



CENTRO DE ESTUDIOS FINANCIEROS

VIRIATO, 52	28010 MADRID	914 44 49 20
PONZANO, 15	28010 MADRID	914 44 49 20
G. DE GRÀCIA, 171	08012 BARCELONA	934 15 09 88
ALBORAYA, 23	46010 VALENCIA	963 61 41 99

www.cef.es

info@cef.es

Índice Tema 14

Introducción.

1. Concepto.

- 1.1. Conceptos básicos de la orientación a objetos.
- 1.2. De la programación estructurada a la programación orientada a objetos.
- 1.3. Ventajas e inconvenientes de la orientación a objetos.

2. Elementos: objetos, clases, herencia y métodos.

- 2.1. Objetos, mensajes, clases y métodos.
 - 2.1.1. Tipos de operaciones.
 - 2.1.2. Visibilidad de las propiedades de una clase.
- 2.2. Propiedades de la orientación a objetos.

3. El ciclo de desarrollo del software orientado a objetos.

- 3.1. Introducción.
- 3.2. Análisis orientado a objetos.
- 3.3. Diseño orientado a objetos.

4. Metodologías de desarrollo orientado a objetos.

- 4.1. Métrica 3.
 - 4.1.1. Actividades del proceso de análisis.
 - 4.1.2. Actividades del proceso de diseño.
 - 4.1.3. Técnicas orientadas a objetos para el desarrollo y la gestión.
- 4.2. El proceso unificado de desarrollo del software.
 - 4.2.1. Captura de casos de uso.
 - 4.2.2. Análisis y diseño para realizar los casos de uso.

- 5. El lenguaje de modelización unificado.
 - 5.1. Introducción.
 - 5.2. Los diagramas de UML.
- 6. El modelo CORBA.
 - 6.1. Introducción.
 - 6.2. La arquitectura CORBA.
- 7. Complementos al diseño orientado a objetos.
 - 7.1. Patrones de diseño.
 - 7.2. Programación orientada al aspecto.
 - 7.3. Diseño por contrato.



CENTRO DE ESTUDIOS FINANCIEROS

VIRIATO, 52	28010 MADRID	914 44 49 20
PONZANO, 15	28010 MADRID	914 44 49 20
G. DE GRÀCIA, 171	08012 BARCELONA	934 15 09 88
ALBORAYA, 23	46010 VALENCIA	963 61 41 99

www.cef.es

info@cef.es

TEMA 14

Diseño orientado a objetos. Concepto. Elementos: objetos, clases, herencia, métodos. Ventajas e inconvenientes. El Lenguaje de Modelización Unificado (UML). El modelo CORBA.

INTRODUCCIÓN.

El paradigma de la Orientación a Objetos (OO) -la Programación Orientada a Objetos- apareció a finales de los años 60 pero su uso surgió con fuerza coincidiendo con la segunda crisis del software en los años 80.

Para poner en contexto la aparición industrial de la OO se puede indicar que, en ese momento, la informática distribuida es la arquitectura informática que se impone mediante el uso de ordenadores personales y sistemas y servidores abiertos (interoperabilidad, escalabilidad y operatividad -Guías de Portabilidad X/OPEN y el POSIX-¹). Paralelamente, el paradigma de la programación orientada a objetos coincide temporalmente con la necesidad de reutilizar software desarrollado, mejorar la interacción hombre-máquina (Interfaces Gráficas de Usuario - GUI), así como el boom de la inteligencia artificial que necesitaba de nuevos y más potentes lenguajes de programación. Los servidores departamentales y corporativos abiertos superan los mainframes y el uso de terminales «tontos». Los años 90 y los inicios del siglo XXI no hacen sino incrementar el uso de la POO tanto en arquitecturas de dos o más niveles como en arquitecturas Web.

El objetivo más claro de la OO es producir software de manera más dinámica por medio de la reutilización del software. Este objetivo coincide con las tecnologías CASE y no es casual porque aparecen al mismo tiempo y se comienzan a usar juntas desde el principio. El Proceso Unificado de Desarrollo del Software de Jacobson, Booch y Rumbaugh consagra las herramientas como instrumentos

¹ POSIX (Portable Operating System Interface) fue un trabajo realizado en 1985 por el IEEE (comité de estándares P1003) para obtener un Sistema Operativo Portable. El estándar POSIX de IEEE para interfaces de Sistema Operativo fue elaborado por delegación de ANSI y se construyó a partir de las especificaciones SVID (System V Interface Definition) de AT&T. X/OPEN definió, a partir de 1984, las guías de portabilidad que incluyen un conjunto de normas de interfaces y protocolos que garantizan la portabilidad y al interoperabilidad entre sistemas o aplicaciones de diversos fabricantes.

esenciales en el proceso de desarrollo. En este sentido, conviene tener en cuenta que las herramientas son buenas para automatizar procesos repetitivos, mantener las cosas estructuradas, gestionar grandes cantidades, reinformación y como guía para un camino de desarrollo concreto. Con poco soporte por herramientas, un proceso debe sostenerse sobre gran cantidad de trabajo manual y será, por tanto, menos formal. Esto podría disminuir la productividad del equipo de manera significativa. Las herramientas se desarrollan para automatizar actividades, de manera completa o parcial, para incrementar la productividad y la calidad, y para reducir el tiempo de desarrollo.

De manera general, podemos caracterizar la reutilización del software por un aumento de la productividad, un incremento de la calidad (software más regular y fiable) y una disminución del coste.

De la misma forma que la programación procedural llevó a refinamientos como la programación estructurada, los modernos métodos de diseño de software orientado a objetos incluyen refinamientos como los patrones de diseño (design patterns), el diseño por contrato (design by contract) o los lenguajes de modelado (como UML).

HISTORIA.

Sketchpad de Ivan Sutherland en 1963 fue la primera aplicación pensada bajo el paradigma de la orientación a objetos pero no fue ningún lenguaje de programación OO. La aplicación permitía, por primera vez, en un momento en que el uso de los ordenadores se limitaba a lanzar trabajos batch, utilizar una interfaz hombre-máquina con una máquina PDP-6 de DEC.²

Simula-67 fue el primer lenguaje orientado a objetos. Creado por Ole-Johan Dahl y Kristen Nygaard en la Universidad de Oslo, fue inicialmente concebido para poder realizar simulaciones de diseño de barcos. El problema al que se enfrentaban los diseñadores era a poder realizar simulaciones teniendo en cuenta que cada parte del barco tenía una cierta cantidad de atributos propios y que, en su conjunto, la explosión combinatoria que surgía cuando se cambiaban los valores de esos atributos hacía imposible trabajar con un lenguaje de programación normal para realizar la simulación. Decidieron crear estructuras de datos que agrupasen diferentes tipos de barcos en diferentes tipos de clases de objetos, cada una de las cuales era responsable de su subconjunto de datos y de su comportamiento (como reconfigurar sus datos basándose en los cambios realizados en atributos de otras clases).

Nygaard no paró con Simula y siguió trabajando en un lenguaje llamado Delta en 1975 y refinando las nociones asociadas de clases, registros (records), tipos y procedimientos para conseguir una estructura de un mayor nivel de abstracción, el patrón (pattern). Éste se incorporó por primera vez al lenguaje Beta.

Smalltalk redefinió todos los conceptos ya incluidos en Simula. Creado en primera versión en 1972 por Xerox PARC (Palo Alto Research Center) en el marco de un desarrollo de un ordenador sin teclado y con comunicación completamente gráfica³. Para este proyecto era necesario un lenguaje

² <http://www.sun.com/960710/feature3/sketchpad.html>

³ Las Interfaces Gráficas de Usuario (GUI) siempre han sido uno de los puntos donde los LOO han tenido grandes ventajas sobre los lenguajes clásicos. Definir las diferentes estructuras visuales (ventanas, botones, listas, contenedores, ...) como objetos y sus interacciones como mensajes es una manera más natural y mejor que hacerlo de la manera estructurada.

OO con características dinámicas, es decir, en el cual los objetos se pudiesen crear y modificar a medida que fuesen necesarios en vez de tenerlos todos ya programados estáticamente en el programa. Desafortunadamente, el lenguaje no fue estandarizado por lo que actualmente existen muchos dialectos como SmallTalk-80 o SmallTalk-V.

En 1985, Bertrand Meyer desarrolló Eiffel mejorando Smalltalk para incrementar la productividad de los programadores en el nuevo lenguaje y mejorar la calidad del software. El incremento de productividad venía dado por incluir funcionalidades pre-existentes y muy utilizadas en los lenguajes clásicos (Basic, Pascal, Ada,...). Estos últimos no estaban diseñados para añadir características propias de la OO y, a menudo, su uso producía problemas de compatibilidad de programas y mantenibilidad del código.

Durante la década de los 80, Niklaus Wirth avanzó en los conceptos de Abstracción de Datos y Programación modular. Módulo-2 incluyó ambos conceptos y fue el precursor a Oberon que dio el paso a la OO definitiva incluyendo herencia, clases, etc. En la actualidad, Oberon se le ha renombrado como Component Pascal (Oberon Microsystems 1997). DEC y Olivetti diseñaron y desarrollaron Módulo-3.

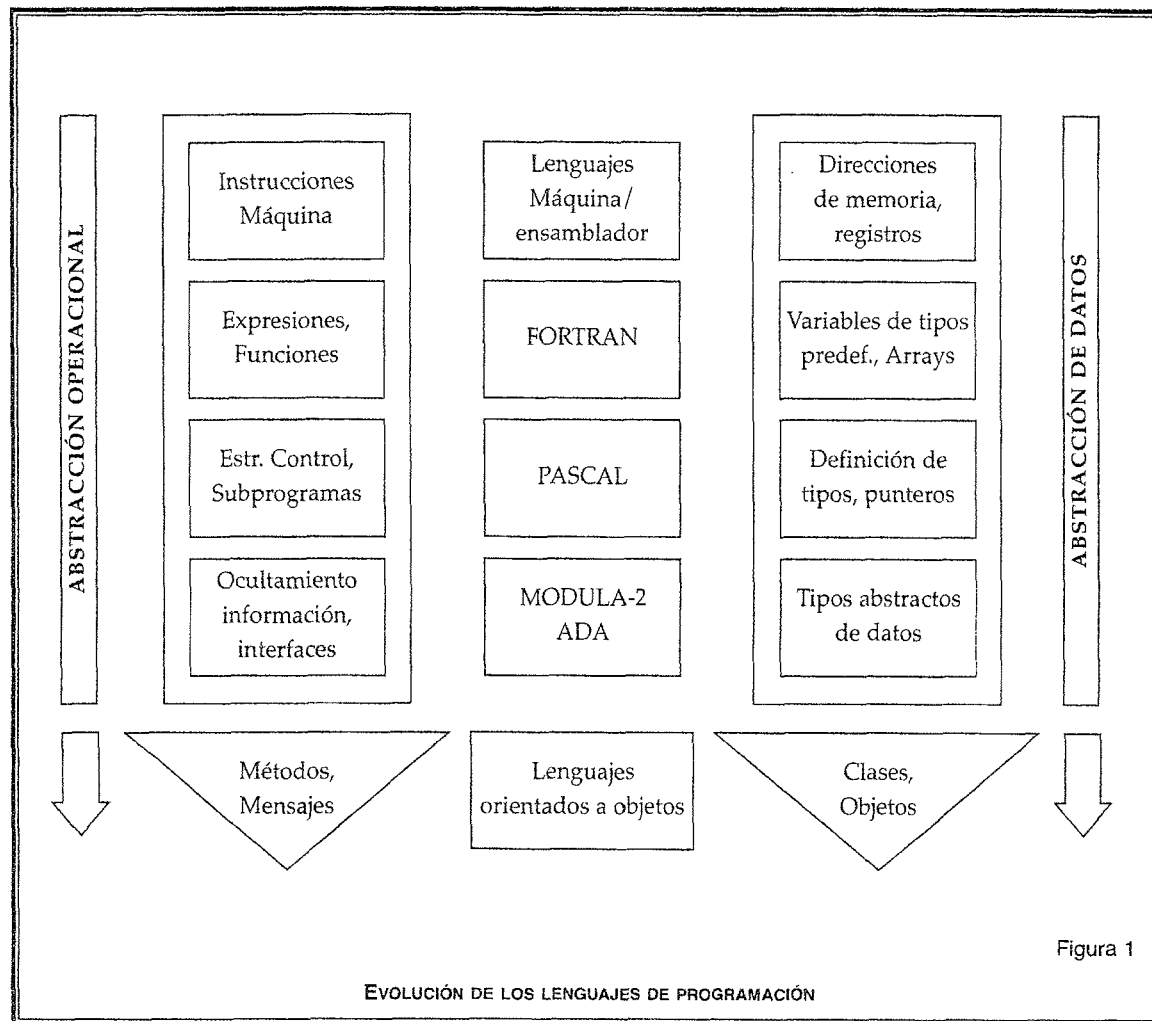
La POO no podría haber tenido el éxito que tuvo en la década de los 80 de no ser por C++. El paso de un lenguaje de programación conocido (C) a otro (C++) permitió que los programas ya existentes no tuviesen que ser re-escritos y que la curva de aprendizaje del nuevo lenguaje fuese menor. En contrapartida, este planteamiento trajo los problemas ya comentados de compatibilidad y malas prácticas. Se trata de un lenguaje compilado, muy portable entre sistemas, con tipado fuerte, múltiples librerías de acceso al sistema operativo donde resida (POSIX), soporta la herencia múltiple, dispone de múltiples entornos de desarrollo visual, no hace limpieza automática de memoria dinámica (garbage collection), admite tratamiento de excepciones. Las mayores ventajas son las inherentes a C (generación de código eficiente, portabilidad y versatilidad) y del uso de las nuevas técnicas de OO. El Objective-C (1984) es un LOO generado para Mac OS X con el que se afrontaron tareas relativas a las librerías dinámicas de GUI.

Otros lenguajes han sido «mejorados» incluyendo características de OO. Así, Microsoft desarrolló el VisualBasic o Borland desarrolló Delphi (incluyendo mejoras en su TurboPascal). La gran ventaja de estos lenguajes fueron los entornos de desarrollo avanzados de que disponen ambos ejemplos (incluyendo generador de pantallas, de código, asistentes, ayuda contextual, debugger, gestor de configuraciones,...). Aparte de los problemas anteriormente expuestos de compatibilidad y mantenibilidad (malas prácticas), se pierde portabilidad ligando el programa a una determinada arquitectura. Por otro lado, ADA tuvo una nueva versión (ADA-95), Lisp tuvo su versión OO con Common Lisp Object System (CLOS) en 1998 e incluso COBOL tuvo su versión en Micro Focus Object COBOL (1998).

Incluso los lenguajes de scripting disponen de características de OO. Tanto Perl como Python disponen de herencia múltiple y polimorfismo. Python (1998) es relativamente nuevo y su sintaxis y semántica se basa en Módulo-3.

En la década de los 90 y principios del siglo XXI, han aparecido nuevos lenguajes e incluso arquitecturas de desarrollo basadas en este paradigma. Java (1995) es uno de los exponentes máximos de esta etapa y la plataforma .NET de Microsoft (2002) es otro. Java es un lenguaje interpretado (semi-compilado) que genera un byte-code interpretado por la máquina virtual (JVM). No permite la he-

rencia múltiple aunque permite múltiples interfaces para una clase. Dispone de garbage collector. A diferencia de C++, todo forma parte de una clase. Su sintaxis y semántica es parecida a C++ lo cual mejora la curva de aprendizaje del lenguaje. Dispone de enlace dinámico y multithread. En el caso de Microsoft, C# es el exponente del LOO.



1. CONCEPTO.

1.1. CONCEPTOS BÁSICOS DE LA ORIENTACIÓN A OBJETOS.

De forma general, se puede considerar que un sistema software bajo el paradigma de la Orientación a Objetos está formado por un conjunto de objetos que cooperan para lograr el resultado final.

Por ejemplo, un sistema de dibujo está formado por objetos como ventanas, botones, una barra de herramientas, una línea, una imagen,...

Para un desarrollador, la programación orientada a objetos significa que debe enfocar su trabajo hacia los datos que trata su sistema y los métodos para manipular esos datos, en vez de pensar en los procesos de modificación de los flujos de datos que existen en el sistema. La programación orientada a objetos es un paradigma opuesto a una visión algorítmica e imperativa (descomposición funcional) del sistema a implementar, es decir, hay un cambio de mentalidad en los programadores y en los ingenieros del software sobre cómo diseñar y construir un sistema software.

