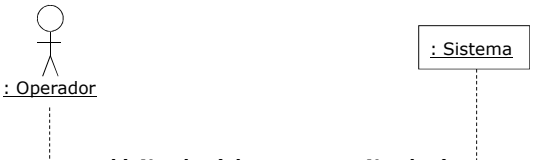
Ejemplo de patrón "Controller"



Vemos que el controlador "CtrlCambiaNombre" es el encargado de servir el evento inicial del sistema "cambiaNombre()".

Sabemos que "cambiaNombre()" es un evento inicial del sistema porque aparece en el DSS.

cambiaNombre() es un evento del sistema



Ya que aparece en el DSS. Estos eventos siempre son servidos por el controlador.

Mail: palasi@palasi.com Web: www.palasi.com

Nosotros veremos cinco patrones para asignar responsabilidades

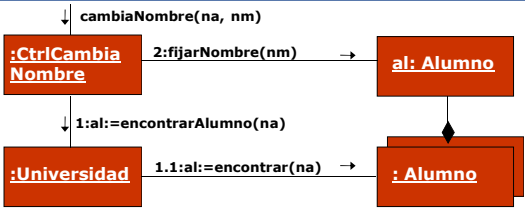
- Los llamaremos por sus nombres en inglés.

- 1. "Controller"
- 2. "Information Expert"
- 3. "Creator"
- 4. "Multiobject"
- 5. "Pure Fabrication".

El patrón "Information Expert"

- Problema: "¿Qué objeto o clase tiene la responsabilidad de un mensaje?"
- Solución: "Es el objeto que tiene la información para poder ejecutar este mensaje (para cumplir esta responsabilidad)".
- A este objeto se le llama experto de información para este mensaje.
- Este es el patrón más usado.

Ejemplo de patrón "Information Expert"



Vemos que el objeto "al:Alumno" es el responsable el mensaje "fijarNombre(nm)", que trata de fijar el nombre de un alumno.

El objeto Alumno tendrá la información sobre el nombre del alumno (normalmente, un atributo) y por tanto, es el experto de información para este mensaje.

Nosotros veremos cinco patrones para asignar responsabilidades

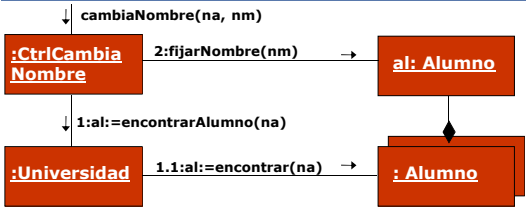
- Los llamaremos por sus nombres en inglés.

- 1. "Controller"
- 2. "Information Expert"
- 3. "Creator"
- 4. "Multiobject"
- 5. "Pure Fabrication".

El patrón "Creator"

- Problema: "¿Qué clase (o objeto) tiene la responsabilidad de crear una instancia de una clase A?"
- Solución: "Es la clase B que cumple una o varios de estas condiciones:
  - B agrega A.
  - B contiene A.
  - B tiene una colección de A.
  - B guarda instancias de A.
  - B usa muy íntimamente A.
  - B tiene los datos de inicialización necesarios para crear A (es decir, B es el experto de información de A) .
- Si hay varias clases que cumplen diferentes de estas condiciones, se elige la que cumple las tres primeras

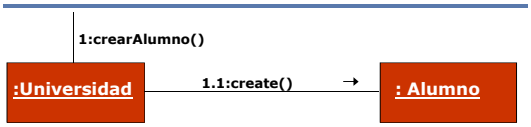
Ejemplo de patrón "Creator"



Aquí no aparecen clases, pero podemos imaginarnos cuáles son por los objetos.

Claramente, la clase Universidad es el creador de Alumno: contiene la colección de alumnos.

Ejemplo de patrón "Creator"



Supongamos que queremos crear una nueva instancia de Alumno.

¿Qué clase es el encargado de crearlo?

Como Universidad es el creador del Alumno es él el que se ocupa de crear una instancia de alumno.

Nosotros veremos cinco patrones para asignar responsabilidades

- Los llamaremos por sus nombres en inglés.

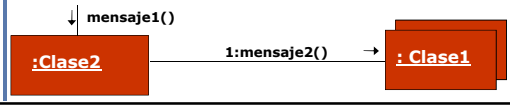
- 1. "Controller"
- 2. "Information Expert"
- 3. "Creator"
- 4. "Multiobject"
- 5. "Pure Fabrication".

Recordemos : a un multiobjeto se le pueden enviar mensajes

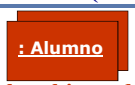
- Un multiobjeto tiene una serie de operaciones predefinidas: encontrar, añadir, eliminar, siguiente, tamaño, contiene.
- Se pueden mandar mensajes con estas operaciones, que son mensajes que se dirigen a la colección.

El patrón "Multiobject"

- Problema: "¿Qué objeto (o clase) se encarga de enviar un mensaje a un multiobjeto?"
- Solución: "Depende del tipo de multiobjeto".
- Distinguimos tres tipos de multiobjeto.



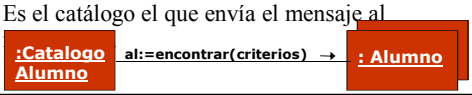
Distinguimos 3 tipos de multiobjetos (se dibujan igual)



- **1. Contiene todos los objetos de esta clase que hay en el sistema.** Por ejemplo, todos los alumnos del sistema.
- **2. Contiene todos los objetos de esta clase que cumplen una cierta condición.** Por ejemplo, todos los alumnos que viven en San Miguel.
- **3. Contiene todos los objetos de esta clase que están relacionados con otro objeto**. Por ejemplo, todos los alumnos de una cierta Facultad (donde las facultades son objetos)

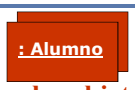
Tipo 1

- **1. Contiene todos los objetos de esta clase que hay en el sistema.** Por ejemplo, todos los alumnos del sistema.
- En este caso, hay un objeto especial que se dedica a dar acceso al multiobjeto. Podemos llamarlo "catálogo".
- Es el catálogo el que envía el mensaje al

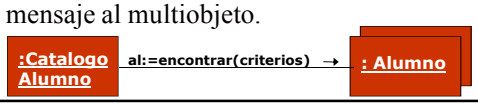


Mail: palasi@palasi.com Web: www.palasi.com

Tipo 2

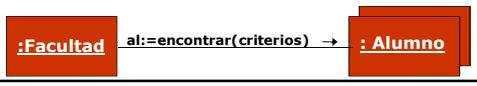


- **2. Contiene todos los objetos de esta clase que cumplen una cierta condición.** Por ejemplo, todos los alumnos que viven en San Miguel.
- En este caso, también es el catálogo (un objeto especial) el encargado de enviar el mensaje al multiobjeto.



Tipo 3

- **3. Contiene todos los objetos de esta clase que están relacionados con otro objeto**. Por ejemplo, todos los alumnos de una cierta Facultad (donde las facultades son objetos)
- Es el otro objeto el que se encarga de enviar un mensaje al multiobjeto.



Resumiendo: El patrón "Multiobject"

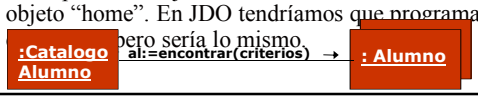
- Problema: "¿Qué objeto (o clase) se encarga de enviar un mensaje a un multiobjeto?"
- Solución: "Depende del tipo de multiobjeto.
  - Si es del tipo 1 o el tipo 2, un objeto especial llamado catálogo.
  - Si es del tipo 3, el objeto con el que está relacionado el objeto.

Nota

- Hay veces en que un multiobjeto se puede considerar al mismo tiempo de Tipo 1 y de Tipo 3.
- Por ejemplo, en un banco "las libretas de un banco", son todas las libretas del sistema y, al mismo tiempo, todas las libretas relacionadas con el objeto "Banco" (pues sólo hay uno de esos objetos).
- En ese caso, se considera que es de Tipo 1. El encargado de enviar mensajes es el catálogo.

Nota.

- En nuestro caso, como sólo estamos considerando las clases de negocio, el multiobjeto se encuentra en memoria.
- Sin embargo, si el multiobjeto se encuentra en disco (persistencia de objetos), la política es la misma.
- En este caso, el catálogo es la clase que se dedica a recuperar los objetos del disco. En EJB sería el objeto "home". En JDO tendríamos que programar pero sería lo mismo.



Nosotros veremos cinco patrones para asignar responsabilidades

- Los llamaremos por sus nombres en inglés.
- 1. "Controller"
- 2. "Information Expert"
- 3. "Creator"
- 4. "Multiobject"
- 5. "Pure Fabrication".

El patrón "Pure Fabrication"

- Problema: "En ocasiones aplicamos uno de los patrones anteriores, pero el resultado no tiene buenas propiedades:
  - Tiene baja cohesión.
  - Tiene alto acoplamiento.
  - Hay variaciones probables no protegidas.
- Solución: Crear clases y objetos que, aunque no están en el modelo de dominio (ni en la realidad), nos sirven para crear estos objetivos.

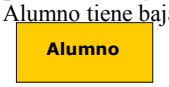
Ejemplo

- Supongamos que necesitamos grabar objetos Alumno en una base de datos.

- Según el patrón "Information Expert", quien tiene la información (y, por tanto, la responsabilidad) para grabar es el alumno.
- Por lo tanto, el objeto (y por tanto, la clase Alumno) deberá tener un método para grabarse en la base de datos.

¿Por qué la solución no es satisfactoria?

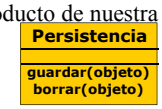
- La tarea de guardar en una base de datos requiere un número de operaciones de gestión de la base de datos. La clase Alumno tiene baja cohesión.



- La clase Alumno debe implementarse sobre una interfaz de base de datos. Esto produce un alto acoplamiento.
- Grabar en una BD es una tarea general y tendrá que ser incorporada a muchas otras clases. Esto produce mucho código redundante (no reutilización de código). Además, a una variación de la interfaz de BD, resulta difícil cambiar todo el

Una mejor solución

- Creamos una nueva clase artificial que se dedica a guardar objetos en una base de datos. Esta clase es "Pure Fabrication", no existe en el dominio (realidad): es producto de nuestra mente.



- Cualquier objeto que quiera guardarse en la base de datos lo hará con los métodos de esta clase. Así conseguimos:
  - Alta cohesión (Alumno y Persistencia).
  - Bajo acoplamiento de Alumno.
  - Variación de la interfaz de BD protegida.

Nota

- Este ejemplo ha sido escogido por ser representativo.
- En la realidad no solemos programar las clases que persisten nuestros objetos.
- Lo que hacemos es comprar un producto comercial o bien usar un producto de código abierto.

"Pure Fabrication" se usa en muchas otras formas

- Lo que hemos visto no es un ejemplo típico como con los otros patrones. "Pure Fabrication" tiene infinitas formas de aplicación.
- Es por eso que este patrón es muy general. Por ello, se han desarrollado un gran número de patrones que son casos particulares de este, para aplicar a cada situación específica.
- Aunque con los cinco patrones hasta ahora podemos diseñar programas, es mucho mejor aprender más patrones para poder diseñar mejor.

Este no es un curso sobre patrones de diseño

- No vamos a estudiar este tema en detalle.
- Para más información, la "biblia" de los patrones de diseño se llama "Design Patterns" de la "banda de los cuatro" ("Gang of Four"o "GoF":Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides). Presenta 23 patrones ("GoF patterns"). Esta escrita para C++ pero los patrones son aplicables a cualquier lenguaje.
- Los patrones que hemos visto aquí son patrones GRASP de Craig Larman, excepto el "Multiobject" que es de elaboración propia.

3. Diseño orientado a objetos con UML

- 3.1. Introducción al diseño orientado a objetos.
- 3.2. Diagramas de secuencia.
- 3.3. Diagramas de colaboración.
- 3.4. Patrones para asignar responsabilidades.
- 3.5. Método para obtener los diagramas de interacción.
- 3.6. Visibilidad.
- 3.7. Diagramas de clases de diseño.

De dónde venimos y adónde vamos

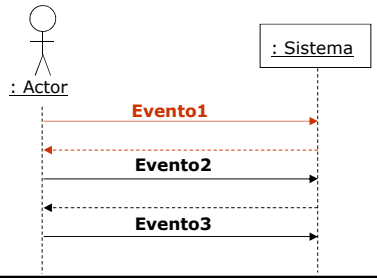
- Ahora que ya conocemos los patrones.
- Estamos en condiciones de comprender el método para obtener diagramas de interacción.

Un diagrama de interacción

- Consta de:
- 1. Una única clase controladora (patrón "Controller"), que es la que coordina a las otras para realizar el caso de uso. Esta clase es común a todo el caso de uso.
- 2. Una o varias clases de dominio, que representan conceptos de la vida real.
- 3. A veces, otras clases artificiales (patrón "Pure fabrication") que sirven para dar más flexibilidad al diseño resultante.