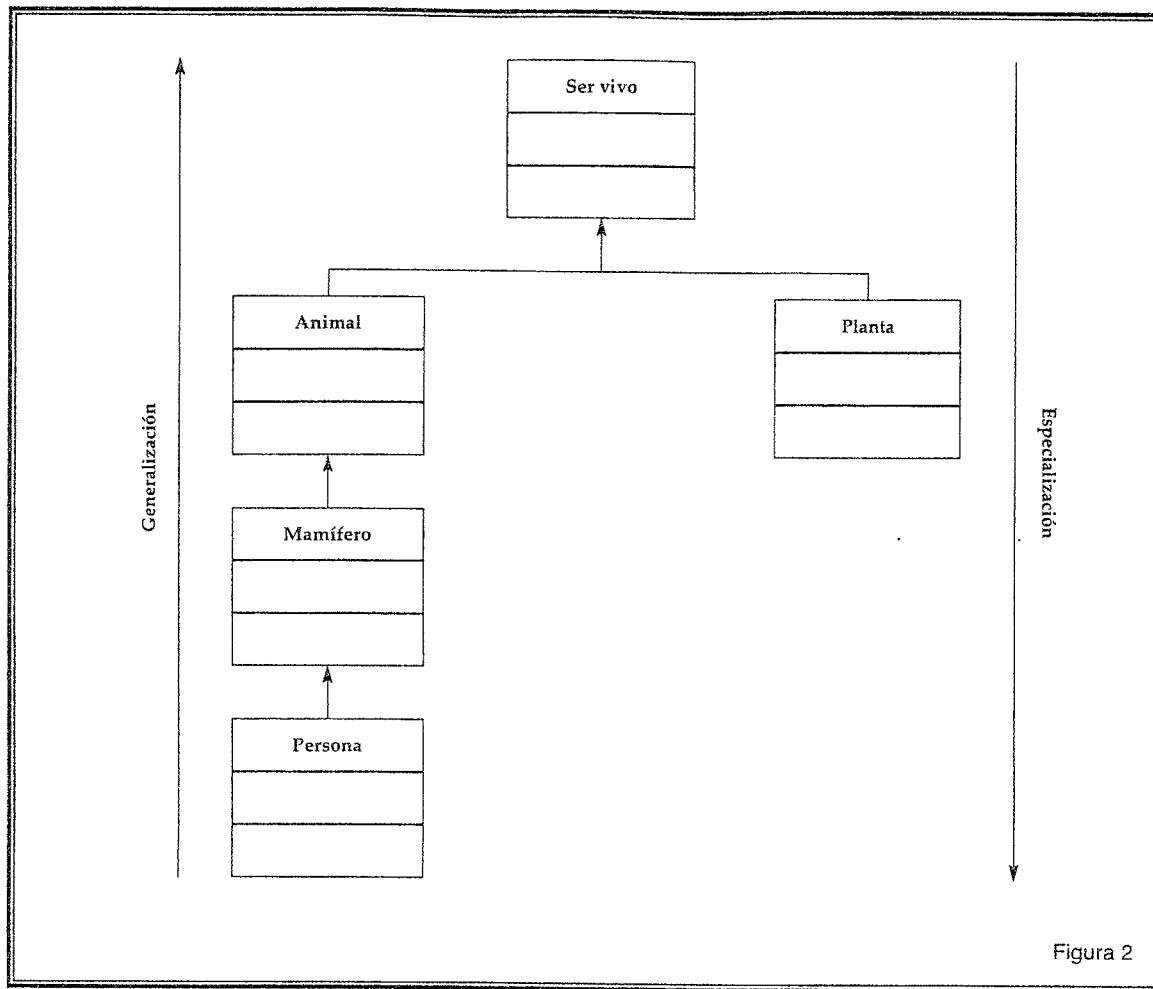
DESCOMPOSICIÓN FUNCIONAL	ORIENTACIÓN A OBJETOS
<pre> programa () { subproceso_1 (); ... subproceso_n (); } </pre>	<pre> graph LR BH[Barra de herramientas] -- dibujar --> C[Círculo] BH -- dibujar --> Cu[Cuadrado] BH -- dibujar --> L[Línea] </pre>
<ul style="list-style-type: none"> • Módulos contruidos alrededor de las operaciones. • Datos globales o distribuidos entre módulos. • Entrada/Proceso/Salida. • Diagramas de Flujo de Datos. 	<ul style="list-style-type: none"> • Módulos contruidos alrededor de las clases. • Clases débilmente acopladas y sin datos globales. • Encapsulación/Mensajes. • Diagramas jerárquicos de clases.

En un sistema orientado a objetos, una clase es una colección de datos y métodos que operan en esos datos. Un objeto es una instancia particular de una determinada clase (un ejemplar de la clase) que tiene vida propia y que mantiene un estado (los datos del objeto se dice que mantienen su estado) durante un cierto tiempo de vida en función de la persistencia de sus datos. Ambos, los datos y los métodos describen el estado y el comportamiento de un objeto, respectivamente. Por lo tanto, los objetos poseen una funcionalidad (las operaciones que son capaces de hacer o los mensajes a los que son capaces de reaccionar) y un estado.

Las clases se organizan en una jerarquía de manera que una subclase puede heredar las propiedades (datos y operaciones) de su superclase o modificarlo si fuese necesario. Una jerarquía de clases siempre tiene una clase raíz la cual tiene siempre propiedades extremadamente genéricas⁴. La jerarquía puede formar un grafo en el caso de que exista la herencia múltiple que consiste en que una clase hereda las propiedades de varias. Algunos lenguajes orientados a objetos no permiten esta última⁵.

⁴ Por ejemplo, la clase Object de Java.

⁵ C++ sí lo permite. Java no pero palía ese problema por medio de la extensión del comportamiento de una clase mediante las interfaces.



1.2. DE LA PROGRAMACIÓN ESTRUCTURADA A LA PROGRAMACIÓN ORIENTADA A OBJETOS.

La Programación Estructurada (imperativa) aporta, básicamente, lo siguiente:

- Estructura de un módulo:
 - Interfaz:
 - Datos de entrada.
 - Datos de salida.
 - Descripción funcionalidad.
 - Implementación:
 - Datos locales.
 - Secuencia de instrucciones.

- Sintaxis del lenguaje:

- Organización del código en bloques de instrucciones. Definición de funciones y procedimientos.
- Extensión del lenguaje con nuevas operaciones. Llamadas a nuevas funciones y procedimientos.

Las ventajas de usar esta aproximación son las siguientes:

- Facilita el desarrollo:

- Se evita la repetición del trabajo.
- Trabajo de programación compartimentado en módulos independientes.
- Diseño top-down: descomposición en subproblemas.

- Facilita el mantenimiento:

- Claridad del código.
- Independencia de los módulos.

- Favorece la reutilización.

Los Tipos Abstractos de Datos (TAD) permiten abstraer datos y operaciones. Un TAD consiste en:

- Una estructura de datos que almacena información para representar un determinado concepto.
- La funcionalidad del TAD obtenida por un conjunto de operaciones que se pueden realizar sobre el TAD.

El lenguaje que soporte TAD debe permitir modularizarlos asociando módulos a tipos de datos. Esto no obliga necesariamente a variar la programación modular que se puede seguir desarrollando de la misma manera en ese lenguaje.

Las ventajas de los TAD son las siguientes:

- Conceptos del dominio de datos reflejados en el código.
- Encapsulamiento: ocultación de la complejidad interna y detalles de los datos y las operaciones.
- Especificación vs. implementación: utilización del tipo de datos independiente de su programación interna.

- Mayor modularidad: también los datos son módulos.
- Mayor facilidad de mantenimiento y reutilización.

La Programación Orientada a Objetos:

- Da un soporte sintáctico explícito para la abstracción de datos.
 - Definición de clases.
 - Funciones explícitamente asociadas a clases.
 - Creación de objetos.
 - Acceso a atributos, invocación de métodos.
- Cambia el punto de vista: los programas son apéndices de los datos.
- Aparece un nuevo concepto: objeto.
 - Objeto = tipo abstracto de datos con estado (atributos) y comportamiento (operaciones) propios.
- Aparece el concepto de jerarquía de tipos, y con él:
 - Herencia de estructura y funcionalidad.
 - Polimorfismo.

1.3. VENTAJAS E INCONVENIENTES DE LA ORIENTACIÓN A OBJETOS.

Las principales ventajas del uso de este paradigma son las siguientes:

- Ventajas de la abstracción de datos + disciplina de programación (ventajas de la programación estructurada).
 - Reutilización de código (reusabilidad).
 - Facilita el mantenimiento y extensión (extensibilidad) de las aplicaciones.
 - Encapsulación y modularidad.
- Potencia del lenguaje: definición de clases, herencia y polimorfismo.
- Reflejar conceptos de problemas reales.
- Los estudios reflejan que es más sencillo de aprender que la descomposición funcional, más natural de usar.

En general, podemos decir que el diseño orientado a objetos ayuda a maximizar la modularidad y la encapsulación porque el sistema se puede descomponer en objetos con unas responsabilidades claramente especificadas. Por lo tanto, los diseños serán más proclives a tener un acoplamiento bajo y una cohesión alta.

- En primer lugar, el acoplamiento es una medida externa a las clases y mide el grado de interdependencia entre módulos. Se debe conseguir que los módulos sean lo más independientes entre sí. En principio, el propio diseño de clases debería favorecer esta medida.
- En segundo lugar, la cohesión es una medida interna a las clases y mide la relación existente entre los elementos de un módulo. Un módulo con alta cohesión realiza una tarea concreta y sencilla. En principio, el comportamiento de la clase debería estar enfocado al estado de datos que debe mantener.

Por otro lado, se mejora la extensibilidad del sistema dado que se tiene la posibilidad de ampliar la funcionalidad de la aplicación de manera sencilla. Si se necesitase un comportamiento no tenido en cuenta bastaría con añadir la clase que diese abrigo a ese conjunto de nueva funcionalidad. En el caso de que hiciese falta que una clase ampliase su comportamiento, bastaría con añadirlo y mantener el resto de su interfaz para evitar problemas de mantenimiento.

Por último, ya se ha destacado la reusabilidad como un principio fundamental de la OO a lo largo del tema. Mediante el uso de la POO podremos reutilizar parte del código para el desarrollo de una aplicación similar. La creación de librerías de componentes facilita el uso en los modernos lenguajes de programación orientada a objetos sin necesidad de recompilar y linkar el código. Simplemente la declaración del componente y su creación es suficiente para su utilización en un nuevo programa.

A pesar de todas las ventajas enumeradas anteriormente, el uso de la OO tiene también inconvenientes que podemos resumir en los siguientes puntos:

- Como en cualquier otro lenguaje, el uso de uno orientado a objetos no evitará malos análisis, diseños o programas. Tal y como hemos visto, la POO ofrece herramientas para minimizar esto pero estas tareas son netamente responsabilidad de los desarrolladores.
- El equipo de trabajo necesita una preparación específica, tanto en el paradigma de OO como en el lenguaje y herramientas que se vayan a usar. Se ha comentado a lo largo del tema que para poder lograr alta productividad es necesario potenciar la reusabilidad del código y la única manera de hacerlo es automatizar la producción y mantenimiento del código y de los diferentes artefactos del sistema.
- Consecuencia del anterior punto es que la curva de aprendizaje de las múltiples librerías de componentes y de las múltiples herramientas así como de las técnicas de Ingeniería del Software necesarias para usar la OO exitosamente es mayor que la simple técnica de codificar todo lo que sea necesario sin reconocer lo existente.
- Los lenguajes OO puros (como Java, que además es interpretado) son poco eficientes y necesitan de arquitecturas hardware relativamente potentes para funcionar correctamente. Los híbridos (como C++, que además es compilado) son mucho más eficientes pero pueden comprometer los principios de la OO.

- La reusabilidad prometida por la OO se ve seriamente comprometida si no hay una apuesta clara por la calidad de los paquetes de componentes desarrollados para ser reutilizados. El control de la calidad externa de los componentes (su funcionalidad) y la calidad interna (la manera de programarlos) será necesaria a lo largo de todo su ciclo de vida.
- Por otro lado, las librerías de componentes que se reusan deben tener una gestión de configuración estricta. Son conocidos los problemas de los modelos de ciclo de vida de Desarrollo Basado en Componentes. La identificación de los componentes adecuados para un determinado sistema y los problemas laterales de un cambio de versión en el sistema deben ser vigilados cuidadosamente.

2. ELEMENTOS: OBJETOS, CLASES, HERENCIA Y MÉTODOS.

2.1. OBJETOS, MENSAJES, CLASES Y MÉTODOS.

Objetos.

Los objetos son módulos que contienen los datos y las instrucciones que manipulan esos datos. Un programa OO consta sólo de objetos que contienen tanto los procedimientos como la estructura de datos. Comparándolo con la programación tradicional, dentro de los objetos residirían los datos de los lenguajes tradicionales y las funciones, instrucciones y subrutinas que operan sobre ellos. Los objetos, por tanto, son entidades que tienen atributos (datos) y formas de comportamiento (procedimientos) particulares. Dentro de un sistema orientado a objetos se representarán todos aquellos relevantes para el Universo de Discurso sobre el que trate el sistema. En este sentido, el diseño orientado a objetos se centrará sobre cualquier cosa real o abstracta sobre la que se tiene una percepción de su individualidad.

Mensajes.

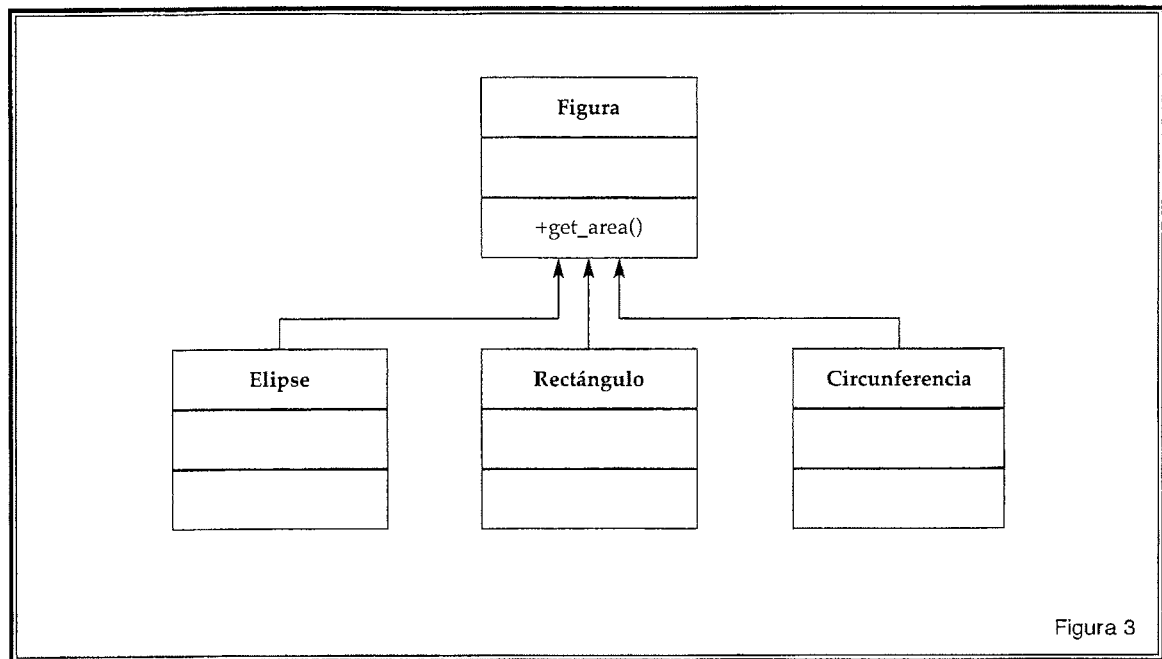
Los objetos tienen la posibilidad de actuar. La actuación sucede cuando un objeto recibe un mensaje, que no es más que una solicitud que le pide que se comporte de alguna forma determinada. El mensaje contiene el nombre del objeto al que va dirigido, el nombre de una operación y, en ocasiones, un grupo de parámetros. Al ejecutar un programa OO, los objetos reciben, interpretan y responden a estos mensajes procedentes de otros objetos. El conjunto de mensajes al que puede responder un objeto se denomina protocolo del objeto.