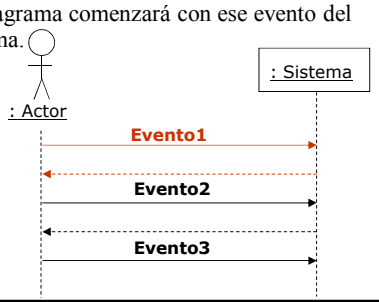
Los diagramas de interacción se crean a partir de los DSS

- Recordemos que los DSS están compuestos por una serie de eventos, llamados eventos del sistema.



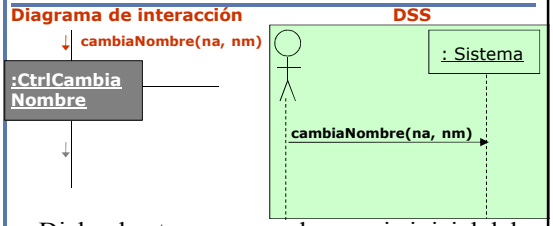
¿Cuántos diagramas de interacción creamos?

- Para cada evento que aparezca en el DSS haremos un diagrama de interacción diferente.
- El diagrama comenzará con ese evento del sistema.



Mail: palasi@palasi.com Web: www.palasi.com

El diagrama de interacción comienza con 1 evento del sistema



- Dicho de otra manera, el mensaje inicial del diagrama de interacción será **el evento del sistema** del DSS.

Método para obtener los diagramas de interacción

- 1. Identificamos los escenarios "importantes" del sistema.
- 2. A partir del DSS de cada uno de estos escenarios, obtenemos los eventos del sistema.
- 3. Cada evento del sistema de ese escenario lo transformamos en un diagrama de interacción (similar a los vistos en las realizaciones).

Método para obtener los diagramas de interacción

- 1. Identificamos los escenarios "importantes" del sistema.

Los escenarios "importantes" son los que nos interesa modelar. Normalmente, son los escenarios principales de cada caso de uso y algún escenario alternativo interesante.

- 3. Cada evento del sistema de ese escenario lo transformamos en un diagrama de interacción (similar a los vistos en las realizaciones).

Método para obtener los diagramas de interacción

- 1. Identificamos los escenarios "importantes" del sistema.

Los escenarios "importantes" no los habremos decidido aquí sino cuando hayamos diseñado los DSS.

Es decir, los escenarios importantes serán aquellos que tengan un DSS. *Sencillo.*

Por supuesto todo esto está sujeto a revisión iterativa

interacción (similar a los vistos en las realizaciones)..

Método para obtener los diagramas de interacción

- 1. Identificamos los escenarios "importantes" del sistema.
- 2. A partir del DSS de cada uno de estos escenarios, obtenemos los eventos del sistema.

Esto es sencillo, simplemente revisamos los DSS y obtenemos los diferentes eventos del sistema.

interacción (similar a los vistos en las realizaciones)..

Método para obtener los diagramas de interacción

- 1. Identificamos los escenarios "importantes" del sistema.

Esto es el paso complicado. Lo veremos con más detalle más adelante.

- 3. Cada evento del sistema de ese escenario lo transformamos en un diagrama de interacción (similar a los vistos en las realizaciones).

Aplicar método a ejemplo: esc. principal de "Deposita Libreta"

- 1. Identificamos los escenarios "importantes" del sistema.
- 2. A partir del DSS de cada uno de estos escenarios, obtenemos los eventos del sistema.
- 3. Cada evento del sistema de ese escenario lo transformamos en un diagrama de interacción (similar a los vistos en las realizaciones).

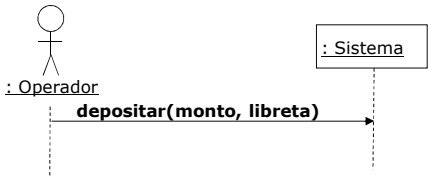
Ejemplo: Escenario principal del caso de uso "Deposita Libreta"

Nombre: Deposita Libreta.
Escenario principal:
1. El Operador selecciona la opción de depositar.
2. El Sistema pide el monto del depósito y el número de libreta para depositar.
3. El Operador entra el monto y el número de libreta.
4. El Sistema deposita el monto en la libreta.

- ¿Es éste un escenario "importante", es decir, que vamos a transformar en diagramas de interacción?
- Esta pregunta se transforma en ¿Tiene este escenario un DSS?

Para saberlo deberíamos haber especificado los DSS

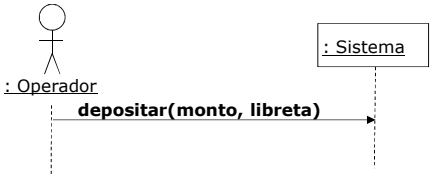
- Suponemos que sí, que cuando hemos diseñado los DSS, hemos concluido que éste es un escenario importante y, por lo tanto, hemos diseñado su DSS que es el siguiente.



Aplicar método a ejemplo: esc. principal de "Otorga Crédito"

- 1. Identificamos los escenarios "importantes" del sistema.
- 2. A partir del DSS de cada uno de estos escenarios, obtenemos los eventos del sistema.
- 3. Cada evento del sistema de ese escenario lo transformamos en un diagrama de interacción (similar a los vistos en las realizaciones).

En este caso sólo hay un evento



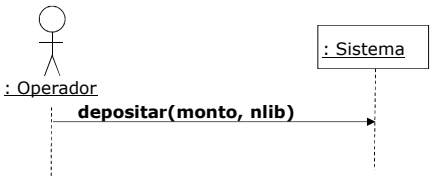
- Para enfatizar que lo que entra el operador es un número de libreta, lo llamaremos

depositar(monto, nlib)

Aplicar método a ejemplo: esc. principal de "Otorga Crédito"

- 1. Identificamos los escenarios "importantes" del sistema.
- 2. A partir del DSS de cada uno de estos escenarios, obtenemos los eventos del sistema.
- 3. Cada evento del sistema de ese escenario lo transformamos en un diagrama de interacción (similar a los vistos en las realizaciones).

Deberemos transformar este evento del sistema



- Deberemos transformar
  depositar(monto, nlib)
  en un diagrama de interacción.
- Pero esto no sabemos cómo hacerlo todavía. Necesitamos un método

Método para obtener 1 diagr. de interacción a partir de 1 evento

- 3.1. El evento del sistema lo dirigimos hasta el controlador del caso de uso y lo ponemos en una lista de mensajes pendientes de tratar.
- 3.2. Se elige un mensaje pendiente de la lista:
  - 3.2.1. Se hace una lista de tareas que deben hacerse para servirlo.
  - 3.2.2. Para cada tarea de la lista:
    - Se decide, aplicando **patrones,** qué objeto tiene la responsabilidad de realizar cada una de estas tareas.
    - Si la responsabilidad cae sobre el mismo objeto, no se hace nada.
    - Si cae sobre otro objeto, se crea un mensaje hacia el objeto (hay que tener en cuenta parámetros y visibilidad). Se añade el mensaje a la lista de pendientes.
- 3.3. Se quita el mensaje de pendientes. Repite 3.2 hasta

El punto crucial

- 3.1. El evento del sistema lo dirigimos hasta el controlador del caso de uso y lo ponemos en una lista de mensajes pendientes de tratar.
- 3.2. Se elige un mensaje pendiente de la lista:
  - 3.2.1. Se hace una lista de tareas que deben hacerse para servirlo.
  - 3.2.2. Para cada tarea de la lista:
    - Se decide, aplicando **patrones,** qué objeto tiene la responsabilidad de realizar cada una de estas tareas.
    - Si la responsabilidad cae sobre el mismo objeto, no se hace nada.
    - Si cae sobre otro objeto, se crea un mensaje hacia el objeto (hay que tener en cuenta parámetros y visibilidad). Se añade el mensaje a la lista de pendientes.
- 3.3. Se quita el mensaje de pendientes. Repite 3.2 hasta

Se decide qué objeto realiza cada una de las tareas

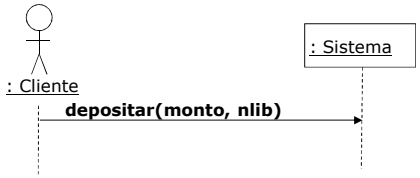
- Este es el punto crucial ¿Cuál es el objeto que se encargará de realizar una de las tareas?
- Aplicaremos patrones. Los objetos a los que aplicaremos patrones pueden ser de tres clases
  - 1. Puede ser un objeto que ya está en el diagrama de interacción.
  - 2. Puede ser una instancia de las clases del modelo de dominio. Por ello, tenemos que tener el modelo de dominio en cuenta al aplicar este método.
  - 3. Si ni 1 ni 2 son adecuados, puede ser una instancia de una clase artificial que "nos inventamos" (patrón "Pure Fabrication")

Nota: no tendremos en cuenta la visibilidad todavía.

- 3.1. El evento del sistema lo dirigimos hasta el
Esto es porque la estudiaremos más tarde. Por ahora no la consideraremos.
de
- 3.2. Se elige un mensaje pendiente de la lista:
  - 3.2.1. Se hace una lista de tareas que deben hacerse para servirlo.
  - 3.2.2. Para cada tarea de la lista:
    - Se decide, aplicando **patrones,** qué objeto tiene la responsabilidad de realizar cada una de estas tareas.
    - Si la responsabilidad cae sobre el mismo objeto, no se hace nada.
    - Si cae sobre otro objeto, se crea un mensaje hacia el objeto (hay que tener en cuenta parámetros y visibilidad). Se añade el mensaje a la lista de pendientes.
- 3.3. Se quita el mensaje de pendientes. Repite 3.2 hasta

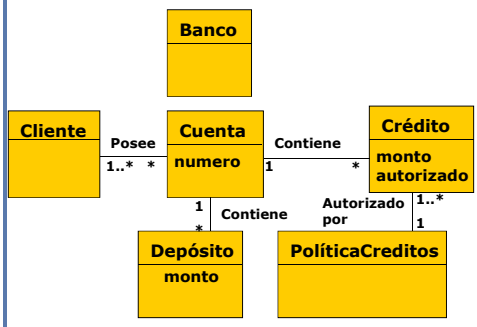
Veamos un ejemplo

- Aplicaremos este método a nuestro ejemplo del caso de uso "Deposita Libreta".
- En nuestro caso, el evento del sistema es "depositar".



Mail: palasi@palasi.com Web: www.palasi.com

Tendremos en cuenta el modelo de dominio



Método para obtener 1 diagr. de interacción a partir de 1 evento

- 3.1. El evento del sistema lo dirigimos hasta el controlador del caso de uso y lo ponemos en una lista de mensajes pendientes de tratar.

Cómo hemos visto, cada caso de uso tendrá una clase controladora. En nuestro caso, es el caso de uso "Deposita Libreta". Entonces, llamaremos a la clase controladora "CtrlDepositaLibreta").

Será un objeto de esta clase el que recibirá el evento del sistema, que es "depositar".Esto según el patrón "Controller"

Además, pondremos a "depositar" en la lista de mensajes pendientes de tratar.

Es decir, tenemos esto

➢Hacemos que el evento del sistema lo recoja la clase controladora. Patrón "Controller".

➢Ponemos a este evento como un mensaje pendiente de tratar.

Método para obtener 1 diagr. de interacción a partir de 1 evento

- 3.1. El evento del sistema lo dirigimos hasta el controlador del caso de uso y lo ponemos en una lista de mensajes pendientes de tratar.
- 3.2. Se elige un mensaje pendiente de la lista:
  - 3.2.1. Se hace una lista de tareas que deben hacerse para servirlo.
  - 3.2.2. Para cada tarea de la lista:
    - Se decide, aplicando **patrones,** qué objeto tiene la responsabilidad de realizar cada una de estas tareas.
    - Si la responsabilidad cae sobre el mismo objeto, no se hace nada.
    - Si cae sobre otro objeto, se crea un mensaje hacia el objeto (hay que tener en cuenta parámetros y visibilidad). Se añade el mensaje a la lista de pendientes.
- 3.3. Se quita el mensaje de pendientes. Repite 3.2 hasta

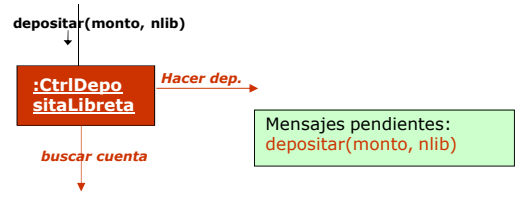
Elegir un mensaje de la lista de pendientes es fácil

➢Elegimos "depositar".

➢Debemos hacer una lista de tareas para servirlo.

➢¿Qué tareas debemos hacer para servir la petición de un crédito?.