Clases.

Una clase es el conjunto de métodos y datos que resumen las características comunes de todos los objetos que la componen. Una clase está compuesta por un conjunto de objetos diferentes que pueden actuar de formas muy similares. Cada uno de los objetos individuales pertenecientes a una clase se denomina instancia. Desde el punto de vista de la programación, se puede considerar a una clase como un tipo de los disponibles en el sistema.

Una clase que no tenga instancias se denomina clase abstracta. Una clase abstracta puede servir para declarar las propiedades de un conjunto determinado de clases pero que a ese nivel de abstracción no se pueden concretar. Por ejemplo, la función obtener_area() puede ser una función común a

todos las figuras por lo que se podrá declarar a nivel de la clase Figura pero no se podrá definir porque no existe una función común para obtener el área de todos los tipos de figuras. Se podrá definir al nivel de Elipse, Circunferencia o de Rectángulo.



En C++, una clase es abstracta cuando no define alguna parte de la abstracción de datos que encapsula, como por ejemplo, un procedimiento virtual (lo tiene pero no lo puede definir).

Al conjunto de clases utilizadas para una determinada tarea de programación se le denomina biblioteca de clases. Un componente es un objeto que se puede reutilizar y que puede interactuar con otros objetos. Los componentes se generan como clases que implementan una determinada interfaz genérica que los identificará como componentes y publicará su protocolo de operaciones para su uso en el sistema destino. Los componentes se agrupan en bibliotecas de componentes. La diferencia entre las bibliotecas de clases y las de componentes es que unas están pensadas en el marco de un sistema concreto y las de componentes son la extensión de las de clases para su reutilización en el marco de múltiples sistemas distintos.

Por ejemplo, En .NET Framework, un componente es una clase que implementa la interfaz `System.ComponentModel.IComponent` o que deriva directa o indirectamente de una clase que implementa la interfaz `IComponent`.

Cada objeto puede heredar la estructura de datos y operaciones definidas para los objetos de una clase más amplia a la que pertenece, constituyendo una subclase dentro de ésta. Una de las utilidades de las clases abstractas es la capacidad de heredar propiedades.

Clases y objetos pueden parecer conceptos similares; sin embargo, las clases son un concepto estático definido en el programa fuente, mientras que los objetos son entes dinámicos que ocupan memoria en la ejecución de un programa.

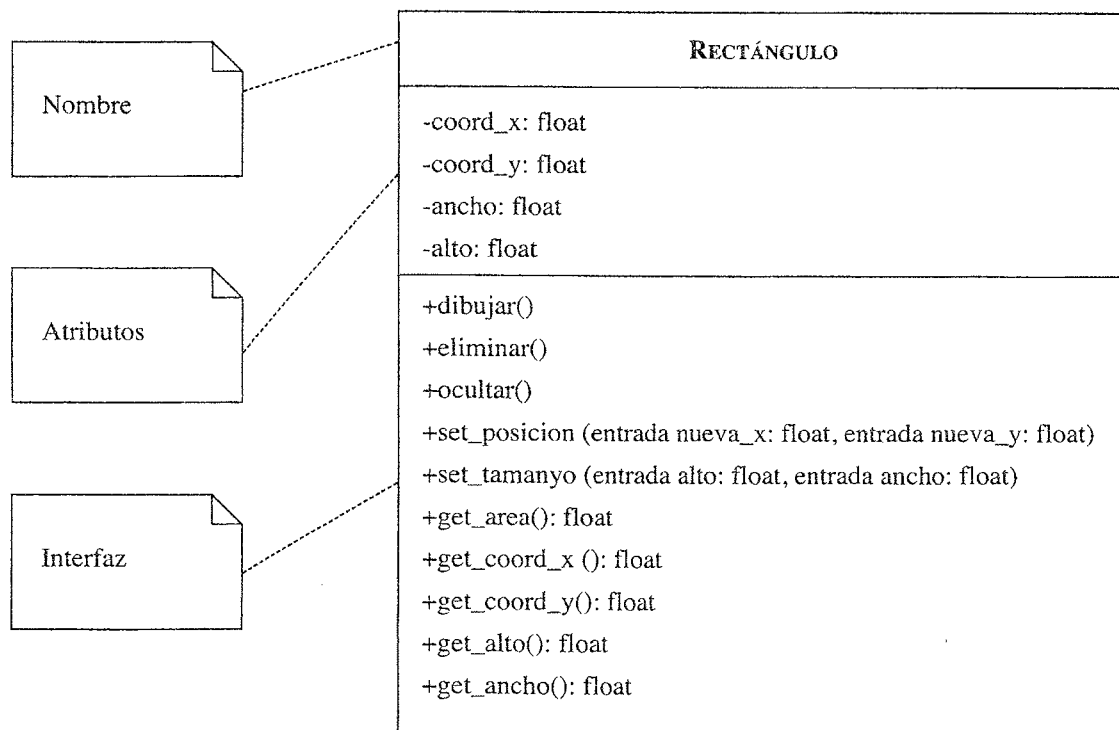
Los objetos se crean cuando se recibe un mensaje solicitando su creación por la clase padre. El nuevo objeto toma sus métodos y datos de su clase padre. Los datos pueden ser de dos tipos: variables de clase, que tiene valores almacenados en una clase y variables de instancia, que tienen valores asociados únicamente con cada instancia u objeto creado a partir de una clase.

Métodos.

El conjunto de procedimientos compartidos que contienen los objetos se denomina métodos. Los métodos definen la forma en la que los datos contenidos en los objetos son manipulados.

Los métodos de un objeto se invocan exclusivamente con el mensaje adecuado, y al ser invocado un método de un objeto, sólo se referirá a la estructura de datos de ese objeto y no a la de otros, aunque sean de la misma clase.

La interfaz de la clase estará definida por el conjunto de métodos que soporta y los mensajes que es capaz de tratar.



2.1.1. Tipos de operaciones.

- Constructor: crea una instancia de la clase.
- Destructor: destruye una instancia.

- Selector (get): selecciona una parte del estado o devuelve una propiedad resultante de la combinación de algunos atributos.
 - Ejemplos: DameCoordX, DameCoordY, DameAncho, DameAlto, Area, etc.
- Modificador (set): modifica una parte del estado según algún criterio.
 - Ejemplos: Mover, CambiarDeTamanho, etc.
- Copiador (copy, clone): copia un objeto en otro.
- Iterador: permite recorrer una estructura de datos (p. ej.: para recorrer una lista).
- Visualizador (view): muestra todo el estado del objeto de una forma elaborada.
 - Ejemplo: dibujar.

2.1.2. Visibilidad de las propiedades de una clase.

- Los atributos y operaciones pueden tener los siguientes tipos de acceso (visibilidad):
 - Público: se pueden acceder desde cualquier clase.
 - Privado: sólo se pueden acceder desde operaciones de la clase.
 - Protegido: sólo se pueden acceder desde operaciones de la clase o de clases derivadas.
- Normas generales:
 - El estado de la clase debe ser privado.
 - Las operaciones que definen la funcionalidad deben ser públicas.
 - Las operaciones que ayudan a implementar parte de la funcionalidad deben ser privadas (si no se utilizan desde clases derivadas) o protegidas (si se utilizan desde clases derivadas).

2.2. PROPIEDADES DE LA ORIENTACIÓN A OBJETOS.

Existe una cierta disputa entre los autores acerca de cuáles son las propiedades fundamentales o las características exactas que debe tener una aproximación orientada a objetos, si bien, generalmente la mayoría de ellos incluyen las que se citan a continuación.

Identidad.

La Identidad significa que los datos se cuantifican en entidades discretas y distintas denominadas objetos.

Los objetos pueden ser cosas concretas, como un fichero en un sistema de ficheros, o conceptuales, como una política de planificación (scheduling) en un sistema operativo de multiprogramación. Cada objeto tiene su propia identidad inherente, es decir, dos objetos con los mismos valores de sus atributos son dos objetos distintos. Por ejemplo, dos bicicletas del mismo modelo y atributos idénticos (color, etc.) son dos objetos diferentes. En el mundo real, los objetos sencillamente existen, pero dentro de un lenguaje de programación cada objeto tiene un identificador único llamado handle, con el que puede ser referenciado unívocamente (su referencia).

Clasificación.

La clasificación se refiere a que los objetos que tienen la misma estructura de datos (atributos), y el mismo comportamiento (operaciones), están agrupados en una clase.

Una clase es una abstracción que describe las propiedades esenciales de los objetos o entidades particulares al dominio de un problema concreto. De esta forma, cada clase describe un conjunto posiblemente infinito de objetos individuales. En este contexto, se dice que un objeto es una instancia de su clase. Los términos objeto e instancia son, por tanto, equivalentes. Cada instancia de la clase tiene su propio valor de cada atributo pero comparte los nombres de atributo y las operaciones con otras instancias de la clase.

Herencia.

La herencia es el mecanismo mediante el cual una clase (subclase) adquiere las propiedades de otra clase jerárquicamente superior (superclase, clase base). La herencia proporciona el mecanismo para compartir automáticamente métodos y datos entre clases, subclases y objetos, y puede ser simple o múltiple, según que una subclase herede los datos y métodos de una sola clase o de más de una.

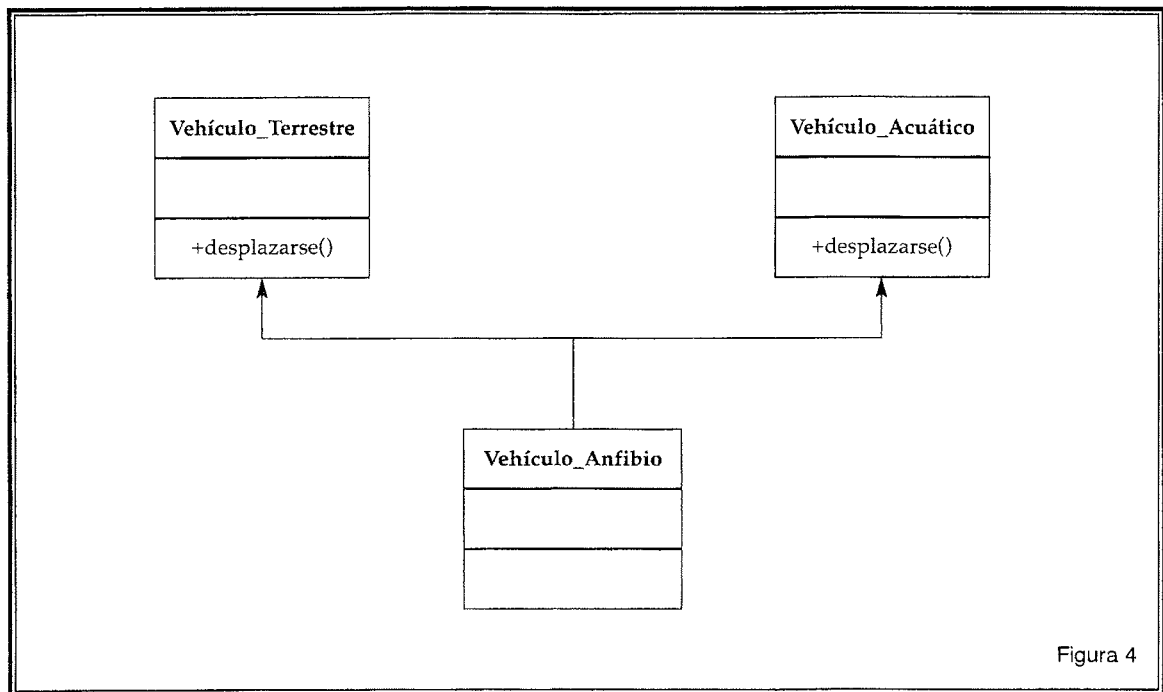
Una clase se puede definir de manera muy amplia y después refinarla en sucesivas subclases. Cada subclase incorpora o «hereda» todas las propiedades de su superclase y añade sus propiedades únicas. Las propiedades de la superclase no necesitan repetirse en cada subclase, por tanto, la posibilidad de reunir las propiedades comunes de varias clases en una superclase común y heredarlas de la misma, reduce en gran medida la repetición en el diseño y la programación y es una de las principales ventajas de la Orientación a Objetos.

La herencia proporciona relaciones entre clases del tipo «es-un» (is-a).

- La herencia de implementación implica que la clase hija hereda la implementación de métodos de la clase padre.
- La herencia de interfaz implica que la clase hija hereda la interfaz (pero no la implementación de las operaciones).
- La herencia de clase combina la herencia de implementación y de interfaz.

Algunos lenguajes tienen palabras clave para distinguir entre herencia de interfaz y clase (ej.: Java), mientras que otros no (ej.: C++).

En el caso de la herencia múltiple, puede haber casos en los que haya problemas de ambigüedad porque, por ejemplo, un método está definido en más de una clase base de la que hereda la subclase.



Supongamos una clase base llamada Vehículo_Terrestre que tiene un método desplazarse() y otra llamada Vehículo_Acuático que tiene el mismo método. Se podría declarar una subclase Vehículo_Anfibio que heredaría ambos comportamientos. Los lenguajes dan soluciones para evitar esos problemas. El más adecuado para este caso supondría ejecutar uno u otro desplazarse en función del estado del vehículo anfibio que se tratase (por ejemplo, si en agua, un_vehiculo_anfibio.Vehículo_Acuático::desplazarse()). Otra posibilidad sería que el compilador detectase los conflictos de nombres para que el programador pueda modificarlos. Otra posibilidad sería establecer una prioridad entre las clases que colisionan para que el sistema pueda resolver el conflicto.

Polimorfismo.

El Polimorfismo es la propiedad por la cual un mismo mensaje puede originar conductas diferentes al ser recibido por objetos diferentes. Es decir, la misma operación puede comportarse de manera diferente para clases diferentes.

El polimorfismo es consecuencia de la herencia. Múltiples funciones pueden estar declaradas con el mismo nombre aunque deben tener alguna característica diferencial una de otra. Podría ser en función de los tipos de los parámetros o de su resultado. También en función del número de parámetros de su prototipo (de la signature de la función). Esto requiere que el lenguaje disponga de enlace dinámico mediante la cual se ejecute el método correcto en tiempo de ejecución (no estáticamente, es decir, cuando se linka el código del programa).

Por tanto, las funciones miembro de una clase base pueden ser sustituidas en una clase derivada mediante la duplicación de su declaración en la clase hija. Por lo tanto, los objetos de las dos clases pueden reaccionar ambos a los mismos mensajes pero lo harán de diferentes maneras.

La implementación específica de una operación para una cierta clase se denomina método. En el caso de los operadores (+, *, =, ...), en un lenguaje orientado a objetos, éstos son polimórficos porque hacen la operación dada en función de los objetos que reciben el mensaje de operación. En un lenguaje de programación orientado a objetos, el propio lenguaje selecciona el método correcto para implementar una operación en función del nombre de la misma y de la clase de objeto sobre la que se aplica.

Encapsulación.

Es el término que se utiliza para expresar que los datos de un objeto sólo pueden ser manipulados mediante los mensajes y métodos predefinidos. Es decir, la encapsulación significa que los datos relativos a algún objeto están almacenados junto con el proceso que crea y manipula esos datos.

Persistencia.

Un objeto en software ocupa un determinado espacio de memoria y existe durante una cierta cantidad de tiempo. La persistencia es la cualidad que se refiere a la permanencia del objeto; es decir, al tiempo durante el cual se le asigna espacio y permanece accesible en la memoria del ordenador.

Por ejemplo, la J2EE de Sun define un estándar para el desarrollo y despliegue de las aplicaciones corporativas y los web services. Se trata de una tecnología de máquina virtual basada en la JVM (Java Virtual Machina). Este framework de desarrollo define, en el Application Server – nivel de negocio, las EJB Entity Beans que son componentes persistentes y que se pueden asociar a entidades de sistemas de bases de datos que mantendrán la persistencia del componente más allá de la vida del proceso que lo contiene.

Extensibilidad.

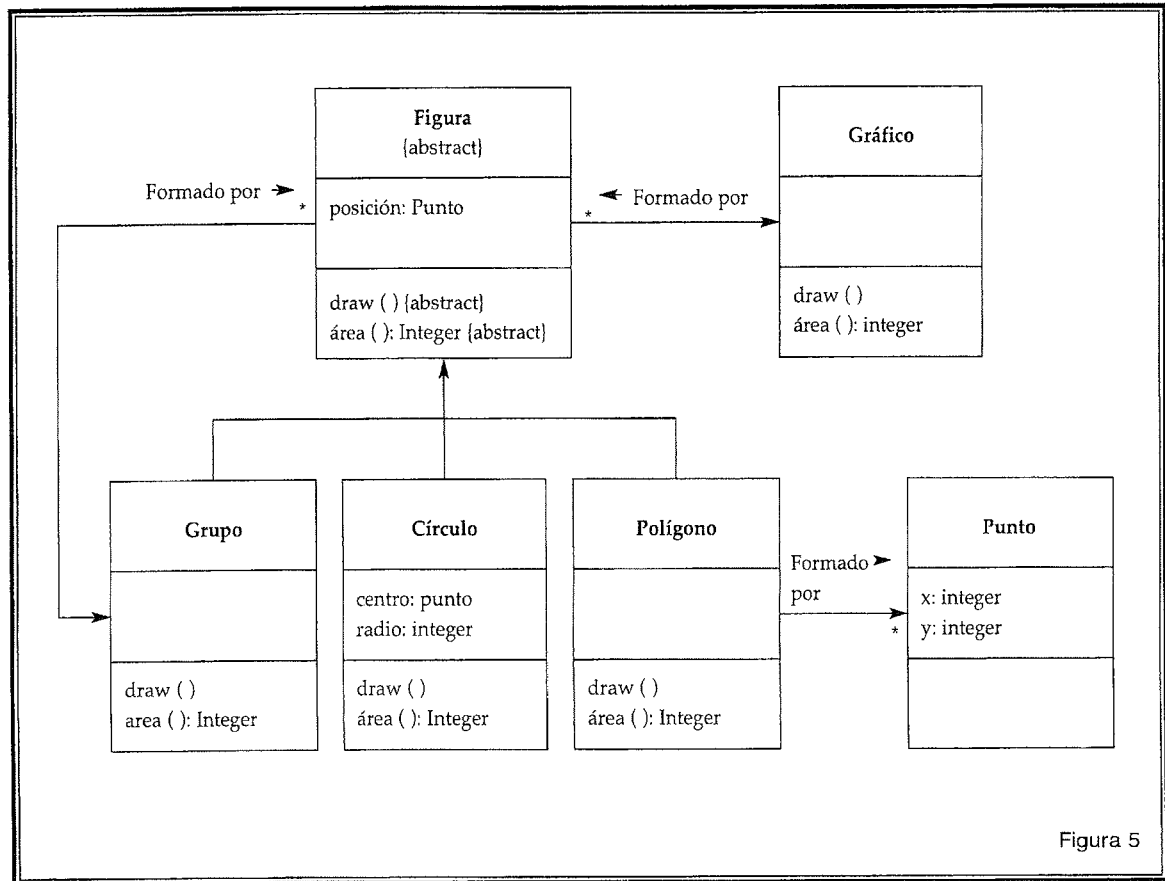
Es la capacidad de un programa para ser fácilmente alterado de forma que pueda tratar con nuevas clases de entrada. Mediante esta propiedad, los objetos pueden ser usados para almacenar y procesar muchos tipos diferentes de datos (gráficos, texto, vídeo, sonido) simplemente añadiendo clases que traten los tipos de datos que sean necesarios.

Reusabilidad.

Es la capacidad de producir componentes reutilizables para otros diseños o aplicaciones. En la OO se consigue de una forma natural mediante el diseño de los componentes para residir en algún repositorio estándar y la utilización de algún repositorio coordinador y herramientas CASE integradas orientadas a objetos (por ejemplo, usando el framework .NET o Java).

Composición/agregación.

Representa el principio de la partición. Una clase B es parte de una clase A. Como se puede ver en el siguiente ejemplo, un gráfico está formado por múltiples figuras.



En este caso, el código de los métodos del agregado (**Gráfico**) delega parte de su implementación en métodos de sus partes (**Figura**). Por ejemplo, el `área()` del **Gráfico** estará dada por la función de `área` que corresponda de la **Figura**. Es decir, la clase **Gráfico** hereda su comportamiento sin estar en una relación de herencia «es-un».

Por otro lado, también ocurre en el ejemplo que el agregado tendrá la responsabilidad de liberar la memoria de sus partes.

Por lo tanto, ya se ha visto que la reutilización de código puede hacerse mediante la herencia o mediante la composición.⁶

⁶ Además del uso de clases y jerarquías para obtener la compartición del comportamiento de los sistemas orientados a objetos existen modelos basados en prototipos que permiten que los objetos en sí mismos sean plantillas en las que se basan otros objetos para obtener su comportamiento. El lenguaje SELF y JavaScript son modelos de programación orientada a objetos basados en prototipos.

	VENTAJAS	INCONVENIENTES
Herencia	<ul style="list-style-type: none"> • Fácil de utilizar. • Fácil de modificar la implementación heredada. 	<ul style="list-style-type: none"> • Establece relaciones estáticas. • Se rompe la encapsulación.
Composición	<ul style="list-style-type: none"> • Establece relaciones dinámicas. • Se mejora la encapsulación. 	<ul style="list-style-type: none"> • Mayor número de objetos. • El comportamiento del sistema depende de las relaciones entre objetos en vez de estar concentrado en una clase.

Parece adecuado adoptar en estos casos los siguientes compromisos:

- Favorecer la composición frente a la herencia de clase.
- Evitar excesivas relaciones entre clases (las clases deben estar débilmente acopladas).
- Evitar jerarquías de clases excesivamente complejas.

Entre todas las propiedades fundamentales de la Orientación a Objetos citadas, la Identificación, Clasificación, Herencia y Polimorfismo son las más importantes y las características esenciales que consideran la gran mayoría de los autores.

3. EL CICLO DE DESARROLLO DEL SOFTWARE ORIENTADO A OBJETOS.

3.1. INTRODUCCIÓN.

Las metodologías de análisis y diseño estructurados se desarrollaron para orientar a los constructores de sistemas de software complejos utilizando algoritmos como bloque fundamental de construcción. De la misma manera, se han desarrollado métodos de diseño orientados a objetos para ayudar a los constructores de software a utilizar con el máximo provecho el poder expresivo de los lenguajes orientados a objetos, utilizando las clases y objetos como elementos básicos de construcción.

El Modelo de Objetos ha influido incluso en las primeras fases del ciclo de desarrollo del software y, de esta forma, podemos hoy día hablar de Análisis Orientado a Objetos (OOA Object Oriented Analysis), Diseño Orientado a Objetos (OOD Object Oriented Design) y Programación Orientada a Objetos (OOP Object Oriented Programming).

El ciclo de desarrollo del software orientado a objetos comienza, en primer lugar, construyendo en el Análisis un modelo que abstrae los aspectos esenciales del dominio del problema, sin tener en cuenta en esta etapa la implementación concreta. Este modelo conceptual contendrá objetos encontrados en el dominio de la aplicación y describirá las propiedades y comportamiento de aquéllos. En segundo lugar, el Diseño añadirá detalles y tomará decisiones que optimicen la implementación. En el diseño, los objetos se describen en términos del dominio del ordenador. Finalmente, el modelo obtenido en el diseño se implementa en un lenguaje de programación, una base de datos y un hardware concretos.

Centraremos la discusión subsiguiente en las etapas de Análisis y Diseño, pero antes conviene destacar los siguientes hechos:

- El Análisis y el Diseño Orientados a Objetos son conceptos evolutivos, no revolucionarios (no hay un cambio entre lo que se hace en una etapa y lo que se hace en la otra). Se trata de refinar el trabajo realizado en cada etapa sucesiva.
- No rompen con los avances conseguidos en el pasado, sino que los mejoran.
- Presentan un marco conceptual para aprovechar al máximo las posibilidades de la Programación Orientada a Objetos.

Sin ese marco conceptual, se podría programar en lenguajes como el C++ o el ADA, etc., pero con un diseño de la aplicación como si fuera a ser implementada en un lenguaje de generaciones anteriores, perdiendo o abusando de la capacidad expresiva de un lenguaje orientado a objetos; y el resultado sería sensiblemente peor que si se hubiera utilizado directamente cualquier lenguaje tradicional, pues el problema de fondo radica en que se está empleando un lenguaje orientado a objetos utilizando solamente una descomposición algorítmica, en lugar de una orientada a objetos.

3.2. ANÁLISIS ORIENTADO A OBJETOS.

Las técnicas de análisis estructurado tradicionales, como las descritas por De Marco, Yourdon, Gane and Searson, etc., centran su interés en el estudio del flujo de datos dentro de un sistema. Por contra, el Análisis Orientado a Objetos enfatiza la construcción de modelos basados en el mundo real, utilizando una perspectiva del mismo basada en las clases y objetos encontrados en el dominio del problema.

Al igual que en los métodos tradicionales, en la etapa Análisis hay que establecer qué es lo que debe hacerse, dejando para etapas posteriores los detalles. El resultado del análisis debe ser una completa comprensión del problema, como preparación al diseño.

Las dos grandes etapas de que consta el Análisis son las siguientes:

1. La descripción o especificación del problema. Esta descripción no debe considerarse inmutable, sino más bien como la base para ir refinando las especificaciones reales.

La especificación del problema debe comprender, en líneas generales, los siguientes aspectos:

- Establecer el ámbito del problema.
- Describir las necesidades y requisitos.
- Describir el contexto de la aplicación.
- Definir los supuestos de que se parte.
- Definir las necesidades de rendimiento del sistema.

En estas especificaciones, el usuario del sistema debe indicar cuáles son obligadas y cuáles se pueden considerar opcionales. También son aceptables especificaciones que definan el rendimiento del sistema y los protocolos de interacción con otros sistemas. Asimismo, otros puntos apropiados pueden ser estándares de Ingeniería del Software, diseño de las pruebas a efectuar, previsión de futuras extensiones, etc.

2. La modelización del Análisis; esto es, las características esenciales deben abstraerse en un modelo.

Las especificaciones expresadas en lenguaje natural tienden a ser ambiguas, incompletas e inconsistentes, sin embargo, el Modelo de Análisis es una representación precisa y concisa del problema, que permite construir una solución. La etapa siguiente de diseño se remitirá a este modelo, en lugar de a las vagas especificaciones iniciales.

El Modelo de Análisis se construye identificando los siguientes elementos:

- Clases y objetos del dominio del problema (estructura estática).
- Interacciones entre los objetos y su secuenciamiento (estructura dinámica).
- Acciones a realizar por el sistema que producen un resultado observable y valioso para los usuarios (estructura funcional).

3.3. DISEÑO ORIENTADO A OBJETOS.

En el enfoque Orientado a Objetos, las etapas de Análisis y Diseño no suelen ser lineales, como preconiza el modelo de ciclo de vida en cascada. Lo usual es que sean procesos iterativos, que se van refinando sucesivamente.

Dentro del nuevo paradigma que supone la Orientación a Objetos, es en la fase de diseño donde más rápida y eficazmente se obtienen beneficios con respecto a la misma fase abordada por los tradicionales métodos funcionales.

El Diseño Orientado a Objetos es aquel que emplea este paradigma y sus conceptos asociados para, partiendo de un cierto análisis cuyos requisitos y funcionalidades se han de cumplir, obtener un modelo del sistema fácilmente implementable en un cierto lenguaje de programación.

Durante el Diseño se llevarán a cabo los siguientes pasos:

1. Decidir la Arquitectura del Sistema. Es decir, identificar los subsistemas de que se compondrá. Esto supone establecer el contexto en el cual se tomarán posteriormente decisiones de detalle.

Un subsistema no es un objeto ni una función, sino un paquete de clases, asociaciones, operaciones y restricciones que están interrelacionados. Normalmente, un subsistema identifica una funcionalidad similar, la misma localización física, la ejecución en una misma clase de hardware, etc. Con respecto al resto del sistema, cada subsistema se identifica por los servicios que proporciona y debe tener una interfaz bien definida con el exterior, de forma que todas las interacciones con otros subsistemas y el flujo de información hacia dentro y hacia fuera del subsistema estén especificados de manera completa y precisa.

2. Identificar las situaciones de concurrencia inherentes al problema.

En el modelo generado en el análisis todos los objetos son concurrentes, pero no ocurre así en la práctica, puesto que un procesador soporta los objetos que no pueden estar activos a la vez. Por lo tanto, un objetivo importante del diseño es identificar qué objetos deben estar activos concurrentemente y cuáles son mutuamente exclusivos.

3. Asignar los subsistemas a los recursos de proceso con los que se cuenta.

Cada subsistema concurrente debe ser asignado a una unidad de hardware, ya sea ésta un procesador o una unidad funcional especializada. Por tanto, en el diseño habrá que:

- Estimar las necesidades de rendimiento y los recursos necesarios para satisfacerlas.
- Elegir la implementación hardware y software de cada subsistema.
- Y asignar los subsistemas de software a los procesadores de tal manera que se cumplan los requisitos de rendimiento y se minimice la comunicación entre procesadores.

4. Elegir una aproximación para la gestión del almacenamiento de datos.

En general, cada almacén de datos del sistema puede combinar estructuras de datos, ficheros y bases de datos implementados, bien en memoria o en dispositivos de almacenamiento secundario. Las diferentes clases de almacenes de datos dan lugar a diversos compromisos entre coste, tiempo de acceso, capacidad y fiabilidad, debiéndose optar por la más apropiada.

Así, por ejemplo, los ficheros son una forma barata, sencilla y permanente de almacenar datos, pero las operaciones con ellos son de bajo nivel y, consecuentemente, la aplicación debe incluir código adicional para proporcionar un nivel de abstracción adecuado. Por contra, las bases de datos intentan mantener en memoria datos que son frecuentemente accedidos, para lograr una combinación lo más óptima posible entre coste y rendimiento del almacenamiento en memoria y disco. Además, tienen otras ventajas como: el proporcionar el acceso a múltiples usuarios a grandes niveles de detalle, la eficiencia en la gestión, las facilidades de recuperación después de un fallo, el que son transportables a diferentes plataformas de hardware y sistema operativo, etc.

Una cuestión importante en el diseño es cuándo parar. Es una cuestión difícil, y es posible llegar a un estado de diseño por exceso, en donde se deciden detalles que sería más conveniente decidir en la implementación. La regla puede ser: «Parar el proceso de diseño cuando los elementos sean tan sencillos que ya no admiten una descomposición posterior».

Además de los consejos ya apuntados con respecto a la composición y la herencia, se puede apuntar que, en general, las características más importantes que presenta el Diseño Orientado a Objetos son las siguientes:

- Modularidad: las clases se identifican perfectamente con el concepto de módulo.
- Ocultación de implementación: la clase aporta la ocultación de detalles de implementación mediante la separación de la interfaz de clase y de su implementación.

- Acoplamiento débil: las clases se diseñan como colecciones de datos y operaciones permitidas sobre las mismas. Por tanto, los operadores que aparecen en la interfaz son los únicos que pueden acceder o alterar los datos internos a la clase, lo que lleva a que exista un menor acoplamiento entre clases.
- Cohesión fuerte: una clase es, de forma natural, un módulo con mucha cohesión puesto que modela claramente una cierta entidad.
- Abstracción: abstrae la especificación de una entidad de los detalles de implementación.
- Extensibilidad: gracias a los mecanismos de herencia, los diseños orientados a objetos son fácilmente extensibles por dos caminos: por la relación de herencia y por el polimorfismo.
- Integrable: el resultado de un diseño orientado a objetos produce resultados que facilitan la integración de trozos individuales en un diseño global.
- Reusabilidad: el paradigma de la Orientación al Objeto combina técnicas de diseño y características de los lenguajes de programación para facilitar una base sólida para la reutilización de módulos software.

4. METODOLOGÍAS DE DESARROLLO ORIENTADO A OBJETOS.

4.1. MÉTRICA 3.

La metodología Métrica versión 3 ofrece a las Organizaciones un instrumento útil para la sistematización de las actividades que dan soporte al ciclo de vida del software dentro del marco que permite alcanzar los siguientes objetivos:

- Proporcionar o definir SI que ayuden a conseguir los fines de la Organización mediante la definición de un marco estratégico para el desarrollo de los mismos.
- Dotar a la Organización de productos software que satisfagan las necesidades de los usuarios dando una mayor importancia al análisis de requisitos.
- Mejorar la productividad de los departamentos TIC permitiendo una mayor capacidad de adaptación a los cambios y teniendo en cuenta la reutilización en la medida de lo posible.
- Facilitar la comunicación y entendimiento entre los distintos participantes en la producción del software a lo largo del ciclo de vida del proyecto, teniendo en cuenta su papel y responsabilidad, así como las necesidades de todos y cada uno de ellos.
- Facilitar la operación, mantenimiento y uso de los productos software obtenidos.