Lista de tareas

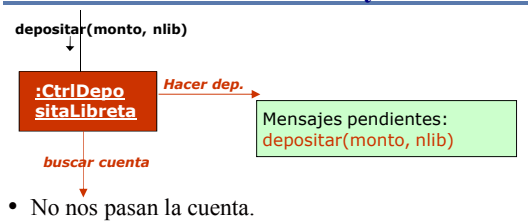


➢Primero: Buscar la cuenta donde se debe depositar.
➢Segundo: Efectuar el depósito.

Para distinguirlas, a las tareas las representaré como flechas rojas.

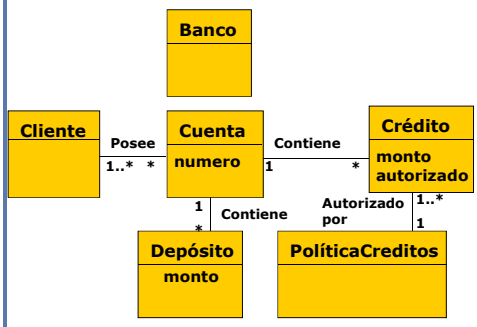
Mail: palasi@palasi.com Web: www.palasi.com

Un momento, ¿no tenemos ya la cuenta?

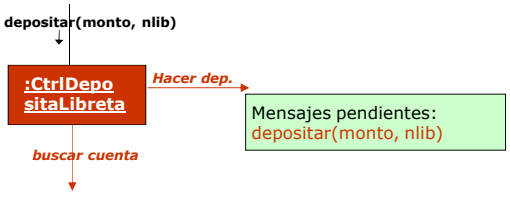


- No nos pasan la cuenta.
- Nos pasan el número de cuenta, pero no la cuenta.
- Debemos encontrar el objeto "Cuenta" a partir del número para poder aplicarle el depósito.

Recordemos que Cuenta es una clase



Lista de tareas

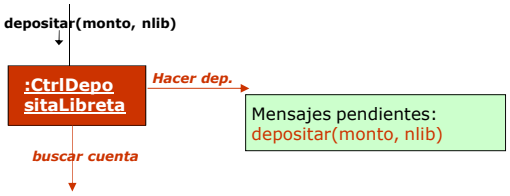


➢Primero: Buscar la cuenta donde se debe depositar.
➢Segundo: Efectuar el depósito.

Método para obtener 1 diagr. de interacción a partir de 1 evento

- 3.1. El evento del sistema lo dirigimos hasta el controlador del caso de uso y lo ponemos en una lista de mensajes pendientes de tratar.
- 3.2. Se elige un mensaje pendiente de la lista:
  - 3.2.1. Se hace una lista de tareas que deben hacerse para servirlo.
  - 3.2.2. Para cada tarea de la lista:
    - Se decide, aplicando **patrones,** qué objeto tiene la responsabilidad de realizar cada una de estas tareas.
    - Si la responsabilidad cae sobre el mismo objeto, no se hace nada.
    - Si cae sobre otro objeto, se crea un mensaje hacia el objeto (hay que tener en cuenta parámetros y visibilidad). Se añade el mensaje a la lista de pendientes.
- 3.3. Se quita el mensaje de pendientes. Repite 3.2 hasta

Comenzamos con la primera tarea



➢Primero: Buscar la cuenta donde se debe depositar.

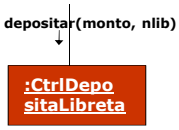
- ¿Qué objeto nos permitiría encontrar la cuenta?
- Aplicando "Information Expert" es el que tenga la información para encontrarla.

Vamos a buscar el objeto

- 1. Puede ser un objeto que ya está en el diagrama de interacción.
- 2. Puede ser una instancia de las clases del modelo de dominio.
- 3. Si ni 1 ni 2 son adecuados, puede ser una instancia de una clase artificial que "nos inventamos" (patrón "Pure Fabrication").

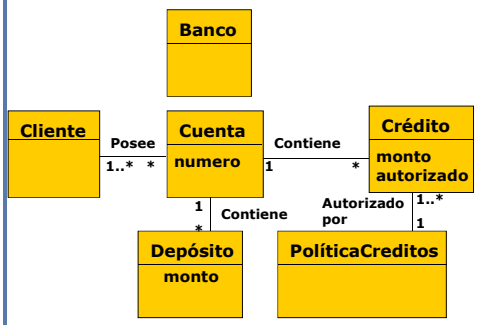
Mail: palasi@palasi.com Web: www.palasi.com

Buscando el objeto para "Information Expert"



- Primero lo buscamos en el diagrama de interacción.
- ¿El objeto que hay tiene la información para poder encontrar una cuenta?
- Claramente no. Tendremos que buscar en otro sitio.

Buscándolo en el modelo de dominio



Cuenta parece estar relacionado.

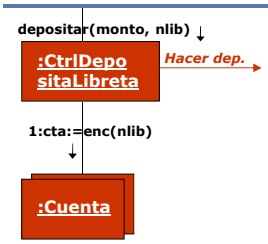
Aplicando "Information Expert" para encontrar el objeto

- Es el **grupo de todas las cuentas** quien tiene la información de cómo encontrar una determinada cuenta.
- Es decir, es el multiobjeto el que es responsable de esta tarea, por "Information Expert".
- A él enviaremos el mensaje de encontrar una cuenta.

Método para obtener 1 diagr. de interacción a partir de 1 evento

- 3.1. El evento del sistema lo dirigimos hasta el controlador del caso de uso y lo ponemos en una lista de mensajes pendientes de tratar.
- 3.2. Se elige un mensaje pendiente de la lista:
  - 3.2.1. Se hace una lista de tareas que deben hacerse para servirlo.
  - 3.2.2. Para cada tarea de la lista:
    - Se decide, aplicando **patrones**, qué objeto tiene la responsabilidad de realizar cada una de estas tareas.
    - Si la responsabilidad cae sobre el mismo objeto, no se hace nada.
    - Si cae sobre otro objeto, se crea un mensaje hacia el objeto (hay que tener en cuenta parámetros y visibilidad). Se añade el mensaje a la lista de pendientes.
- 3.3. Se quita el mensaje de pendientes. Repite 3.2 hasta

Tendríamos esto

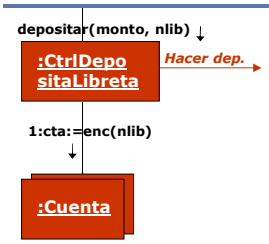


Abreviamos "enc" por "encontrar", la operación de todo multiobjeto.

Como no cae sobre el mismo objeto, debemos hacer un mensaje a este objeto.

Vemos los parámetros y resultados de encontrar. El parámetro es un número de libreta y el resultado es una cuenta.

Sin embargo



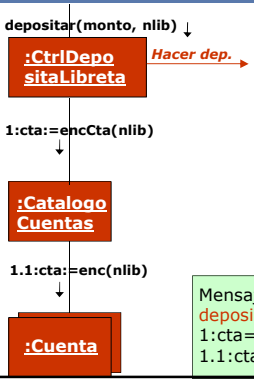
Tenemos un multiobjeto. Siempre que tenemos un multiobjeto aplicaremos el patrón "Multiobject".

Este multiobjeto es de "tipo 1": todas las cuentas del sistema.

Por lo tanto quien tiene que emitir el mensaje de encontrar es un objeto Catálogo.

Mail: palasi@palasi.com Web: www.palasi.com

Aplicando el patrón "Multiobject"

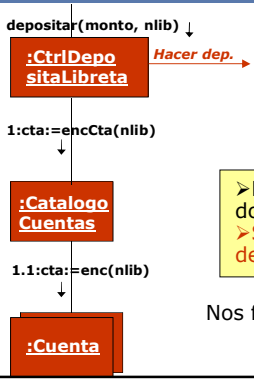


Hay una nueva clase "CatalogoCuentas", que envía el mensaje al multiobjeto. Ella recibe un mensaje para encontrar cuenta.

Hay dos nuevos mensajes, que pondremos en la lista de programas pendientes.

Mensajes pendientes:
depositar(monto, nlib)
1:cta=encCta(nlib)
1.1:cta:=enc(nlib)

Pero aún no hemos acabado con el primer mensaje



Mensajes pendientes:
depositar(monto, nlib)
1:cta=encCta(nlib)
1.1:cta:=enc(nlib)

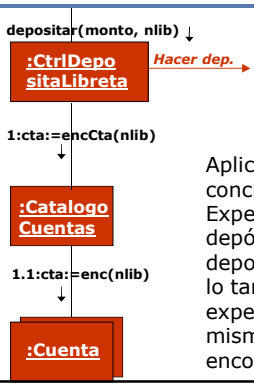
- Primero: Buscar la cuenta donde se debe depositar.
- Segundo: Efectuar el depósito.

Nos falta la segunda tarea

Método para obtener 1 diagr. de interacción a partir de 1 evento

- 3.1. El evento del sistema lo dirigimos hasta el controlador del caso de uso y lo ponemos en una lista de mensajes pendientes de tratar.
- 3.2. Se elige un mensaje pendiente de la lista:
  - 3.2.1. Se hace una lista de tareas que deben hacerse para servirlo.
  - 3.2.2. Para cada tarea de la lista:
    - Se decide, aplicando **patrones,** qué objeto tiene la responsabilidad de realizar cada una de estas tareas.
    - Si la responsabilidad cae sobre el mismo objeto, no se hace nada.
    - Si cae sobre otro objeto, se crea un mensaje hacia el objeto (hay que tener en cuenta parámetros y visibilidad). Se añade el mensaje a la lista de pendientes.
- 3.3. Se quita el mensaje de pendientes. Repite 3.2 hasta

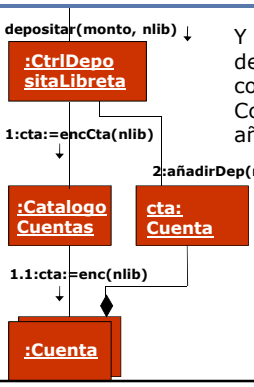
¿Qué objeto tiene la responsabilidad de la 2ª tarea?



- Primero: Buscar la cuenta donde se debe depositar.
- Segundo: Efectuar el depósito.

Aplicamos patrones, concretamente "Information Expert". Para efectuar el depósito necesitamos depositarlo en la cuenta. Por lo tanto, parece que el experto en información es la misma cuenta que hemos encontrado.

Dibujamos por separado la cuenta que hemos obtenido

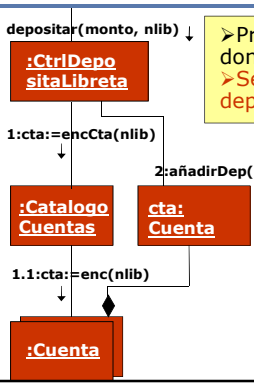


Y a ella le enviamos la tarea de efectuar el depósito convertida en mensaje. Como es lógico, le añadimos el monto.

Tenemos 1 mensaje pendiente más:

Mensajes pendientes:
depositar(monto, nlib)
1:cta=encCta(nlib)
1.1:cta:=enc(nlib)
2:añadirDep(monto)

Con esto hemos acabado con el primer mensaje



- Primero: Buscar la cuenta donde se debe depositar.
- Segundo: Efectuar el depósito.

Quitamos el mensaje de pendientes.

Mensajes pendientes:
1:cta=encCta(nlib)
1.1:cta:=enc(nlib)
2:añadirDep(monto)

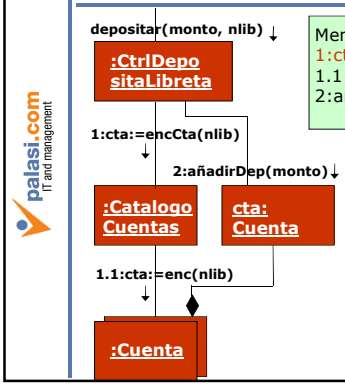
Mail: palasi@palasi.com Web: www.palasi.com

Método para obtener 1 diagr. de interacción a partir de 1 evento

- 3.1. El evento del sistema lo dirigimos hasta el controlador del caso de uso y lo ponemos en una lista de me
- 3.2
  - 3.2.1. Se hace una lista de tareas que deben hacerse para servirlo.
  - 3.2.2. Para cada tarea de la lista:
    - Se decide, aplicando **patrones,** qué objeto tiene la responsabilidad de realizar cada una de estas tareas.
    - Si la responsabilidad cae sobre el mismo objeto, no se hace nada.
    - Si cae sobre otro objeto, se crea un mensaje hacia el objeto (hay que tener en cuenta parámetros y visibilidad). Se añade el mensaje a la lista de pendientes.
- 3.3. Se quita el mensaje de pendientes. Repite 3.2 hasta

A partir de ahora iremos más rápido sin entrar tanto en detalle pero siempre siguiendo este método.

Tenemos los siguientes mensajes pendientes



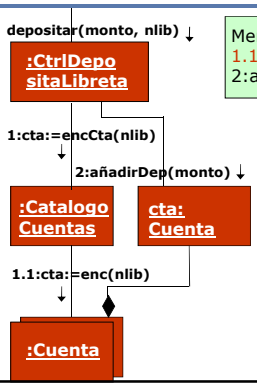
Mensajes pendientes:
1:cta=encCta(nlib)
1.1:cta:=enc(nlib)
2:añadirDep(monto)

Qué tareas tiene el primer mensaje pendiente: Encontrar una cuenta a partir del número.