El Consejo Superior de Informática y para el Impulso de la Administración Electrónica (CSI-yAE) dirige la política TIC del Gobierno. Uno de sus objetivos es unificar el desarrollo de sistemas mediante el patrocinio y desarrollo de, entre otros:

- Metodología de desarrollo (Métrica).
- Plan General de Garantía de Calidad (PGGC).
- Metodología de Análisis y Gestión de Riesgos (Magerit).

Métrica3 contempla el paradigma de Orientación a Objetos y propone actividades específicas dentro de sus procesos de desarrollo y gestión. De la misma forma, propone técnicas específicas para el soporte de dichas actividades.

Las técnicas de modelado están basadas en el Lenguaje Unificado de Modelado (UML) y su ejecución en el ciclo de vida de desarrollo es similar a la que se verá para el Proceso Unificado de Desarrollo del Software. Métrica añade la posibilidad de utilizar el paradigma de Análisis Estructurado y detalla cuatro interfaces para la gestión de proyecto, la seguridad, la gestión de la configuración y el aseguramiento de la calidad.

4.1.1. Actividades del proceso de Análisis.

En el caso de un análisis orientado a objetos, se elaboran el modelo de clases y el de interacción de objetos mediante el análisis de los casos de uso.

- En la actividad Definición del Sistema (ASI.1) se efectúa una descripción del sistema, delimitando su alcance y estableciendo las interfaces con otros sistemas e identificando a los usuarios representativos.

La tarea Determinación del Alcance del Sistema (ASI.1.1) delimita el sistema de información ⁷ utilizando como punto de partida, en el caso de análisis orientado a objetos, estableciendo el contexto del sistema a partir del modelo de negocio obtenido en el proceso de Estudio de Viabilidad del Sistema (EVS) y, además, opcionalmente el modelo de dominio. El modelo de negocio especifica los procesos a los que se quiere dar repuesta en el sistema de información, en forma de casos de uso de alto nivel, y el subconjunto de objetos del dominio requerido para ello. Para lograrlo se usan las técnicas de Casos de Uso y de Diagrama de Clases.

- En la actividad Establecimiento de Requisitos (ASI.2) se lleva a cabo la definición, análisis y validación de los requisitos a partir de la información facilitada por el usuario, completándose el catálogo de requisitos obtenido en la actividad Definición del Sistema (ASI.1).

Para ello, en la tarea Obtención de Requisitos (ASI.2.1) se utilizarán los Casos de Uso como técnica de especificación válida para identificar requisitos. Cabe destacar que en Métrica 3, este paso puede ser utilizado también opcionalmente en el caso de que se utilizase un Análisis Estructurado, como una técnica más para la educación de requisitos. Se detallarán Actores, Casos de Uso y breve descripción de cada caso de uso.

La tarea Especificación de Casos de Uso (ASI.2.2) desarrollará el escenario de cada caso de uso identificado anteriormente:

- Descripción del escenario, es decir, cómo un actor interactúa con el sistema, y cuál es la respuesta obtenida.

⁷ El diagrama de contexto del sistema para el análisis estructurado.

- Precondiciones y poscondiciones.
- Identificación de interfaces de usuario.
- Condiciones de fallo que afectan al escenario, así como la respuesta del sistema (escenarios secundarios).

En escenarios complejos, es posible utilizar como técnica de especificación los diagramas de transición de estados, así como la división en casos de uso más simples, actualizando el modelo de casos de uso.

- En la actividad Identificación de Subsistemas de Análisis (ASI.3) se lleva a cabo la descomposición del sistema en subsistemas que faciliten el análisis posterior.

Tanto en Determinación de Subsistemas de Análisis (ASI.3.1) como en Integración de Subsistemas de Análisis (ASI.3.2) se describen los subsistemas de análisis y las interfaces entre subsistemas. Se usará la técnica de Diagrama de Paquetes.

Las siguientes dos actividades son específicas para el uso de la Orientación en el proceso de Análisis de Métrica 3 y no son compatibles con las dos relativas al Análisis Estructurado.

- La actividad Análisis de los Casos de Uso (ASI.4) tiene como objetivo identificar las clases cuyos objetos son necesarios para realizar un caso de uso y describir su comportamiento mediante la interacción de dichos objetos.

Esta actividad se lleva a cabo para cada uno de los casos de uso contenidos en un subsistema de los definidos en la actividad Identificación de Subsistemas de Análisis (ASI 3). Las tareas de esta actividad no se realizan de forma secuencial sino en paralelo, con continuas realimentaciones entre ellas y con las realizadas en las actividades Establecimiento de Requisitos (ASI 2), Identificación de Subsistemas de Análisis (ASI 3), Análisis de Clases (ASI 5) y Definición de Interfaces de Usuario (ASI 8).

La tarea Identificación de Clases Asociadas a un Caso de Uso (ASI.4.1) comienza a identificar los objetos necesarios para realizar el caso de uso en función de la información que se tienen del mismo. Para ello se emplea la técnica del Diagrama de Clases.

A partir del estudio del caso de uso, se extrae una lista de objetos candidatos a ser clases. Es posible que, inicialmente, no se disponga de la información necesaria para identificar todas, por lo que se hace una primera aproximación que se va refinando posteriormente, durante esta actividad y en el proceso de diseño. Además, algunos de los objetos representan mejor la información del sistema si se les identifica como atributos en vez de como clases. Para poder diferenciarlas, es necesario estudiar el comportamiento de esos objetos en el diagrama de interacción y además se debe tener en cuenta una serie de reglas, como puede ser el suprimir palabras no pertinentes, con significados vagos o sinónimos.

Las clases que se identifican en esta tarea pueden ser:

- Clases de Entidad (representan la información manipulada en el caso de uso).
- Clases de Interfaz de Usuario (se utilizan para describir la interacción entre el sistema y sus actores. Suelen representar abstracciones de ventanas, interfaces de comunicación, formularios, etc.).

- Clases de Control (son responsables de la coordinación, secuencia de transacciones y control de los objetos relacionados con un caso de uso).

Una vez definidas cada una de las clases, se incorporan al modelo de clases de la actividad Análisis de Clases (ASI 5), donde se identifican sus atributos, responsabilidades y relaciones.

La tarea Descripción de la Interacción de Objetos (ASI.4.2) complementa la anterior como ya se ha comentado. Se usan para ello los Diagramas de Interacción que contiene instancias de los actores participantes, objetos y la secuencia de mensajes intercambiados entre ellos. Estos diagramas pueden ser tanto de secuencia como de colaboración y su uso depende de si se quieren centrar en la secuencia cronológica o en cómo es la comunicación entre los objetos.

- La actividad Análisis de Clases (ASI.5) describirá cada una de las clases que han surgido, identificando las responsabilidades que tienen asociadas, sus atributos y las relaciones entre ellas.

Teniendo en cuenta las clases identificadas en la actividad Análisis de los Casos de Uso (ASI.4) se elabora el modelo de clases para cada subsistema. A medida que avanza el análisis, dicho modelo se va completando con las clases que vayan apareciendo, tanto del estudio de los casos de uso, como de la interfaz de usuario necesaria para el sistema de información.

TAREA	PRODUCTOS	TÉCNICAS Y PRÁCTICAS	PARTICIPANTES
ASI 5.1 Identificación de responsabilidades y atributos	<ul style="list-style-type: none"> • Modelo de clases de análisis. • Comportamiento de clases de análisis. 	<ul style="list-style-type: none"> • Diagrama de clases. • Diagrama de transición de estados. 	<ul style="list-style-type: none"> • Analistas.
ASI 5.2 Identificación de asociaciones y agregaciones.	<ul style="list-style-type: none"> • Modelo de clases de análisis. 	<ul style="list-style-type: none"> • Diagrama de clases. 	<ul style="list-style-type: none"> • Analistas.
ASI 5.3 Identificación de generalizaciones.	<ul style="list-style-type: none"> • Modelo de clases de análisis. 	<ul style="list-style-type: none"> • Diagrama de clases. 	<ul style="list-style-type: none"> • Analistas.

- Por último, en la actividad Análisis de Consistencia y Especificación de Requisitos (ASI.9) aseguran la calidad de los distintos modelos generados y que los usuarios y Analistas tienen el mismo concepto del sistema. Para ello, se llevan a cabo las siguientes acciones:
 - Verificación de la calidad técnica de cada modelo.
 - Aseguramiento de la coherencia entre los distintos modelos.
 - Validación del cumplimiento de los requisitos.

Se requiere una herramienta de apoyo para realizar el análisis de consistencia.

La Especificación de Requisitos Software se convierte en la línea base para los procesos posteriores del desarrollo del software, de modo que cualquier petición de cambio en los requisitos que pueda surgir posteriormente, debe ser evaluada y aprobada.

4.1.2. Actividades del proceso de Diseño.

En el caso de Diseño Orientado a Objetos, conviene señalar que el diseño de la persistencia de los objetos se lleva a cabo sobre bases de datos relacionales, y que el diseño detallado del sistema de información se realiza en paralelo con la actividad de Diseño de la Arquitectura de Soporte (DSI.2), y se corresponde con las siguientes actividades:

- Diseño de la Arquitectura de Soporte (DSI 2), que incluye el diseño detallado de los subsistemas de soporte, el establecimiento de las normas y requisitos propios del diseño y construcción, así como la identificación y definición de los mecanismos genéricos de diseño y construcción.
- Diseño de Casos de Uso Reales (DSI 3), con el diseño detallado del comportamiento del sistema de información para los casos de uso, el diseño de la interfaz de usuario y la validación de la división en subsistemas.

En esta actividad se especifica el comportamiento del sistema de información para un caso de uso mediante objetos o subsistemas de diseño que interactúan, y determinar las operaciones de las clases e interfaces de los distintos subsistemas de diseño.

Para ello, una vez identificadas las clases participantes dentro de un caso de uso, es necesario completar los escenarios que se recogen del análisis, incluyendo las clases de diseño que correspondan y teniendo en cuenta las restricciones del entorno tecnológico, esto es, detalles relacionados con la implementación del sistema. Es necesario analizar los comportamientos de excepción para dichos escenarios. Algunos de ellos pueden haber sido identificados en el proceso de análisis, aunque no se resuelven hasta este momento. Dichas excepciones se añadirán al catálogo de excepciones para facilitar las pruebas.

Algunos de los escenarios detallados requerirán una nueva interfaz de usuario. Por este motivo es necesario diseñar el formato de cada una de las pantallas o impresos identificados.

- Diseño de Clases (DSI 4), con el diseño detallado de cada una de las clases que forman parte del sistema (provenientes del modelo de clases lógico proveniente del análisis), sus atributos, operaciones, relaciones y métodos, y la estructura jerárquica del mismo. En el caso de que sea necesario, se realiza la definición de un plan de migración y carga inicial de datos.

Se identifican las clases de diseño, que denominamos clases adicionales, en función del estudio de los escenarios de los casos de uso, que se está realizando en paralelo en la actividad Diseño de Casos de Uso Reales (DSI 3), y aplicando los mecanismos genéricos de diseño que se consideren convenientes por el tipo de especificaciones tecnológicas y de desarrollo. Entre ellas se encuentran clases abstractas, que integran características comunes con el objetivo de especializarlas en clases derivadas. Se diseñan las clases de interfaz de usuario, que provienen del análisis. Como consecuencia del estudio de los escenarios secundarios que se está realizando, pueden aparecer nuevas clases de interfaz.

También hay que considerar que, por el diseño de las asociaciones y agregaciones, pueden aparecer nuevas clases, o desaparecer incluyendo sus atributos y métodos en otras, si se considera conveniente por temas de optimización.

La jerarquía entre las clases se va estableciendo a lo largo de esta actividad, a medida que se van identificando comportamientos comunes en las clases, aunque haya una tarea propia de diseño de la jerarquía.

Otro de los objetivos del diseño de las clases es identificar para cada clase, los atributos, las operaciones que cubren las responsabilidades que se identificaron en el análisis, y la especificación de los métodos que implementan esas operaciones, analizando los escenarios del Diseño de Casos de Uso Reales (DSI 3). Se determina la visibilidad de los atributos y operaciones de cada clase, con respecto a las otras clases del modelo.

Una vez que se ha elaborado el modelo de clases, se define la estructura física de los datos correspondiente a ese modelo, en la actividad Diseño Físico de Datos (DSI 6).

- En la actividad Verificación y Aceptación de la Arquitectura del Sistema (DSI7) se garantiza la calidad de las especificaciones del diseño del sistema de información y la viabilidad del mismo.

En concreto, es importante destacar la tarea Análisis de Consistencia de las Especificaciones de Diseño (DSI.7.2) porque tiene verificaciones de consistencia específicas para el diseño orientado a objetos.

4.1.3. Técnicas Orientadas a Objetos para el desarrollo y la gestión.

- Casos de Uso.
- Diagrama de Clases.
- Diagrama de Componentes.
- Diagrama de Despliegue.
- Diagrama de Transición de Estados.
- Diagramas de Interacción: Secuencia y Colaboración.
- Diagrama de Paquetes.
- Técnica de Gestión de Proyectos: Staffing Size. Es un conjunto de métricas para estimar el número de personas necesarias en un desarrollo Orientado a Objetos, y para determinar el tiempo de su partición en el mismo.

4.2. EL PROCESO UNIFICADO DE DESARROLLO DEL SOFTWARE.

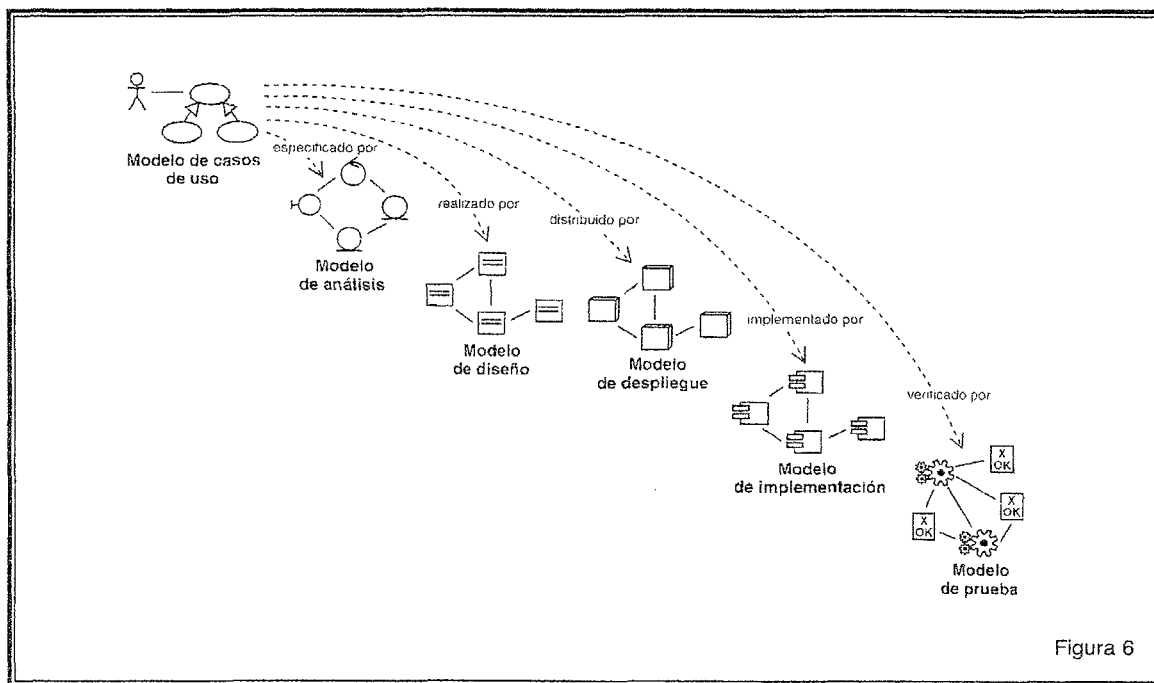
Es un conjunto de actividades necesarias para transformar los requisitos de un usuario en un sistema software. Está basado en componentes, es decir, que en su construcción se usan componentes software interconectados a través de interfaces bien definidas. Utiliza el estándar de modelado visual UML (Lenguaje Unificado de Modelado) y se sostiene sobre tres ideas básicas:

- Dirigido por casos de uso, lo que indica que todo el proceso está basado en las funcionalidades cuyo resultado es importante y que el sistema proporciona a los usuarios (los requisitos del sis-

tema). Los casos de uso no sólo sirven para especificar los requisitos del sistema sino que guían el proceso de desarrollo en su generalidad (análisis, diseño, implementación y prueba).