Qué objeto lo debe cumplir: Ya lo hemos decidido: el catálogo.

Hay algo más a hacer con este mensaje: No. Lo suprimimos.

Tenemos los siguientes mensajes pendientes



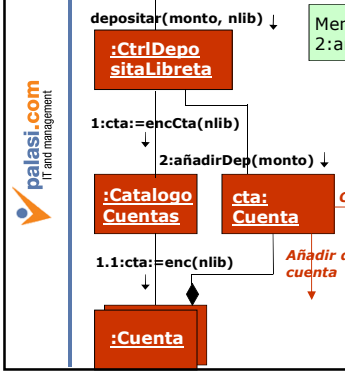
Mensajes pendientes:
1.1:cta:=enc(nlib)
2:añadirDep(monto)

Qué tareas tiene el primer mensaje pendiente: Encontrar una cuenta entre el multiobjeto.

Qué objeto lo debe cumplir: El mismo multiobjeto. Como es el mismo objeto, no añadimos ningún mensaje.

Hay algo más a hacer con este mensaje: No. Lo suprimimos.

Tenemos los siguientes mensajes pendientes

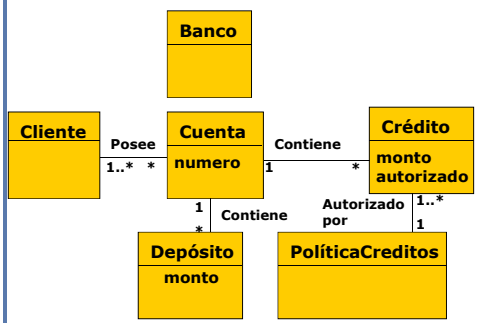


Mensajes pendientes:
2:añadirDep(monto)

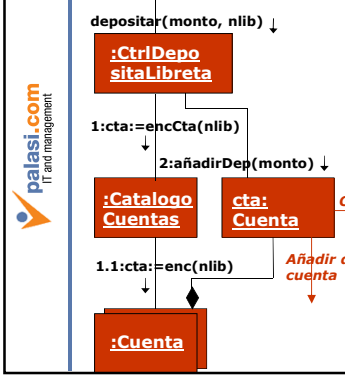
Qué tareas tiene el primer mensaje pendiente: Para añadir un nuevo depósito se deben hacer dos tareas.

- Primero: Crear un nuevo depósito.
- Segundo: Añadirlo a la cuenta.

Recordemos que Depósito también es un objeto.



Comenzamos por la primera tarea



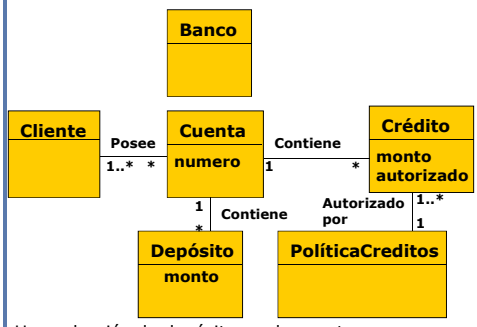
¿Quién es el objeto encargado de crear un nuevo depósito?

Aplicamos el patrón "Creator".

Un objeto que contiene o agrega depósitos.

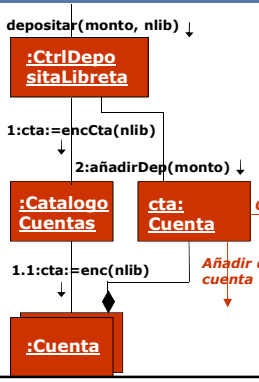
Mail: palasi@palasi.com Web: www.palasi.com

Por lo que parece una cuenta contiene depósitos



Una colección de depósitos cada cuenta, como vemos en la multiplicidad.

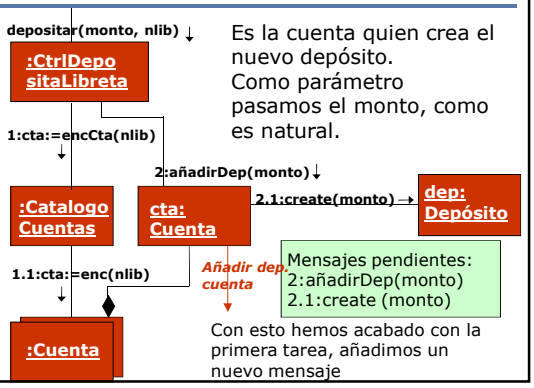
Decidimos que es la cuenta la que creará el depósito



No cualquier cuenta, sino la que hemos encontrado, ya que es esta la que debe contener el depósito que crearemos.

Recordemos que seguimos el patrón "Creator", pues la cuenta contiene depósitos.

Tendríamos esto

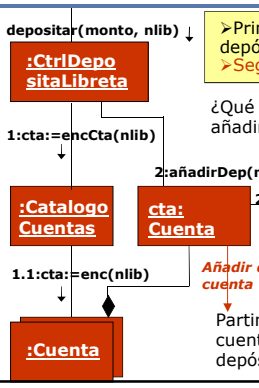


Es la cuenta quien crea el nuevo depósito.
Como parámetro pasamos el monto, como es natural.

Mensajes pendientes:
2:añadirDep(monto)
2.1:create (monto)

Con esto hemos acabado con la primera tarea, añadimos un nuevo mensaje

Con la primera tarea hemos acabado, vamos con la segunda

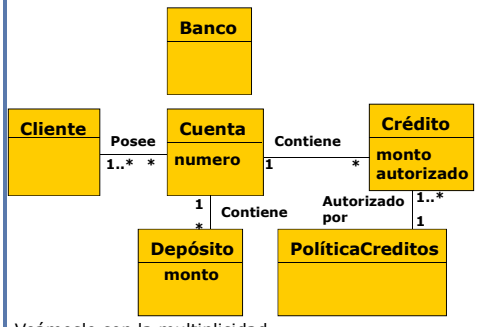


- Primero: Crear un nuevo depósito.
- Segundo: Añadirlo a la cuenta.

¿Qué objeto es el encargado de añadir el depósito a la cuenta?

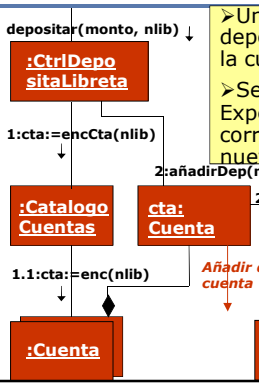
Partimos del hecho de que una cuenta tiene una colección de depósitos.

Una cuenta contiene una colección de depósitos



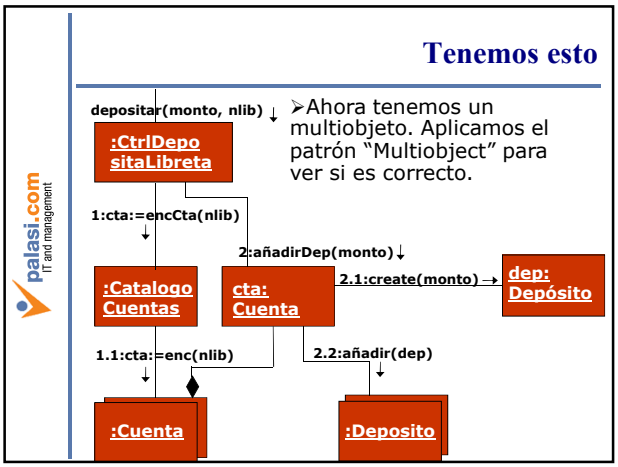
Veámoslo con la multiplicidad.

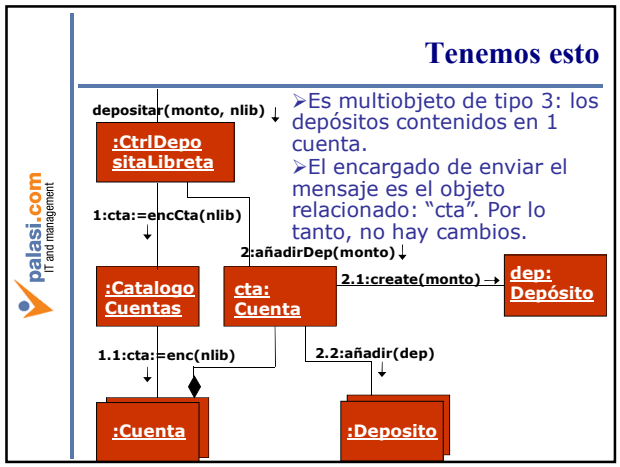
Tenemos lo siguiente

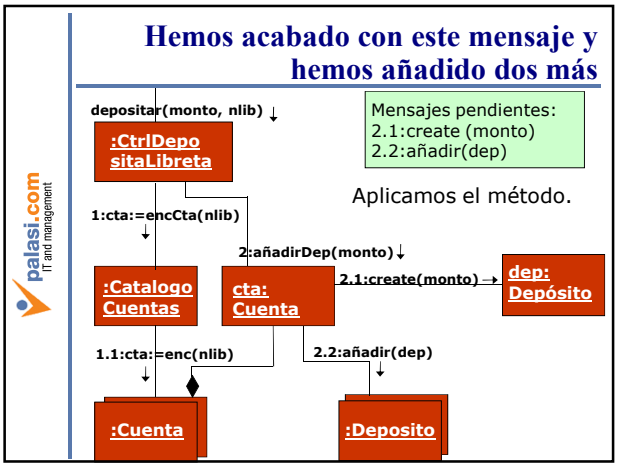


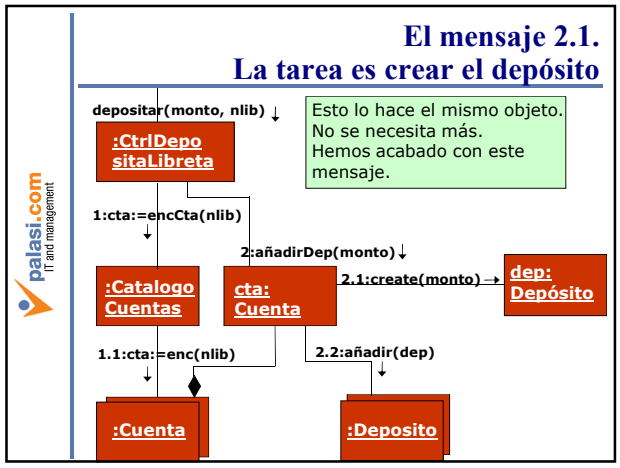
- Un multiobjeto de depósitos que pertenecen a la cuenta.
- Según "Information Expert" es el que le corresponde añadir el nuevo objeto.

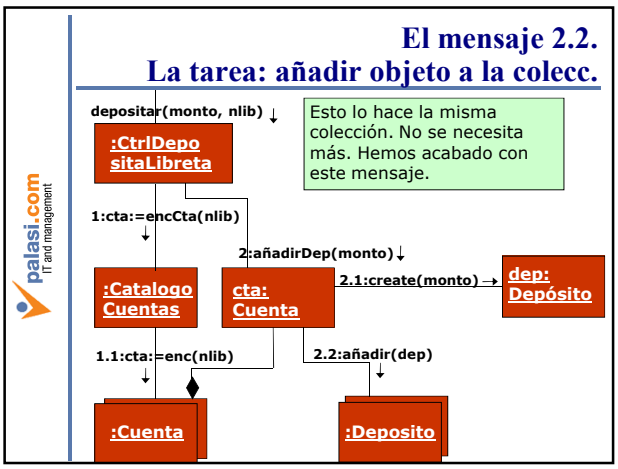
Ya que es el multiobjeto es el que puede añadir a sí mismo.









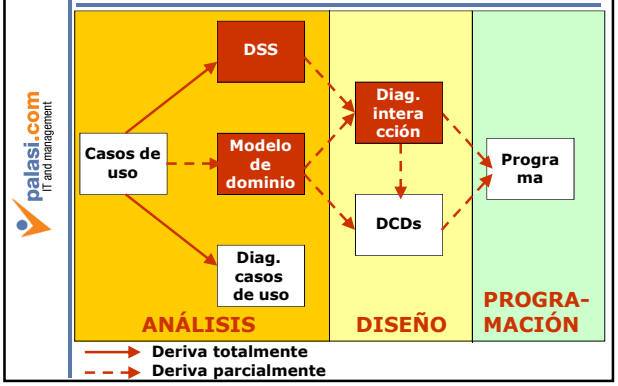


Mail: palasi@palasi.com Web: www.palasi.com

Hemos realizado este ejemplo muy detallado

- Para que se vea que sólo hay que aplicar el método y seguir los patrones.
- Que todas las decisiones de diseño están justificadas y que no hay que inventarse nada.
- Sin embargo, a la hora de la realidad esto es muy rápido, pues no se apuntan todos los datos sino que se hace sobre la marcha.

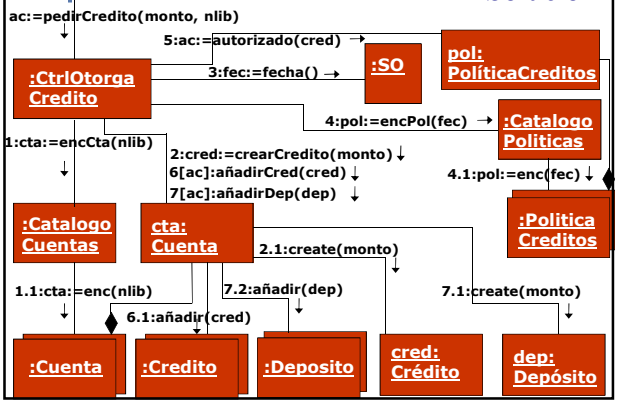
Relación de los diagramas de interacción con otros artefactos



Ejercicio

- Conseguir el diagrama de interacción para el caso de uso "Otorga Crédito".
- Apliquen el método.

Solución



3. Diseño orientado a objetos con UML

Visibilidad

- Tema importantísimo a la hora de hacer el diseño de software y de crear realizaciones de casos de uso como las que hemos visto.
- Hasta ahora no habíamos hablado de él para no complicar la exposición presentándolo todo al mismo tiempo.
- Vamos a explicarlo ahora y tendrá importancia a partir de ahora.

Mail: palasi@palasi.com Web: www.palasi.com

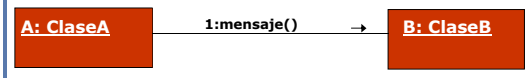
¿Qué es la visibilidad?

- La visibilidad es la habilidad de un objeto de poder "ver" otro objeto.
- Esto es muy parecido al concepto de "ámbito de las variables", pero en programación orientada a objetos.
- Se trata de saber si un objeto es visible (y accesible) dentro de otro.
- ¿Por qué es importante esto?

Esto es importante porque

Regla de oro:

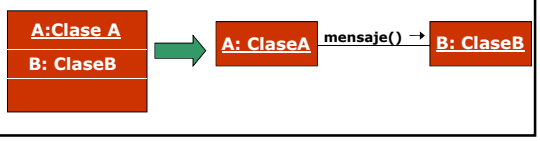
Para que un objeto A envíe un mensaje a B, B debe ser visible en A.



- Esto es importante porque determina mucho como se diseñan los diagramas de interacción y, por lo tanto, como es el programa.
- Cada vez que escribimos un mensaje en un diagrama de interacción, debemos pensar "¿este mensaje

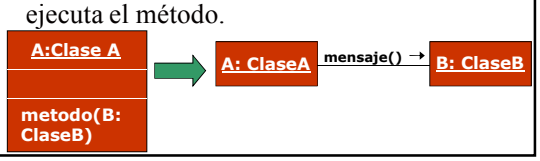
Cuatro tipos de visibilidad (1)

- **1. Visibilidad de atributo.** B es visible en A porque B es un atributo de A.
- Los atributos de un objeto son visibles al mismo objeto.
- Es la más común. Es estable pues dura mientras duran los objetos B y A.



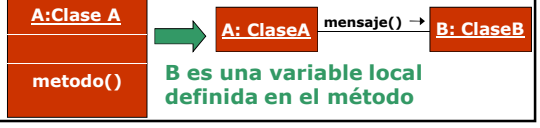
Cuatro tipos de visibilidad (2)

- **2. Visibilidad de parámetro.** B es visible en A porque B es un parámetro de un método de A.
- Dentro del método, B es visible y por lo tanto, se le pueden enviar mensajes.
- Es la 2ª más común. Sólo dura mientras se ejecuta el método.



Cuatro tipos de visibilidad (3)

- **3. Visibilidad local.** B es visible en A porque B es una variable local de un método de A.
- Dentro del método, B es visible y por lo tanto, se le pueden enviar mensajes.
- Es la 3ª más común. Sólo dura mientras se ejecuta el método.



Cuatro tipos de visibilidad (4)

- **4. Visibilidad global.** B es visible en A porque B es global (y visible) a todos los objetos.
- En algunos lenguajes, B es una variable global. En otros lenguajes, B es un singleton o una posición JNDI, etc.
- Es poco común y es estable. Dura mientras existan A y B.

3. Diseño orientado a objetos con UML

- 3.1. Introducción al diseño orientado a objetos.
- 3.2. Diagramas de secuencia.
- 3.3. Diagramas de colaboración.
- 3.4. Patrones para asignar responsabilidades.
- 3.5. Método para obtener los diagramas de interacción.
- 3.6. Visibilidad.
- 3.7. Diagramas de clases de diseño.

Recordemos: El diseño del programa constará de dos artefactos.

- De hecho, puede constar de más pero estos dos son los principales.
- Estos dos artefactos son:
  - **El diagrama de interacción.**
  - **El diagrama de clases de diseño.**
- Ya hemos visto los diagramas de interacción pero ahora veremos el diagrama de clases de diseño.

Diagramas de clases de diseño (DCDs)

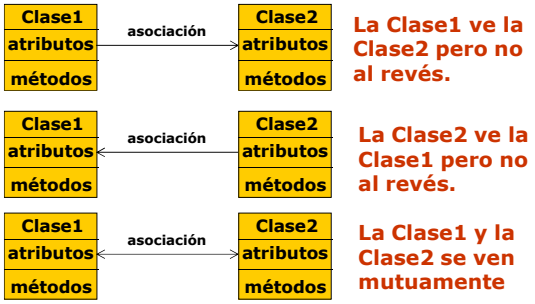
- Son diagramas que representan las diferentes **clases de software** que tiene nuestra aplicación junto con las relaciones entre estas clases.
- Importante: se trata de
  - clases de software (clases del programa)
  - **NO** de clases conceptuales (clases del mundo real).

Los DCDs son parecidos al modelo de dominio

- Son clases relacionadas entre sí por asociaciones.
- Las diferencias son:
  - En un DCD las clases son de software no conceptuales.
  - En un DCD se indican los métodos.
  - En un DCD se indica la visibilidad entre las clases.

¿Cómo indicamos la visibilidad en un DCD?

- La visibilidad se indica en las asociaciones por una flecha.

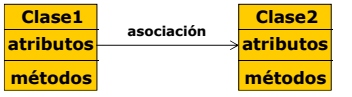


Recordemos que si B es visible des de A (si A ve B)

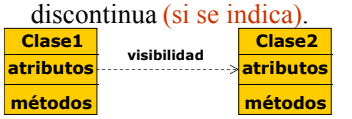
- Quiere decir que se puede mandar un mensaje desde A a B.
- Por esto, esta visibilidad se deduce de los diagramas de interacción que es donde indicamos los mensajes.
- ¿Qué tipo de visibilidad?

Tipos de visibilidad

- La visibilidad de atributo se indica con una flecha sólida.



- Las otras visibilidades (de parámetro, global o local) se indican con una flecha discontinua (si se indica).



Tipos de visibilidad

- Las asociaciones entre clases tienen visibilidad de atributo.

- Las otras visibilidades no están relacionadas con las asociaciones.

¿Qué visibilidades se deben indicar?

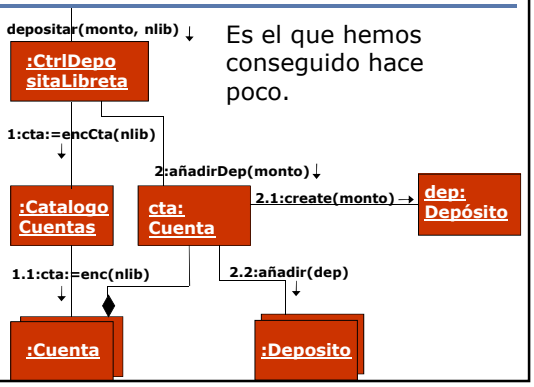
- Las asociaciones deben tener todas su visibilidad en el DCD.

- Las otras visibilidades se indican raramente, sólo si son significativas.

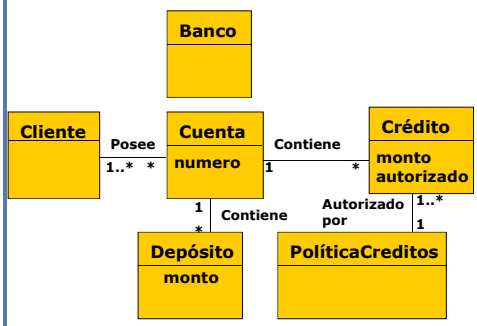
¿Cómo se crea un DCD a partir de los otros artefactos?

- El DCD se crea a partir del diagrama de interacción y del modelo de dominio.
- Un diagrama DCD está formado por:
  - Clases.
  - Asociaciones.
  - Opcionalmente, visibilidad no de asociaciones.
  - Métodos.
  - Atributos.
- A partir de ahora, explicaremos cómo se obtienen cada uno de estos elementos. Lo aplicaremos al ejemplo de depositar libreta.

Diagrama de interacción de "Depositar Libreta"



Modelo de dominio de "Depositar Libreta"



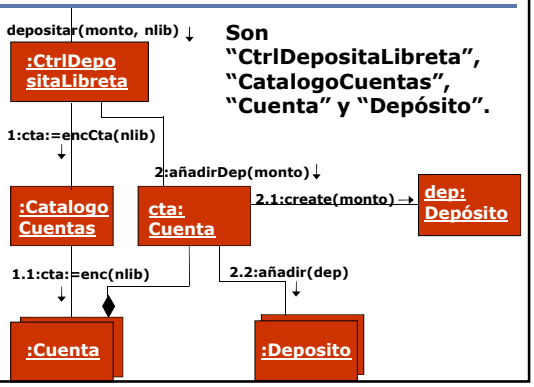
¿Cómo se crea un DCD a partir de los otros artefactos?

- El DCD se crea a partir del diagrama de interacción y del modelo de dominio.
- Un diagrama DCD está formado por:
  - Clases.
  - Asociaciones.
  - Opcionalmente, visibilidad no de asociaciones.
  - Métodos.
  - Atributos.
- A partir de ahora, explicaremos cómo se obtienen cada uno de estos elementos. Lo aplicaremos al ejemplo de depositar libreta.

Clases

- Sencillo. Las clases del DCD son las que aparecen en los diagramas de interacción.

En nuestro caso, hay cuatro clases



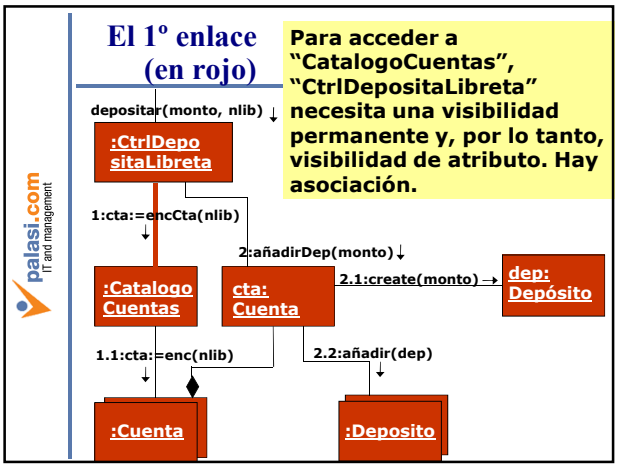
El DCD tendrá 4 clases

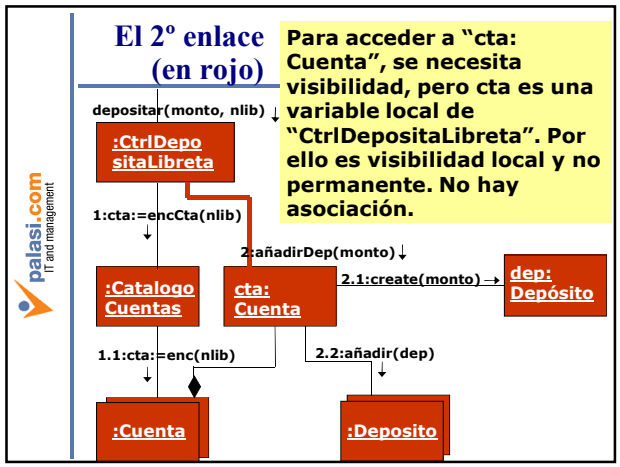
¿Cómo se crea un DCD a partir de los otros artefactos?