- Centrado en la arquitectura indica que completa todas las vistas que engloban a todo el sistema en construcción. La arquitectura surge de las necesidades de la empresa (que se reflejan en los casos de uso). También se ve influida por otros factores como la plataforma en la que tienen que funcionar el software (hardware, sistema operativo, SGBD, protocolos de red,...), los bloques de construcción reutilizables, consideraciones de implantación, sistemas heredados y requisitos no funcionales (rendimiento, fiabilidad,...).
- Proceso iterativo e incremental en el sentido de que el ciclo de vida que sigue el proceso está basado en la iteración de flujos de trabajo (Modelado del Negocio, Requisitos, Análisis, Diseño, Implementación, Pruebas, Despliegue, Gestión del cambio y configuraciones, Gestión del Proyecto, Entorno) y en los incrementos del producto (el crecimiento del sistema).

Para desarrollar el producto final del sistema de información, los desarrolladores necesitan todas las representaciones del producto:



- Un modelo de casos de uso, con todos los casos de uso y su relación con los usuarios.
- Un modelo de análisis, con dos propósitos: refinar los casos de uso con más detalle y establecer la asignación inicial de funcionalidad del sistema a un conjunto de objetos que proporcionan el comportamiento.
- Un modelo de diseño que define:
 - La estructura estática del sistema en la forma de subsistemas, clases e interfaces y
 - Los casos de uso reflejados como colaboraciones entre los subsistemas, clases e interfaces.

- Un modelo de implementación, que incluye componentes (que representan al código fuente) y la correspondencia de las clases con los componentes.
- Un modelo de despliegue (también llamado de distribución) que define los nodos físicos (ordenadores) y la correspondencia de los componentes con esos nodos.
- Un modelo de prueba, que describe los casos de prueba que verifican los casos de uso.
- Y una representación de la arquitectura.
- El sistema también debe tener un modelo del dominio o modelo del negocio que describe el contexto del negocio en el que se halla el sistema.

Todos estos modelos están relacionados. Juntos representan al sistema como un todo. Los elementos de un modelo poseen dependencias de traza hacia delante y atrás, mediante relaciones hacia otros modelos. Por ejemplo, podemos hacer el seguimiento de un caso de uso (en el modelo de casos de uso) hasta la realización del caso de uso (en el modelo de diseño) y hasta un caso de prueba (en el modelo de prueba). La trazabilidad facilita la comprensión y el cambio.

4.2.1. Captura de casos de uso.

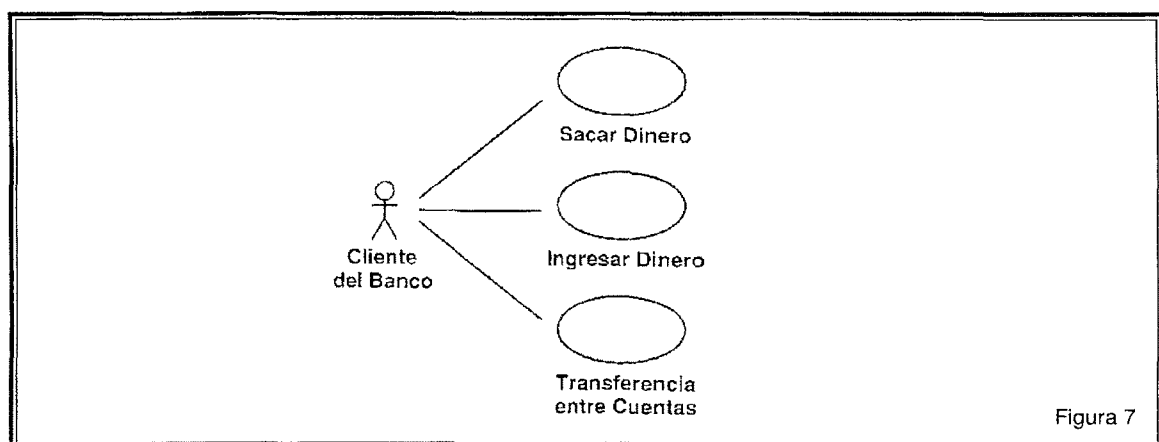
Durante el flujo de trabajo de los requisitos identificamos las necesidades de usuarios y clientes como requisitos. Los requisitos funcionales se expresan como casos de uso en el modelo de casos de uso, y los demás requisitos (no funcionales) o bien se adjunta a los casos de uso a los que afectan, o bien se guardan en una lista aparte (el catálogo de requisitos) o se describen de alguna otra manera.

Por lo tanto, el modelo de casos de uso representa requisitos funcionales.

Los actores son el entorno del sistema. No todos los actores representan a personas. Pueden ser actores otros sistemas o hardware externo que interactuará con el sistema. Los actores se comunican con el sistema mediante el envío y recepción de mensajes hacia y desde el sistema, según esté, lleva a cabo los casos de uso.

Un caso de uso especifica una secuencia de acciones, incluyendo variantes, que el sistema puede llevar a cabo, y que producen un resultado observable de valor para un actor concreto.

Por ejemplo, modelo de casos de uso para un sistema de cajero automático.



4.2.2. Análisis y diseño para realizar los casos de uso.

Durante el análisis y el diseño, transformamos el modelo de casos de uso mediante un modelo de análisis en un modelo de diseño, es decir, en una estructura de clasificadores (clases) y realizaciones de casos de uso. El objetivo es realizar los casos de uso de una forma económica de manera que el sistema ofrezca un rendimiento adecuado y pueda evolucionar en el futuro.

El modelo de análisis crece incrementalmente a medida que se analizan más y más casos de uso. En cada iteración, elegimos un conjunto de casos de uso y los reflejamos en el modelo de análisis. Construimos el sistema como una estructura de clasificadores (clases de análisis) y relaciones entre ellas. También describimos las colaboraciones que llevan a cabo los casos de uso, es decir, las realizaciones de los casos de uso.

Al igual que en Métrica 3, las clases que se identifican son Clases de Entidad (representan la información manipulada en el caso de uso, información que puede ser persistente), Clases de Interfaz de Usuario (interacción entre el sistema y sus actores) y Clases de Control (coordinación, secuencia de transacciones y control de los objetos relacionados con un caso de uso).

Por ejemplo, realización de un caso de uso (colaboración) en el modelo de análisis. Se describe cómo se lleva a cabo el caso de uso Sacar Dinero mediante una colaboración con una dependencia de traza. Salida e Interfaz de Cajero son clases de interfaz, Retirada de Efectivo es una clase de control y Cuenta es una clase de entidad.

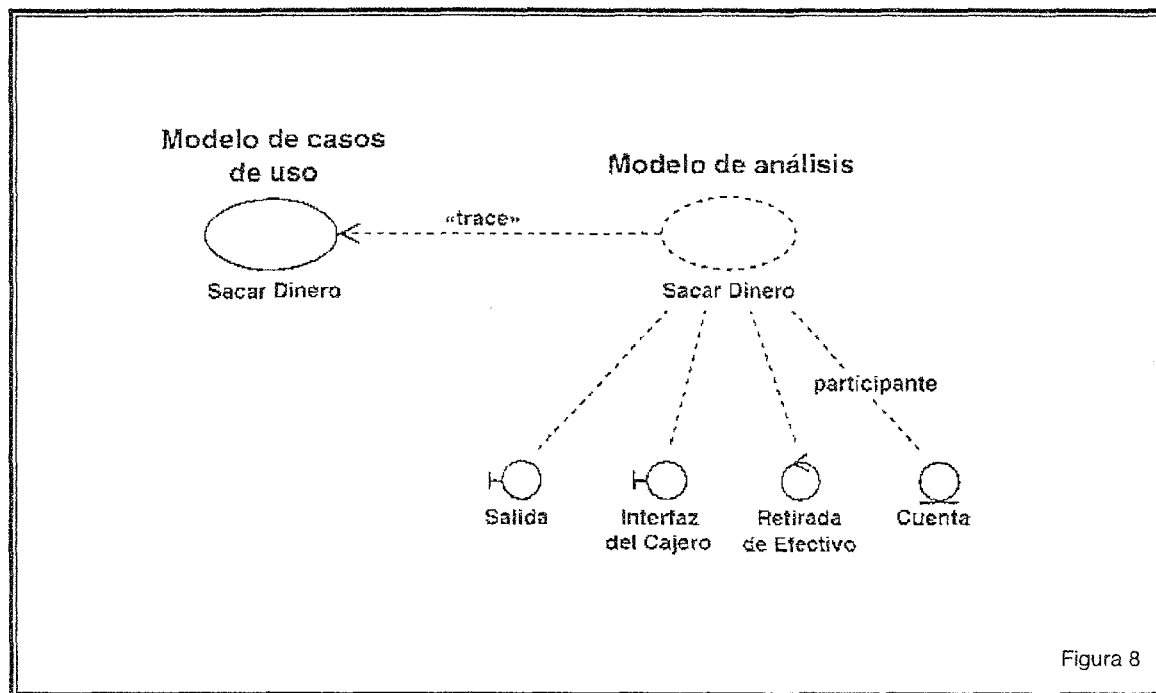
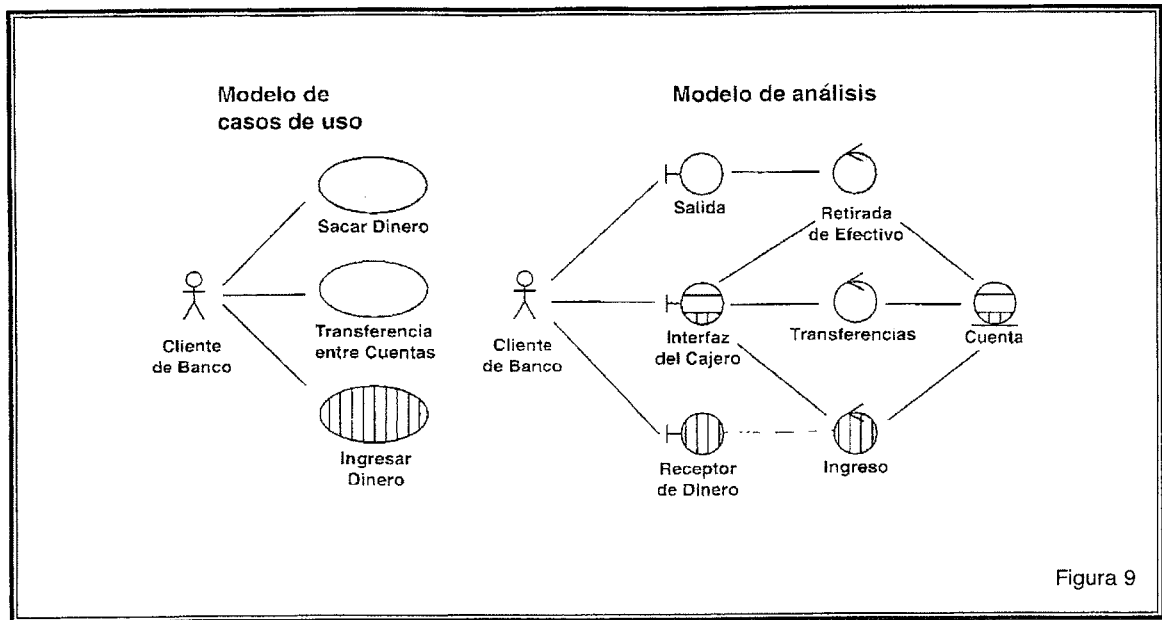
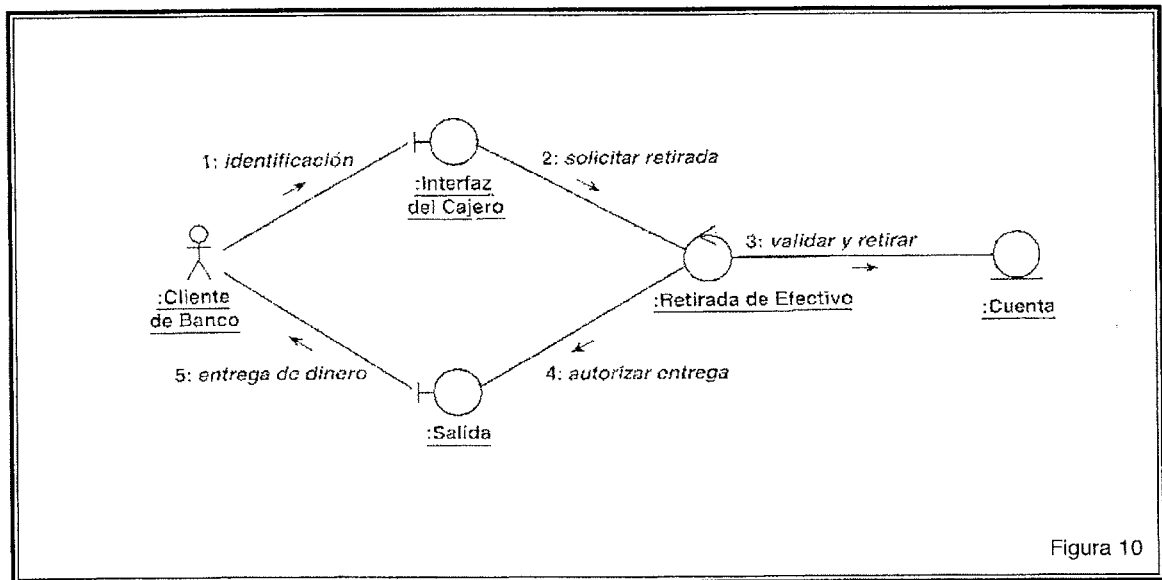


Figura 8

A continuación se presenta cómo cada caso de uso se realiza como una estructura de clases del análisis (se puede observar que Interfaz de Cajero desempeña un rol en los tres casos de uso).



En la siguiente figura se puede observar cómo colaboran las diferentes clases para realizar el caso de uso en el modelo de análisis.



Mientras que el modelo de análisis sirve como una primera aproximación del modelo de diseño, el modelo de diseño funciona como esquema para la implementación. Al igual que el análisis, el modelo de diseño también define clasificadores (clases, subsistemas e interfaces), relaciones entre esos clasificadores y colaboraciones que llevan a cabo los casos de uso. El modelo de diseño es más «físico» porque se adapta al entorno de la implementación mientras que el de análisis es más «conceptual».

De la misma manera que antes, los casos de uso del modelo de diseño son realizaciones de los casos de uso del modelo de análisis (con relaciones de traza) que, a su vez, lo son del modelo de casos de uso.

A continuación se pueden observar las clases de diseño con sus trazas hacia las clases del modelo de análisis.