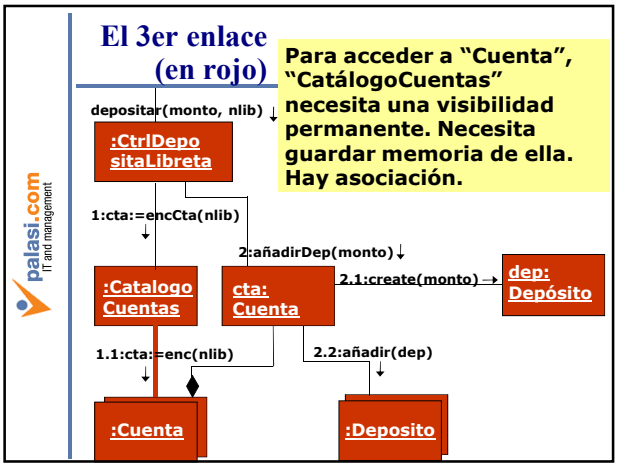
- El DCD se crea a partir del diagrama de interacción y del modelo de dominio.
- Un diagrama DCD está formado por:
  - Clases.
  - Asociaciones.
  - Opcionalmente, visibilidad no de asociaciones.
  - Métodos.
  - Atributos.
- A partir de ahora, explicaremos cómo se obtienen cada uno de estos elementos. Lo aplicaremos al ejemplo de depositar libreta.

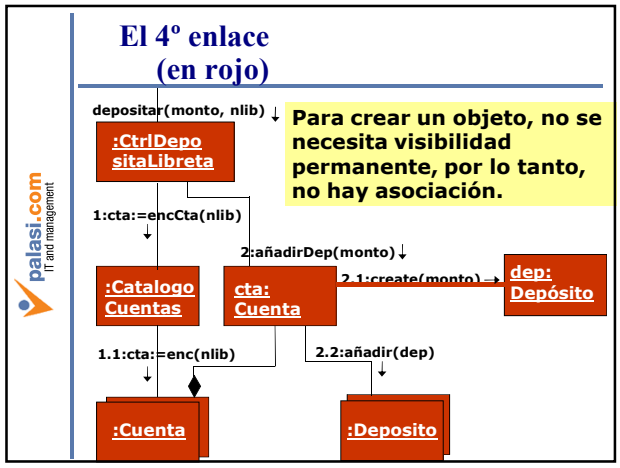
Asociaciones

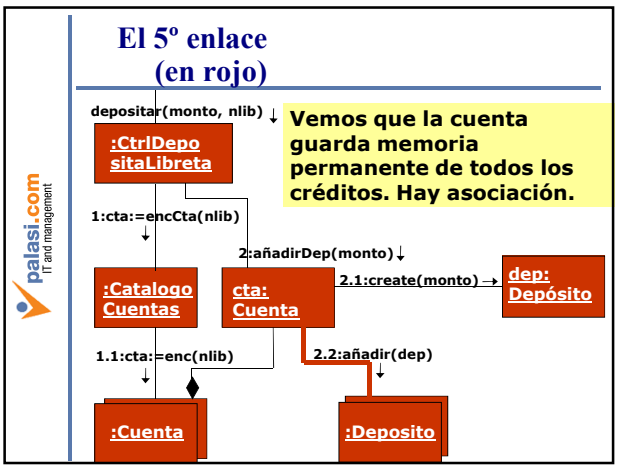
- Se definen las asociaciones que son necesarias para satisfacer la visibilidad y memoria que se necesitan en los diagramas de interacción.
- Más concretamente, A tiene una asociación con B si en el diagrama de interacción una instancia de A tiene un enlace con una instancia de B y además:
  - A tiene visibilidad de atributo con B, o bien
  - A debe guardar conocimiento con B.
- El método es examinar todos los enlaces en el diagrama y ver si cumplen estas condiciones.

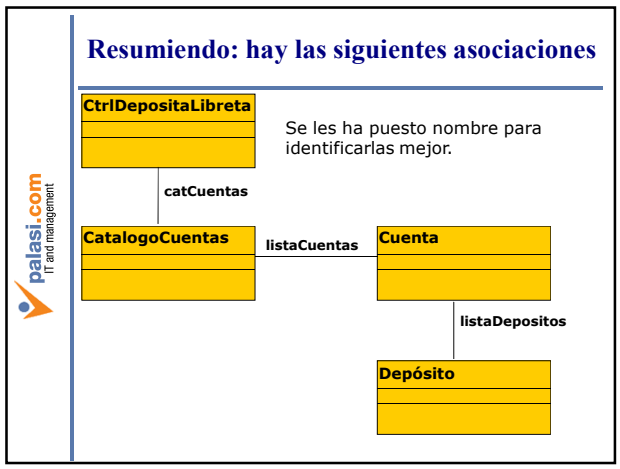








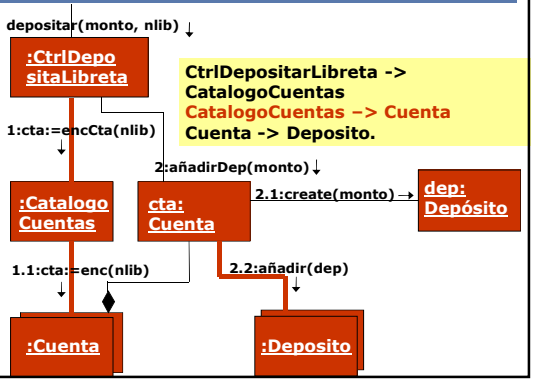




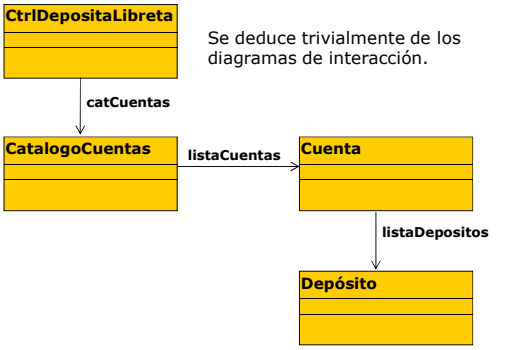
Visibilidad de las asociaciones

- Sencillo. La visibilidad de las asociaciones se deriva de los diagramas de interacción.
- Es decir, si sobre el enlace viaja un mensaje de una instancia de A a una de B, esto quiere decir que la visibilidad va de A a B.

Visibilidad de asociaciones



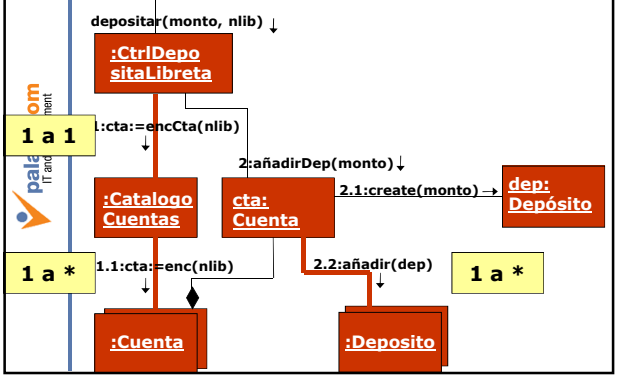
La visibilidad de las asociaciones



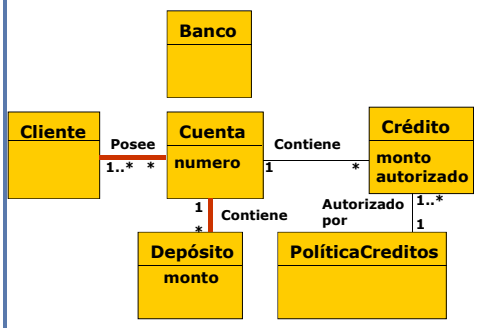
Multiplicidad de las asociaciones

- Los diagramas de interacción nos la indican en forma de los multiobjetos.
- Si una instancia de A está relacionada con una de B, es que hay una relación 1 a 1 (y similares).
- Si una instancia de A está relacionada con un multiobjeto de B, la relación es de 1 a * (o más raramente de * a *).
- Se puede afinar más con la multiplicidad que deducimos del modelo de dominio o de

Multiplicidad de asociaciones

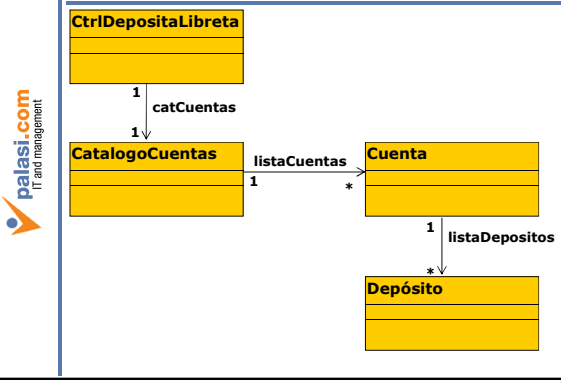


El modelo de dominio confirma esta multiplicidades



Mail: palasi@palasi.com Web: www.palasi.com

Tenemos esto



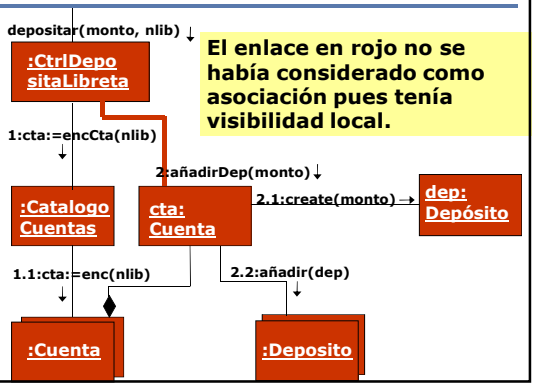
¿Cómo se crea un DCD a partir de los otros artefactos?

- El DCD se crea a partir del diagrama de interacción y del modelo de dominio.
- Un diagrama DCD está formado por:
  - Clases.
  - Asociaciones.
  - Opcionalmente, visibilidad no de asociaciones.
  - Métodos.
  - Atributos.
- A partir de ahora, explicaremos cómo se obtienen cada uno de estos elementos. Lo aplicaremos al ejemplo de depositar libreta.

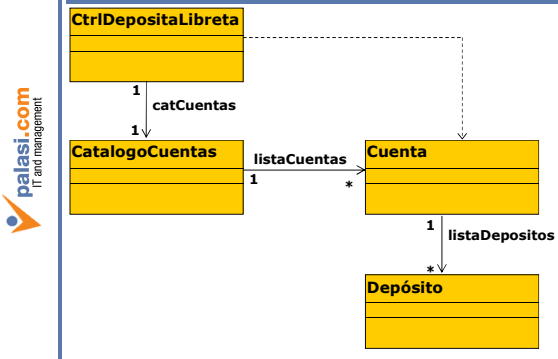
Visibilidad que no está relacionada con las asociaciones

- Es la visibilidad que no es de atributo: local, global o de parámetro.
- Como sabemos, se indica con una línea discontinua.
- Puede no indicarse si no es muy significativa.
- Se deduce fácilmente de los diagramas de interacción: son los enlaces que no se han tomado como asociaciones.

En nuestro caso



Tenemos esto



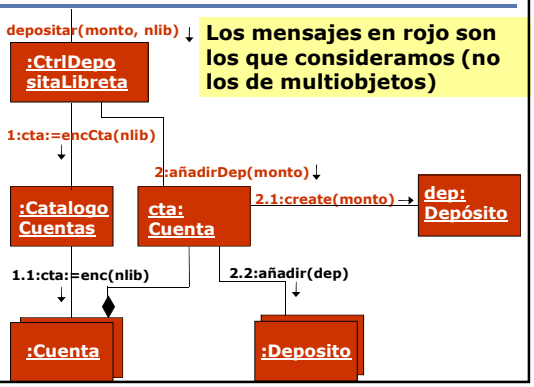
¿Cómo se crea un DCD a partir de los otros artefactos?

- El DCD se crea a partir del diagrama de interacción y del modelo de dominio.
- Un diagrama DCD está formado por:
  - Clases.
  - Asociaciones.
  - Opcionalmente, visibilidad no de asociaciones.
  - Métodos.
  - Atributos.
- A partir de ahora, explicaremos cómo se obtienen cada uno de estos elementos. Lo aplicaremos al ejemplo de depositar libreta.

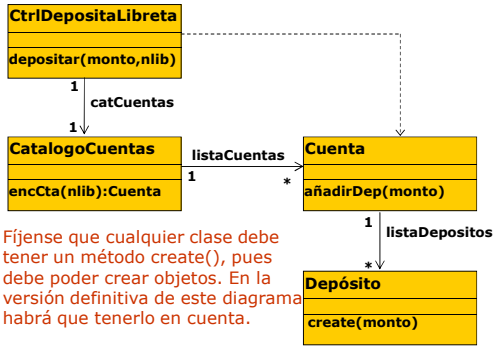
Métodos

- Si en el diagrama de interacción, una instancia de A objeto **recibe** un mensaje M.
- En el DCD, la clase A tiene el método correspondiente a M.
- En este proceso, no se consideran los mensajes que reciben los multiobjetos.

En nuestro caso



Los métodos quedan así



Fíjense que cualquier clase debe tener un método create(), pues debe poder crear objetos. En la versión definitiva de este diagrama habrá que tenerlo en cuenta.