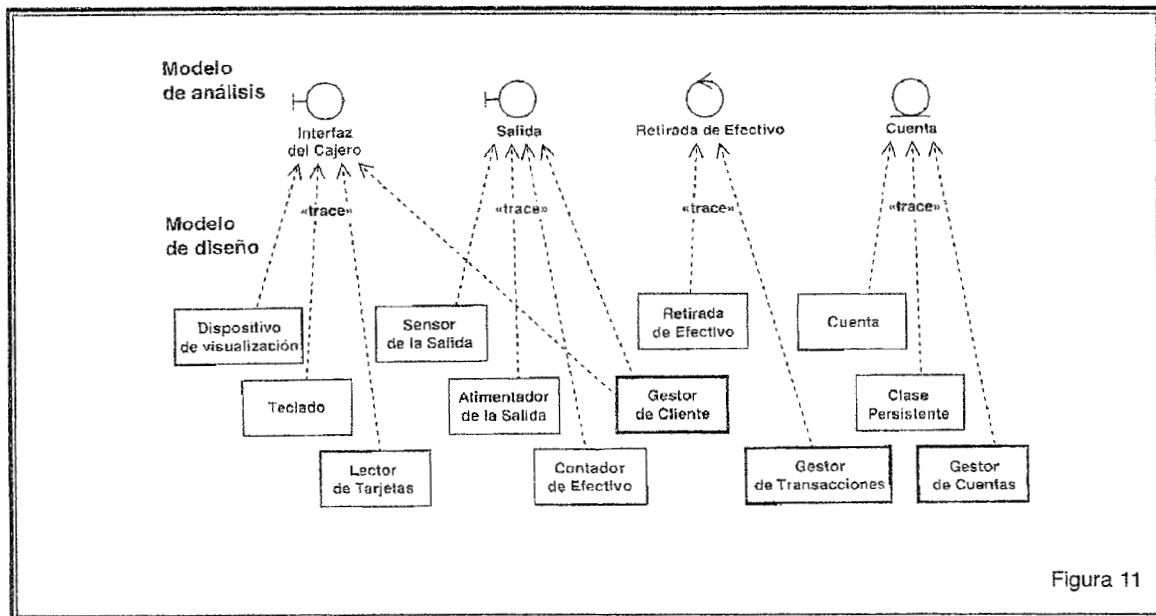


Figura 11

A continuación se muestra el diagrama de clases que es parte de la realización del caso de uso Sacar Dinero en el modelo de diseño.

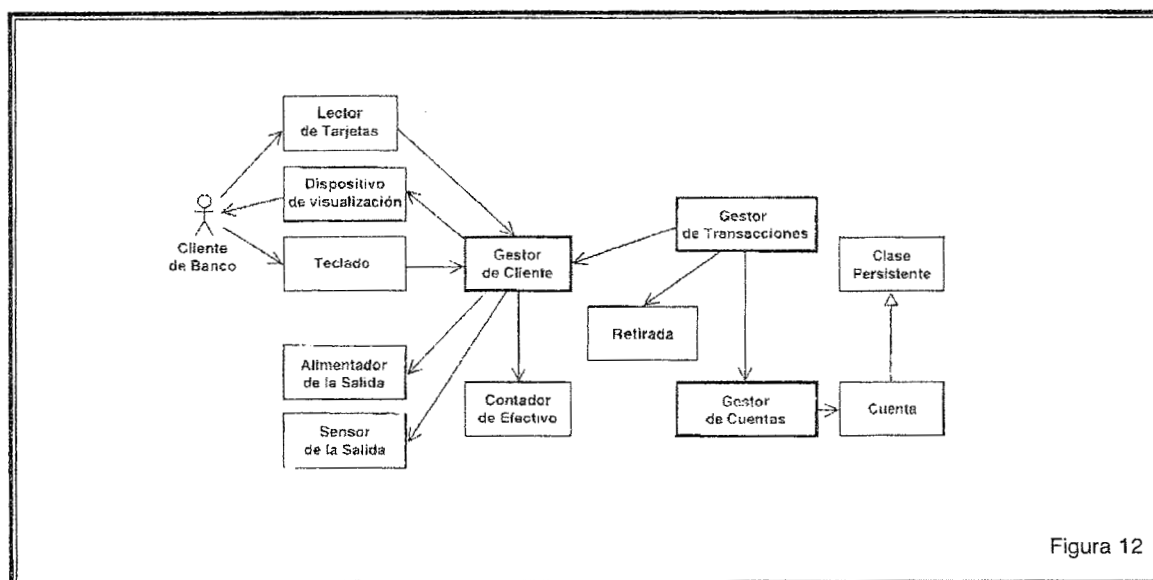


Figura 12

Y el diagrama de secuencia parte de la realización del caso de uso Sacar Dinero en el modelo de diseño entre:

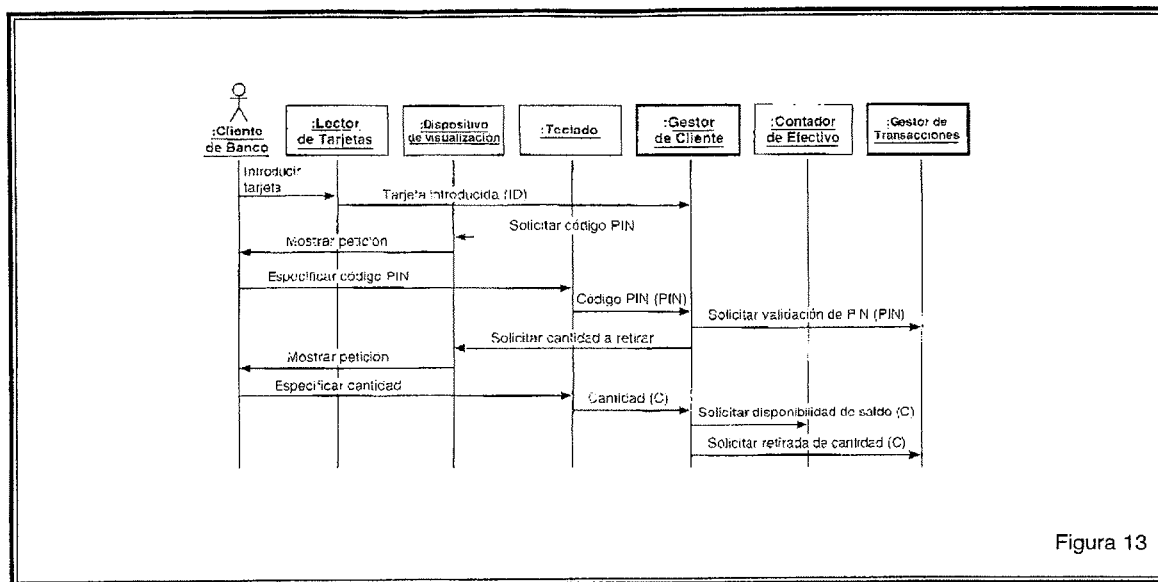


Figura 13

En resumen, en el Proceso Unificado de Desarrollo del Software, los casos de uso dirigen el proceso. Durante el flujo de trabajo de los requisitos, los desarrolladores pueden representar los requisitos en la forma de casos de uso. Los jefes de proyecto pueden después planificar el proyecto en términos de los casos de uso con los cuales trabajan los desarrolladores. Durante el análisis y diseño, los desarrolladores crean realizaciones de casos de uso en términos de clases y subsistemas.

5. EL LENGUAJE DE MODELIZACIÓN UNIFICADO.

5.1. INTRODUCCIÓN.

UML (Unified Modeling Language) es un lenguaje que permite modelar, construir y documentar los elementos que forman un sistema software orientado a objetos. En junio de 2003, UML 2.0 se convirtió en un estándar oficialmente adoptado por el Object Management Group (OMG). Inicialmente concebido por los autores de los tres métodos más usados en orientación a objetos: Booch, Rumbaugh y Jacobson. UML no define ningún proceso de desarrollo específico, tan sólo se trata de una especificación de notación orientada a objetos.

Esta notación incorpora las principales ventajas de cada uno de los métodos particulares en que se basa: BOOCH, OMT y OOSE, y ha puesto fin a las llamadas «guerras de métodos» que se han mantenido a lo largo de los años 90. La utilización de UML es independiente del lenguaje de programación y de las características de los proyectos, ya que ha sido diseñado para modelar cualquier tipo de proyecto informático. Es por tanto, un lenguaje de modelado independiente de la metodología de desarrollo elegida (como se ha podido apreciar en los puntos de ciclos de vida de desarrollo de Métrica 3 y Proceso Unificado de Desarrollo del Software).

El objetivo principal cuando se empezó a gestar UML era posibilitar el intercambio de modelos entre las distintas herramientas CASE orientadas a objetos del mercado. Para ello era necesario definir una notación y semántica común.

UML resta protagonismo al diagrama de clases, ya que si bien representa una parte importante del sistema, sólo representa una vista estática. El diagrama de clases muestra la estructura del sistema y ayuda a diseñar un sistema robusto con partes reutilizables, pero no ayuda a solucionar problemas de propagación de mensajes ni de sincronización o recuperación ante estados de error. Esto es, con el diagrama de clases conocemos el sistema parado, pero no lo que le sucede a sus diferentes partes cuando empieza a funcionar.

Para solucionar esto UML introduce nuevos diagramas que representan una visión dinámica del sistema. Gracias al diseño de la parte dinámica del sistema se pueden detectar en la fase de diseño problemas de la estructura al propagar errores o de las partes que necesitan ser sincronizadas, así como del estado de cada una de las instancias en cada momento.

UML también intenta solucionar el problema de propiedad de código que se da con los desarrolladores. Al implementar un lenguaje de modelado común para todos los desarrollos se crea una documentación también común, que cualquier desarrollador con conocimientos de UML será capaz de entender, independientemente del lenguaje utilizado para el desarrollo.

Por último, UML permite la modificación de todos sus componentes mediante mecanismos de extensibilidad: estereotipos, restricciones y valores etiquetados.

- Un estereotipo permite indicar especificaciones del lenguaje al que se refiere el diagrama de UML (por ejemplo, un paquete de estereotipo «subsistema» o una clase con estereotipo «persistente»).
- Una restricción identifica un comportamiento forzado de una clase o relación. Mediante la restricción se está forzando el comportamiento que debe tener el objeto al que se le aplica.
- Un valor etiquetado extiende las propiedades de un bloque de construcción UML, permitiendo añadir nueva información en la especificación de ese elemento. Por ejemplo, si se está trabajando en un producto que atraviesa muchas versiones a lo largo del tiempo, se querrá registrar la versión y el autor de ciertas abstracciones críticas. Éstos no son conceptos primitivos de UML pero se podrán asociar mediante valores etiquetados.

5.2. LOS DIAGRAMAS DE UML.

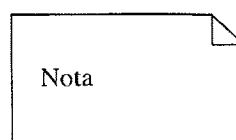
UML define 12 tipos de diagramas divididos en tres grandes categorías:

- Diagramas Estructurales: representan la estructura estática de la aplicación.
 - Diagrama de clases: muestra las clases, interfaces, colaboraciones y sus relaciones. Son los más comunes y dan una vista estática del proyecto.
 - Diagrama de objetos: es un diagrama de instancias de las clases mostradas en el diagrama de clases. Muestra las instancias y cómo se relacionan entre ellas. Da una visión de casos reales.
 - Diagrama de componentes: muestra la organización de los componentes del sistema. Un componente se corresponde con una o varias clases, interfaces o colaboraciones.
 - Diagrama de despliegue: muestra los nodos y sus relaciones. Un nodo es un conjunto de componentes. Se utiliza para reducir la complejidad de los diagramas de clases y componentes de un gran sistema. Sirve como resumen e índice.

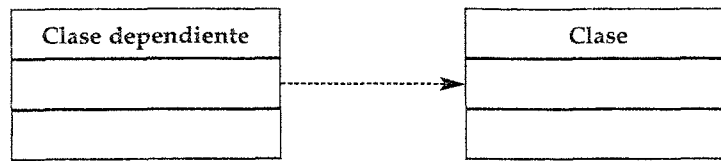
- Diagramas de Comportamiento: representan el comportamiento dinámico.
 - Diagrama de casos de uso: muestra los casos de uso, actores y sus relaciones. Muestra quién puede hacer qué y qué relaciones existen entre acciones (casos de uso). Son muy importantes para modelar y organizar el comportamiento del sistema.
 - Diagramas de Interacción: muestran los diferentes objetos y las relaciones que pueden tener entre ellos, esto es, los mensajes que se intercambian. Los Diagramas de Interacción son dos: el Diagrama de secuencia y el Diagrama de colaboración, que dan puntos de vista diferentes del sistema, pero se puede pasar de uno a otro sin pérdida de información.
 - Diagrama de estados: muestra los estados, eventos, transiciones y actividades de los diferentes objetos. Son útiles en sistemas que reaccionen a eventos.
 - Diagrama de actividades: es un caso especial del diagrama de estados. Muestra el flujo entre los objetos. Se utilizan para modelar el funcionamiento del sistema y el flujo de control entre objetos.
- Diagramas de Gestión del Modelo: representan posibles formas de organizar y gestionar los módulos de la aplicación.
 - Diagramas de Paquetes: se utilizan para organizar los elementos de modelado en partes mayores que se pueden manipular como un grupo. La visibilidad de estos elementos puede controlarse para que algunos sean visibles fuera del paquete mientras que otros permanezcan ocultos. Los paquetes bien diseñados agrupan elementos cercanos semánticamente y que suelen cambiar juntos (Acoplamiento Bajo y Coherencia Alta).
 - Diagramas de Subsistemas: un Sistema se representa como un paquete estereotipado. Un subsistema es una parte de un sistema y se utiliza para descomponer un sistema complejo en partes casi independientes. La relación principal (no la única) entre los sistemas y los subsistemas es la agregación.
 - Diagramas de Modelos: un modelo es un tipo especial de paquete que contiene los artefactos del sistema o subsistema de acuerdo con las cinco vistas de la arquitectura que se ven después. Un modelo (por ejemplo, un modelo de proceso) puede llegar a contener tantos artefactos (clases activas, relaciones e interacciones) que sería imposible incluir todos a la vez. Se puede pensar que se trata de una vista que es una proyección del modelo. Por cada modelo se tendrán varios diagramas que permitirán observar los elementos del modelo.

Los elementos comunes a todos los diagramas son:

- Las notas, que sirven para añadir cualquier tipo de comentario a un diagrama o a un elemento de un diagrama.



- Las dependencias entre dos elementos de un diagrama significa que un cambio en el elemento destino puede implicar un cambio en el elemento origen.



UML divide cada proyecto en un número de diagramas que representan las diferentes vistas del proyecto. Estos diagramas juntos son los que representan la arquitectura del proyecto. Se puede asegurar que la arquitectura de un sistema con gran cantidad de software puede describirse mejor a través de cinco vistas relacionadas:

- Vista de Diseño: soporta los requisitos funcionales del sistema.
 - Modelos estáticos: Clases y de Objetos.
 - Modelos dinámicos: Interacción, Transición de Estados y Actividades.
- Vista de Implementación: Gestión de Configuración, Diagrama de Componentes. Para los aspectos dinámicos se utilizan los diagramas de interacción, estados y actividades.
- Vista de Procesos: sincronización y concurrencia, funcionamiento, capacidad de crecimiento y rendimiento del sistema. Pone énfasis en las clases activas de la vista de diseño.
- Vista de Despliegue: Diagramas de Despliegue. Para los aspectos dinámicos se utilizan los diagramas de interacción, estados y actividades.
- Vista de Casos de Uso: es la vista central y que reúne a todas las demás. Diagramas de Casos de Uso. Para los aspectos dinámicos se utilizan los diagramas de interacción, estados y actividades.

El número de diagramas es muy alto, en la mayoría de los casos excesivos, pero UML permite definir sólo los necesarios en cada proyecto en función de las necesidades que se tengan.

DIAGRAMA DE CASOS DE USO

El diagrama de casos de uso se emplea para visualizar el comportamiento del sistema, una parte de él o de una sola clase, de forma que se pueda conocer cómo responde esa parte del sistema. Dicho de otra forma, representa la funcionalidad que ofrece el sistema en lo que se refiere a su interacción externa.

Además de esta utilidad, el diagrama de casos de uso es un buen sistema de documentar partes del código que deban ser reutilizables por otros desarrolladores, también puede ser utilizado para que los clientes o usuarios se comuniquen con los informáticos sin llegar a niveles de complejidad.

Los elementos de que consta un diagrama de casos de uso son:

- Los actores, que son las entidades externas que realizan algún tipo de interacción con el sistema. Se representan mediante una figura humana dibujada con palotes.
- Los casos de uso, que son una descripción de la secuencia de interacciones que se producen entre un actor y el sistema cuando el actor usa el sistema para llevar a cabo una tarea específica. Se representan mediante una elipse con el nombre dentro.

Un caso de uso es una secuencia de acciones realizadas por el sistema, que producen un resultado observable y valioso para el usuario, esto es, representa el comportamiento del sistema con el fin de dar respuestas a los usuarios, o lo que es lo mismo, un requerimiento funcional.