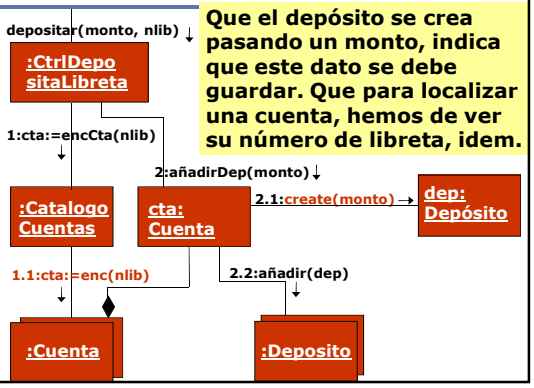
¿Cómo se crea un DCD a partir de los otros artefactos?

- El DCD se crea a partir del diagrama de interacción y del modelo de dominio.
- Un diagrama DCD está formado por:
  - Clases.
  - Asociaciones.
  - Opcionalmente, visibilidad no de asociaciones.
  - Métodos.
  - Atributos.
- A partir de ahora, explicaremos cómo se obtienen cada uno de estos elementos. Lo aplicaremos al ejemplo de depositar libreta.

Atributos

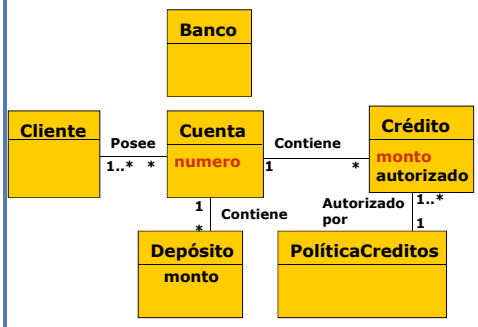
- Los atributos (no clase) se consiguen de tres fuentes:
- De los diagramas de interacción, nos pueden dar una idea de los atributos que necesita conservar cada objeto.
- Del modelo de dominio, podemos extraer atributos que sigan siendo útiles.
- Cualquier atributo que consideremos necesario.
- Sin embargo, hay que tener en cuenta que los atributos (no clase) es lo menos importante.

En nuestro caso

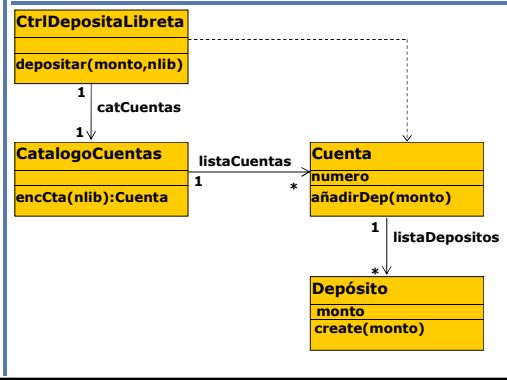


Mail: palasi@palasi.com Web: www.palasi.com

El modelo de dominio confirma estos atributos



Quedamos con esto



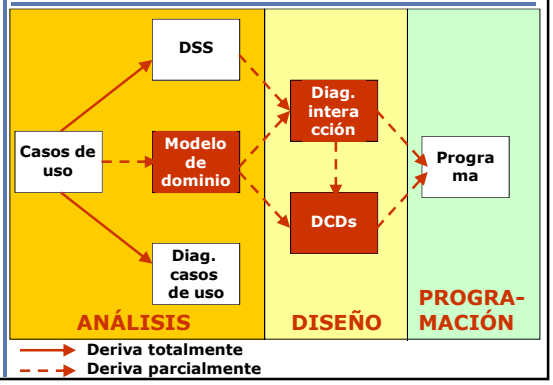
Ya hemos acabado

- Sin embargo, este DCD aún es provisional.
- Se debe completar con la información de otros diagramas de interacción.
- Se debe refinar iterativamente.

Nota

- Aunque aquí se ha explicado primero el diagrama de interacción y después el DCD (por motivos didácticos).
- Normalmente, el diagrama de interacción y el DCD se realizan de forma simultánea.
- Se hace un poco de diagramas de interacción, se modifica un poco el DCD con esta información, se hace un poco más de diagramas de interacción, se vuelve a modificar el DCD y así sucesivamente.
- La información de uno y otro artefacto va influyéndose mutuamente en el diseño.

Relación de los DCDs con otros artefactos



Temario del curso

- 1. Introducción al A & D O-O y a UML.
- 2. Análisis orientado a objetos con UML.
- 3. Diseño orientado a objetos con UML.
- 4. Conclusiones finales.

4. Conclusiones finales

- 4.1. Derivando la programación a partir del diseño.
- 4.2. Reflexiones sobre los artefactos que se han estudiado.
- 4.3. Introducción al modelo de capas.

4. Conclusiones finales

- 4.1. Derivando la programación a partir del diseño.
- 4.2. Reflexiones sobre los artefactos que se han estudiado.
- 4.3. Introducción al modelo de capas.

Derivando la programación del diseño

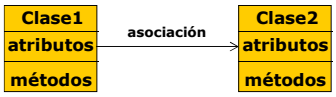
- Éste no es un curso de programación O-O.
- Sin embargo, debemos conocer cómo se aprovechan los artefactos vistos aquí para conseguir el código de programa.
- Claramente, la programación deriva del diseño y, por lo tanto, los diagramas de diseño son los que tenemos que considerar a la hora de programar.

Hay dos artefactos principales en el diseño

- Son:
  - Los diagramas de clases de diseño (DCD).
  - Los diagramas de interacción.
- Los dos nos van a servir para hacer nuestra programación.

El diagrama de clases de diseño

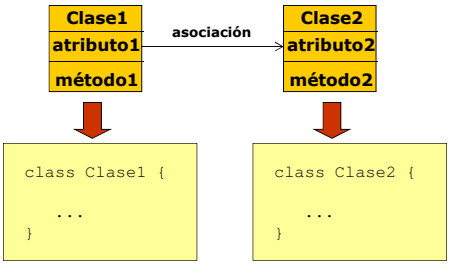
- Contiene clases, atributos (no clase), asociaciones con visibilidad y métodos.



- Este diagrama nos indicará la estructura general de nuestro programa: es decir, su división en clases.

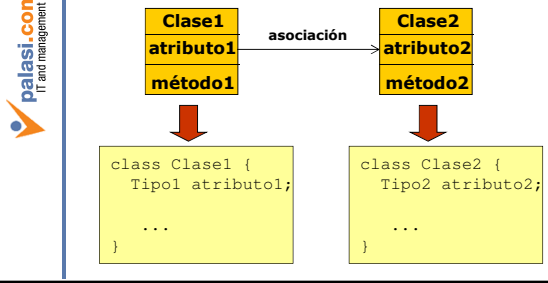
Más concretamente (1)

- Las clases del DCD serán las clases de nuestro programa.



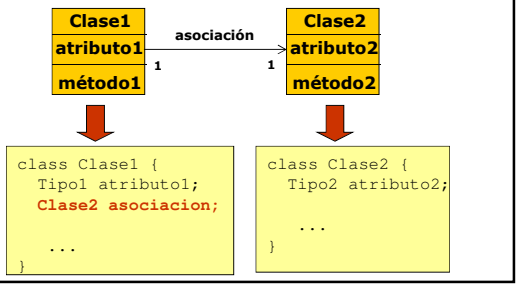
Más concretamente (2)

- Los atributos de las clases del DCD serán atributos de las clases de nuestro programa.



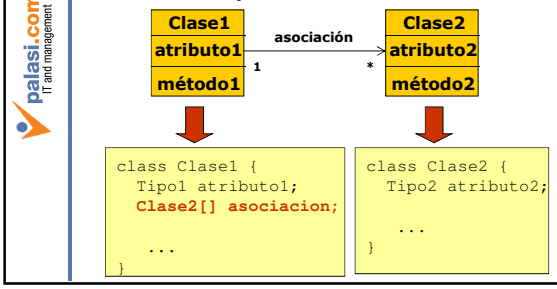
Más concretamente (3)

- Las asociaciones del DCD también serán atributos. Concretamente, la clase que tendrá el atributo es la que ve la otra (es decir, el origen de la flecha). El atributo será del tipo de la otra clase.



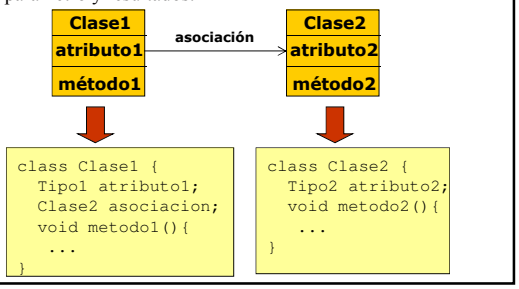
Hay que tener en cuenta la multiplicidad de las asociaciones

- Si es uno a uno, el atributo será un objeto de la clase que es visible (destino de la flecha), como antes. Si es uno a varios o varios a varios será una estructura de datos conteniendo objetos de esta clase.



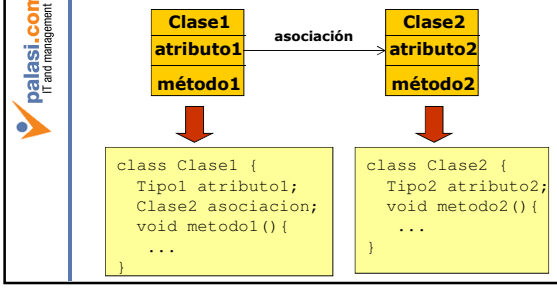
Más concretamente (4)

- Los métodos del DCD se traducirán en las cabeceras de los métodos de las clases del programa, pero no en su implementación (cuerpo). Cabecera: nombre, tipos de parámetro y resultados.

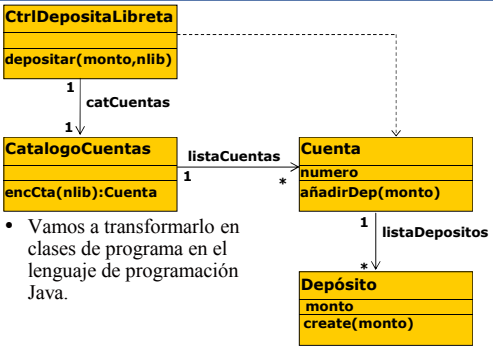


Y eso es todo lo que podemos deducir del DCD

- Es mucho. Prácticamente toda la estructura del programa. Sólo nos queda la implementación (el cuerpo de los métodos).



Ejemplo



- Vamos a transformarlo en clases de programa en el lenguaje de programación Java.

Estructura del código del programa (1)

```
class CtrlDepositaLibreta {
  CatalogoCuentas catCuentas;
  void depositar (int monto, int
    nlib){
  }
}

class CatalogoCuentas {
  Cuenta[] listaCuentas;
  Cuenta encCta(int nlib){
  }
}
```

Estructura del código del programa (2)

```
class Cuenta {
  int numero;
  Deposito[] listaDepositos;
  void añadirDep (int monto){
  }
}

class Deposito {
  int monto;
  Deposito (int monto){
  }
}
```

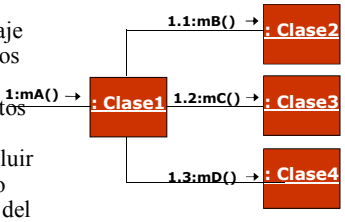
Esta es la forma en que se escribe el método create() en Java. Se le llama constructor.

Ya tenemos la estructura del programa

- Sólo nos falta el cuerpo de los métodos.
- Esto nos lo dará el diagrama de interacción.

Diagrama de interacción

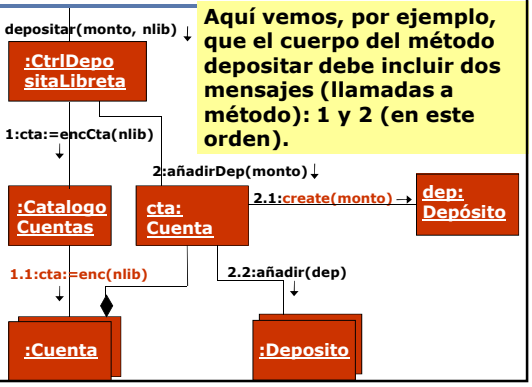
- Si un mensaje produce otros mensajes, entonces estos últimos se deberán incluir en el cuerpo del método del primero.



➢En este caso, el método mA() incluirá en su cuerpo, mensajes (llamadas a métodos) mB(), mC() y mD().

➢Se conserva el orden que hay en el diagrama de interacción.

En nuestro caso



Aquí vemos, por ejemplo, que el cuerpo del método depositar debe incluir dos mensajes (llamadas a método): 1 y 2 (en este orden).

Estructura del código del programa

```
class CtrlDepositaLibreta {
  CatalogoCuentas catCuentas;
  void depositar (int monto, int
    nlib){
    cta:=catCuentas.encCta(nlib);
    cta.añadirDep(monto);
  }
}
```

El objeto de tipo CatalogoCuentas se llama catCuentas (esto no aparece en el diagrama de interacción pero sí en el DCD y en el programa).

Se usa la notación O-O: para mandar un mensaje m() a un objeto A, la sintaxis es A.m()

Mail: palasi@palasi.com Web: www.palasi.com

Sin embargo, esto es demasiado bonito

```
class CtrlDepositaLibreta {
  CatalogoCuentas catCuentas;
  void depositar (int monto, int
    nlib){
    cta:=catCuentas.encCta(nlib);
    cta.añadirDep(monto);
  }
}
```

Este caso por ser simple, todo el código del método "depositar" se puede deducir directamente del diagrama de interacción.

Sin embargo, muchas veces no es así.

El diagrama de interacción no prod. todo el cuerpo de 1 metodo

- Si fuera así, no haría falta programar, la herramienta CASE generaría el programa completo.
- Sólo produce algunas líneas de programa (la que corresponden a los mensajes). Las otras líneas las hemos de programar nosotros.
- Programar sigue siendo una tarea creativa: no se deduce automáticamente del diseño, aunque el diseño nos soluciona todo el tema de la estructura y las cosas importantes.
- Recordemos que el diseño son como los planos de la casa y el programa la casa construida.

En la programación debemos resolver aspectos

- Que no hemos tratado en el diseño.
- No sólo las líneas que faltan del cuerpo de los métodos (como hemos visto), sino cosas como control de errores y excepciones, distribución del programa, etc.
- Pero esto lo trataríamos en un curso de programación orientada a objetos.

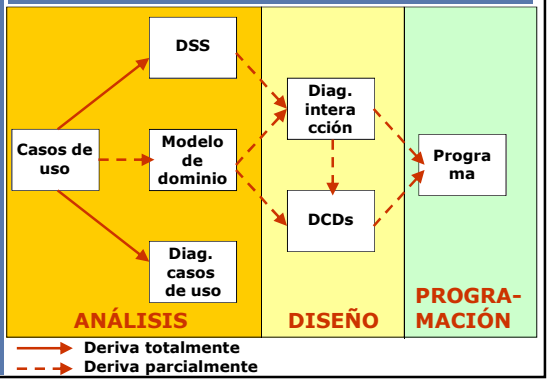
La programación es lo que ocupa la mayor parte del tiempo

- El análisis y diseño, que aquí hemos visto en profundidad y detalle, sólo ocupa unos pocos días al principio de cada iteración.
- Todo el resto del tiempo se dedica a programar y hacer pruebas.
- La idea no es hacer un análisis y diseño "perfecto". Este se irá perfeccionando iterativamente con la retroalimentación que nos da la programación.

4. Conclusiones finales

- 4.1. Derivando la programación a partir del diseño.
- 4.2. Reflexiones sobre los artefactos que se han estudiado.
- 4.3. Introducción al modelo de capas.

Relación de los artefactos que hemos visto



Mail: palasi@palasi.com Web: www.palasi.com

¿Qué artefactos se deben mantener?

- Conforme el desarrollo avanza puede pasar que algunos artefactos sigan siendo útiles (y por lo tanto, debamos modificarlos y mantenerlos).
- Otros sin embargo pueden ser de "usar y tirar". Una vez lo hemos utilizado no nos interesa mantenerlos.
- ¿Cuáles son unos y cuáles son los otros?

¿Qué artefactos se deben mantener?

- Los casos de uso (junto con los DSS y los diagramas de casos de uso) se deben mantener.
- El modelo de dominio después de unas cuantas iteraciones ya no nos interesará y nos guiaremos por el DCD. Por lo tanto, no se debe mantener.
- Los diagramas de interacción deben mantenerse.
- Los DCDs pueden ser generados automáticamente a partir del código por una herramienta CASE. Se mantienen pero automáticamente, sin esfuerzo nuestro.
- Por supuesto, el programa se debe mantener.

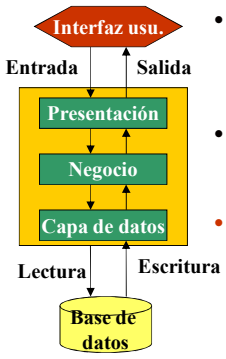
La importancia de una herramienta CASE

- Todo el proceso de análisis y diseño de programación debe realizarse con la ayuda de una herramienta CASE que trabaje con UML.
- La herramienta debe permitir generar código desde los diagramas (generación de código) y diagramas a partir del código (ingeniería inversa).
- Sin una herramienta así, corremos el riesgo de que los diagramas no sean actualizados al mismo tiempo que el código y nuestro proceso de análisis y diseño no sea mantenible.

4. Conclusiones finales

- 4.1. Derivando la programación a partir del diseño.
- 4.2. Reflexiones sobre los artefactos que se han estudiado.
- 4.3. Introducción al modelo de capas.

Recordemos: arquitectura en n-capas



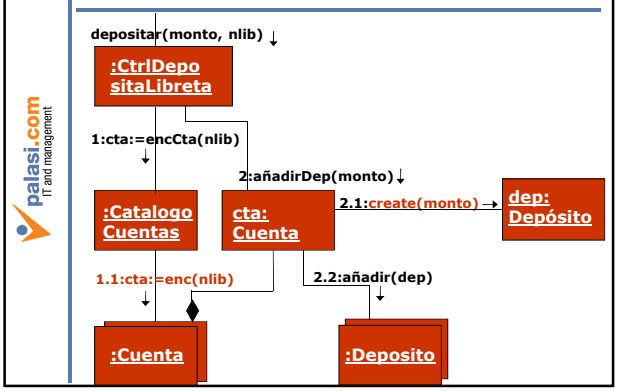
- El programa está compuesto por una serie de capas que son conjuntos de clases.
- Cada capa sólo accede a la inferior.
- **Es decir, las clases de una capa sólo llaman a métodos de clases de la capa inferior.**

Queremos saber

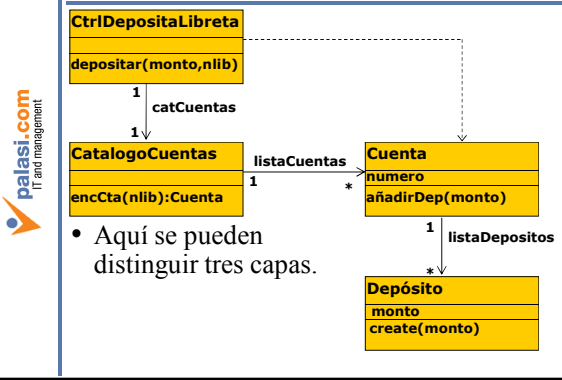
- Qué relación tiene lo que hemos aprendido con la arquitectura en n-capas.
- De hecho, el método de diseño que hemos enseñado a partir de patrones produce automáticamente una estructura de capas.

Mail: palasi@palasi.com Web: www.palasi.com

Recordemos el ejemplo anterior

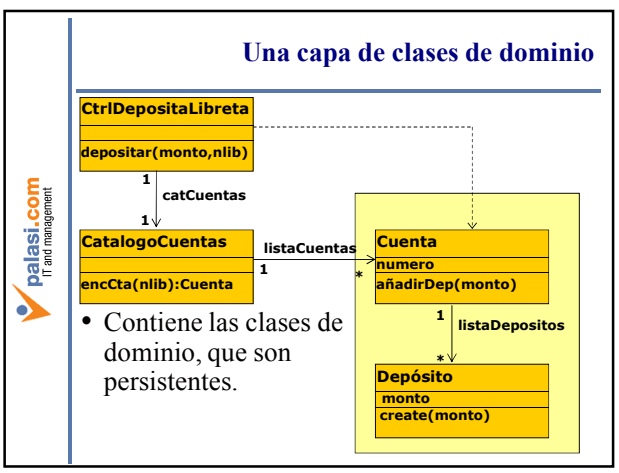


El DCD era así



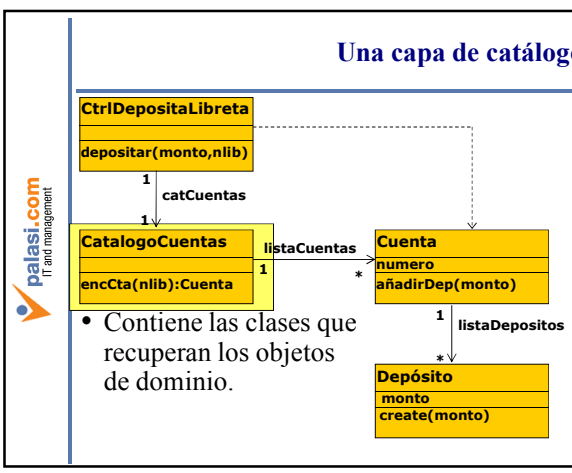
- Aquí se pueden distinguir tres capas.

Una capa de clases de dominio



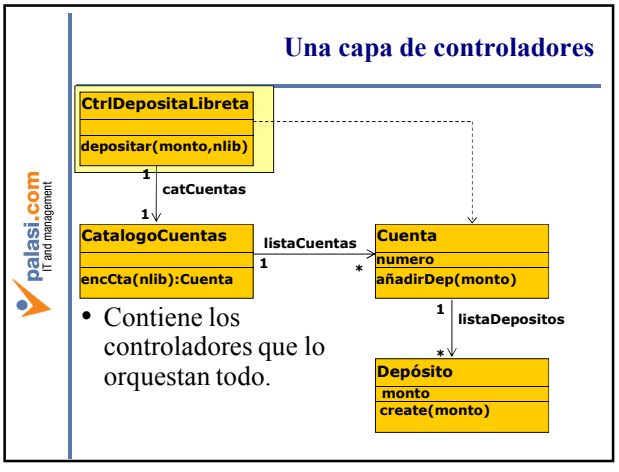
- Contiene las clases de dominio, que son persistentes.

Una capa de catálogos



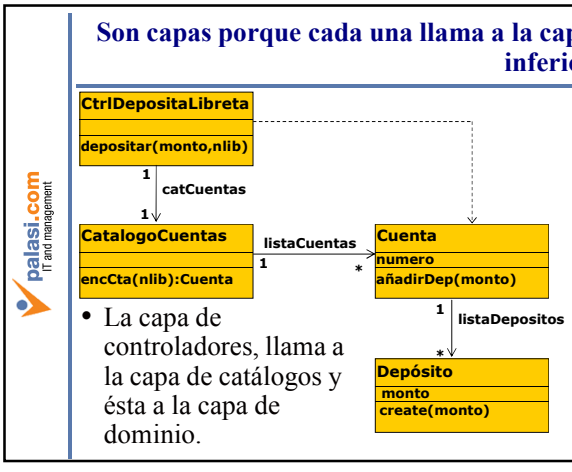
- Contiene las clases que recuperan los objetos de dominio.

Una capa de controladores



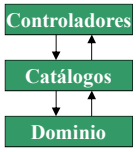
- Contiene los controladores que lo orquestan todo.

Son capas porque cada una llama a la capa inferior



- La capa de controladores, llama a la capa de catálogos y ésta a la capa de dominio.

Esto lo podemos expresar de la siguiente manera.

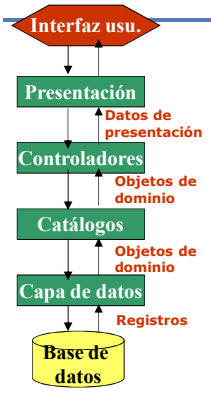


- Sin embargo, mirándolo de forma más detallada, los objetos de dominio son devueltos por los catálogos a los controladores.
- Propiamente, no son una capa, sino los objetos que se pasan entre capas.

Nos queda esto

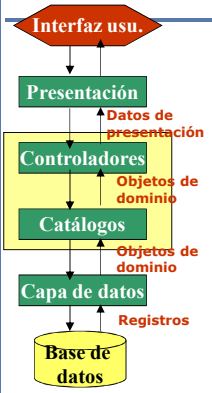
- Sin embargo, estas clases que hemos visto son todas de negocio.
- Como dijimos antes, no tratábamos clases de presentación ni de datos en este curso.
- Sin embargo, ahora las vamos a incorporar para tener un esquema general.

Esquema general



- La capa de presentación llama a la capa de controladores para realizar su trabajo.
- La capa de catálogos llama a la capa de datos para guardar y recuperar los objetos de dominio en la BD.

Esquema general



- La capa de presentación está más allá del alcance de este curso.
- La capa de datos no se programa, sino que se adquiere o es de código abierto. Se le llama capa de persistencia, motor de persistencia, O/R mapping.
- Lo único que hemos visto aquí es la capa de dominio (enmarcada en amarillo).

Acerca de esta presentación

Aurum Solutions

Consultoría internacional en desarrollo de soluciones de software.

Tel: 275-4254
E-mail: info@aurumsol.com

Método de desarrollo (treient el diagrama de casos d'ús)