- Las relaciones entre los casos de uso: extiende (extend), cuando un caso especializa a otro extendiendo su funcionalidad, y usa (include), cuando un caso utiliza a otro. Se representan mediante una flecha que une los casos relacionados y con una etiqueta indicativa del tipo de relación. Por ejemplo, «usa».
- Las relaciones entre actores o entre actores y casos de uso: comunica (communicate), comunica un actor con un caso de uso, o con otro actor.
- La parte del sistema (System boundary): se representa por un cuadro, identifica las diferentes partes del sistema y contiene los casos de uso que la forman.

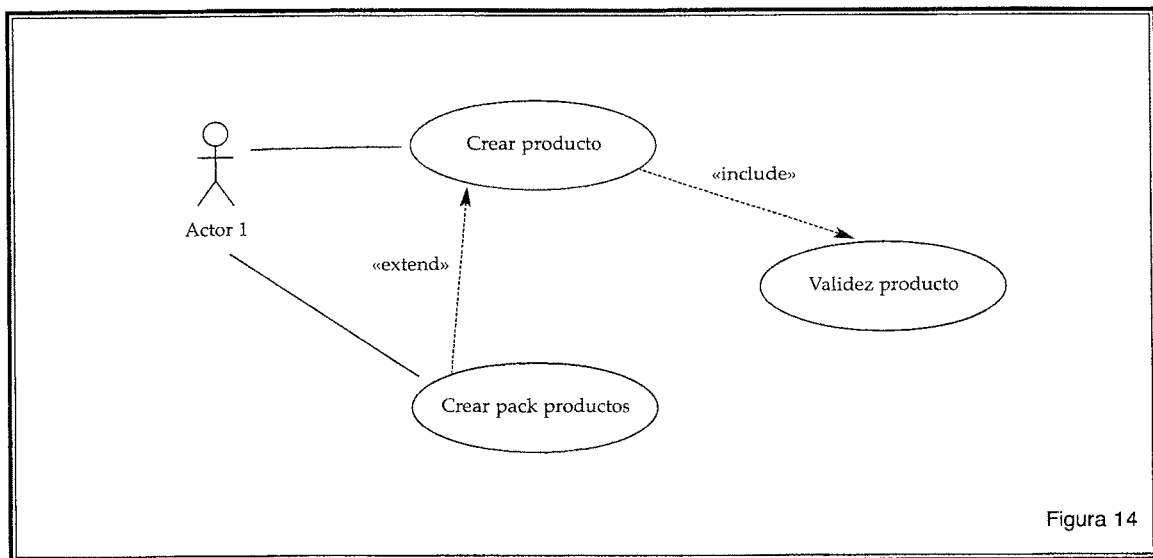


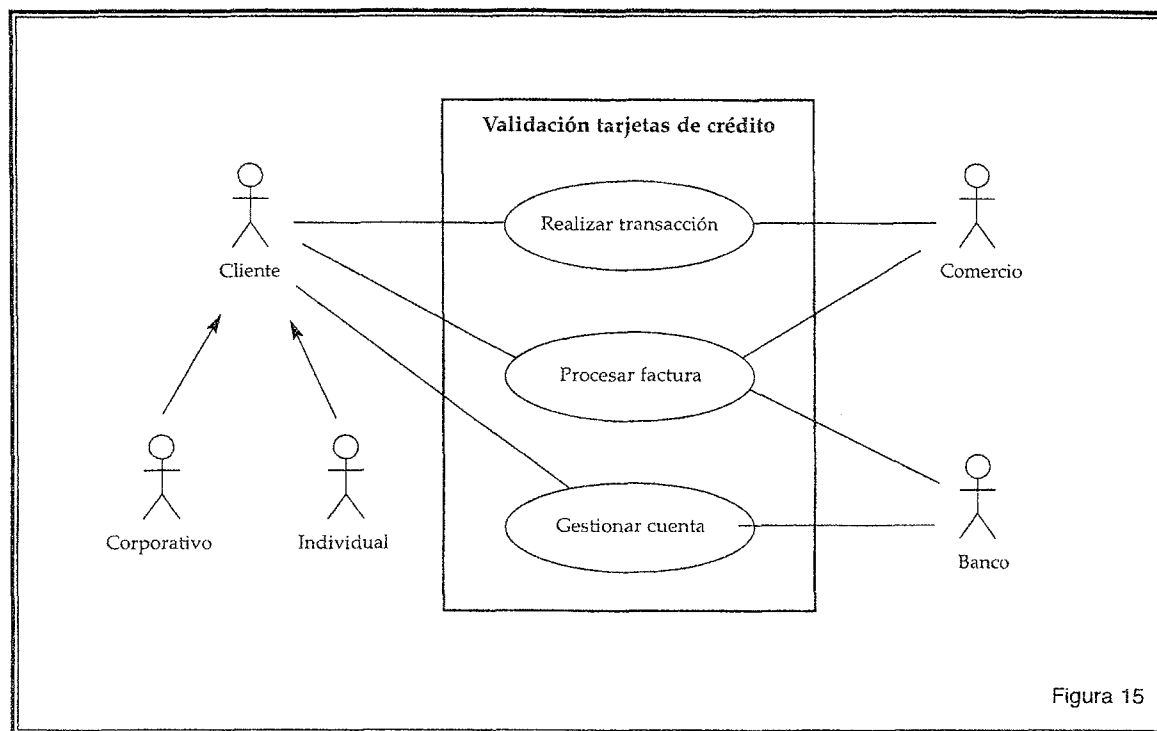
Figura 14

En este gráfico el caso de uso «Crear Producto» utiliza el caso «Validar Producto», y el caso «Crear Pack Productos» es una especialización del «Crear Productos».

El diagrama de casos de uso se puede emplear de dos formas diferentes, para modelar el contexto de un sistema, y para modelar los requisitos del sistema.

Para el Modelado del Contexto, se debe modelar la relación del sistema con los elementos externos, ya que son estos elementos los que forman el contexto del sistema. Los pasos a seguir son:

- Identificar los actores que interactúan con el sistema.
- Organizar a los actores.
- Especificar sus vías de comunicación con el sistema.



El Modelado de Requisitos es la función principal y la más conocida del diagrama de casos de uso. Los requisitos establecen un contrato entre el sistema y su exterior y definen lo que se espera que realice el sistema, sin definir su funcionamiento interno. El modelado de requisitos indica cuáles deben ser las funcionalidades (requisitos) del sistema, e incorpora los casos de uso necesarios que no son visibles desde los usuarios del sistema.

Para modelar los requisitos es recomendable:

- Establecer su contexto, para lo que también se puede usar un diagrama de casos de uso.
- Identificar las necesidades de los elementos del contexto (Actores).
- Nombrar esas necesidades, y darles forma de caso de uso.
- Identificar los casos de uso que pueden ser especializaciones de otros, o buscar especializaciones comunes para los casos de uso ya encontrados.

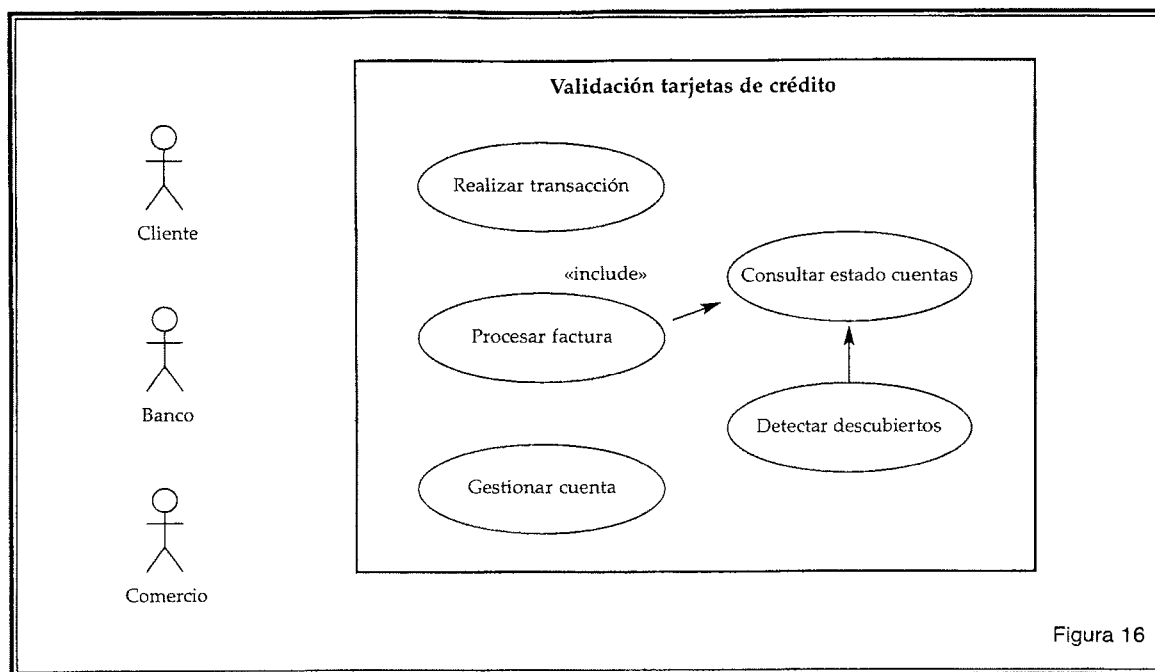


DIAGRAMA DE CLASES

El diagrama de clases forma parte de la vista estática del sistema y recoge las clases de objetos y sus relaciones o asociaciones. En este diagrama se definen las características de cada una de las clases, interfaces, colaboraciones y relaciones de dependencia y generalización.

Los elementos básicos del diagrama de clases son: las clases, las relaciones y las interfaces.

Las Clases:

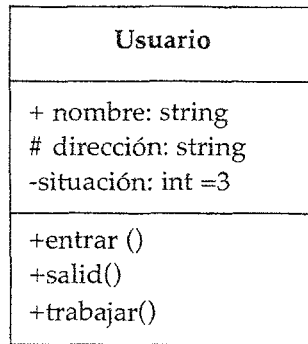
Una clase describe un conjunto de objetos con propiedades (atributos) similares y un comportamiento común. Los objetos son las instancias de las clases.

La clase se representa por un rectángulo dividido en tres partes; la primera para indicar el nombre, la segunda para los atributos y la tercera para los métodos. Cada clase debe tener un nombre único, que las diferencie de las otras.

Un atributo representa alguna propiedad de la clase que se encuentra en todas las instancias de la clase. Los atributos pueden representarse sólo mostrando su nombre, mostrando su nombre y su tipo, e incluso su valor por defecto.

Un método u operación es la implementación de un servicio de la clase, que muestra un comportamiento común a todos los objetos. En resumen es una función que le indica a las instancias de la clase que hagan algo.

Para separar las grandes listas de atributos y de métodos se pueden utilizar estereotipos.



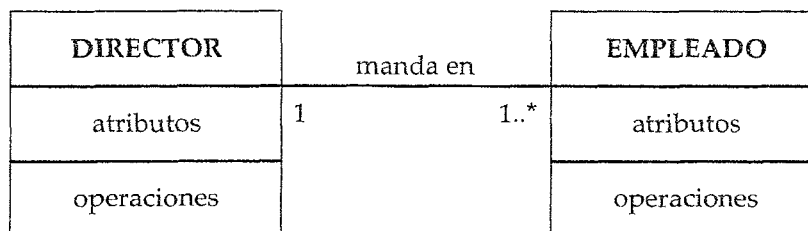
En este ejemplo, la clase «Usuario» contiene tres atributos: «Nombre», que es public; «Dirección», que es protected; y «Situación», que es private y empieza con el valor 3. También dispone de tres métodos: Entrar, Salir y Trabajar.

Relaciones entre clases.

Los principales tipos de relaciones estáticas entre clases son: la Asociación, la Generalización o Herencia, la Agregación y la Dependencia.

En las relaciones se habla de una clase destino y de una clase origen. La origen es desde la que se realiza la acción de relacionar. Es decir desde la que parte la flecha, la destino es la que recibe la flecha. Las relaciones se pueden modificar con estereotipos o con restricciones.

Las relaciones entre dos clases se representan mediante una línea que las une. El nombre de la relación es opcional y se puede añadir un triángulo negro que indique la dirección en la que leer el nombre de la relación.



La multiplicidad es una restricción que se pone a una relación y que limita el número de instancias de una clase que pueden tener esa relación con una instancia de la otra clase. El símbolo * indica que puede tomar cualquier valor (0 o más). En el ejemplo se ha representado un rango (1 o más).

Para indicar el papel que juega una clase en una relación se puede especificar el nombre de rol, en el extremo de la relación junto a la clase que desempeña dicho rol. Por ejemplo, contratante (Director) y contratado (Empleado).

La Asociación.

Es el tipo de relación más general y denota básicamente una dependencia semántica. Especifica que los objetos de una clase están relacionados con los elementos de otra clase. Se representa mediante una línea continua, que une las dos clases. Podemos indicar el nombre, multiplicidad en los extremos, su rol, y agregación.

La Generalización o Herencia.

Herencia es el mecanismo que permite a una clase de objetos incorporar atributos y métodos de otra clase, añadiéndolos a los que ya posee. Con la herencia se refleja una relación «es_un» entre clases en la que la clase de la que se hereda se denomina superclase y la clase que hereda, subclase. La relación de herencia se representa mediante un triángulo en el extremo de la relación que corresponde a la clase más general (superclase).

UML soporta tanto herencia simple como herencia múltiple, y además, permite modificar la relación de Generalización con un estereotipo y dos restricciones.

- Estereotipo de generalización.
 - Implementation: la subclase hereda la implementación de la superclase, sin publicar ni soportar sus interfaces.
- Restricciones de generalización.
 - Complete: la generalización ya no permite más subclases.
 - Incomplete: se pueden incorporar más subclases a la generalización.
 - Disjoint: sólo se puede tener un tipo en tiempo de ejecución, es decir, una instancia de la superclase sólo podrá ser de un tipo de subclase.
 - Overlapping: se puede cambiar de tipo durante su vida, o sea, una instancia de la superclase puede ir cambiando de tipo entre los de sus subclases.

La Agregación.

Es un tipo de relación jerárquica entre una clase que representa el todo y las partes que la componen. Refleja una relación «es_parte_de» y permite el agrupamiento físico de estructuras relacionadas lógicamente. La relación de agregación se representa mediante un diamante colocado en el extremo en el que está la clase que representa el «todo».

La Dependencia.

Es una relación de uso, es decir, una clase usa a otra, que la necesita para su cometido. Se representa con una flecha discontinua que va desde la clase utilizadora a la clase utilizada. Con la dependencia se muestra que un cambio en la clase utilizada puede afectar al funcionamiento de la clase uti-

lizador, pero no al contrario. Aunque las dependencias se pueden crear tal cual, es decir, sin ningún estereotipo, UML permite dar más significado a las dependencias, es decir concretar más, mediante el uso de estereotipos.

- Estereotipos de relación Clase-objeto.
 - Bind: la clase utilizada es una plantilla y necesita de parámetros para ser utilizada. Con Bind se indica que la clase se instancia con los parámetros pasándole datos reales para sus parámetros.
 - Derive: se utiliza para indicar relaciones entre dos atributos e indica que el valor de uno depende directamente del valor de otro.
 - Friend: especifica una visibilidad especial sobre la clase relacionada. Es decir, podrá ver las interioridades de la clase destino.
 - InstanceOF: indica que el objeto origen es una instancia del destino.
 - Instantiate: indica que el origen crea instancias del destino.
 - Powertype: indica que el destino es un contenedor de objetos del origen, o de sus hijos.
 - Refine: se utiliza para indicar que una clase es la misma que otra, pero más refinada, es decir, dos vistas de la misma clase, la destino con mayor detalle.

Interfaces.

Un interfaz es una especificación de la semántica de un conjunto de operaciones de una clase o paquete que son visibles desde otras clases o paquetes. Normalmente se corresponde con una parte del comportamiento del elemento que la proporciona. Se representa como una caja con tres compartimentos, igual que las clases. En la zona superior se incluye el nombre y el estereotipo «Interface», y en la zona inferior se coloca la lista de operaciones. La zona media estará vacía o puede omitirse.

DIAGRAMA DE OBJETOS.

El Diagramas de Objetos también forma parte de la vista estática del sistema. En este diagrama se modelan las instancias de las clases del diagrama de clases. Muestra los objetos y sus relaciones, pero en un momento concreto del sistema. Estos diagramas contienen objetos y enlaces. En los diagramas de objetos también se pueden incorporar clases para mostrar la clase de la que es representado un objeto.

Por ejemplo, para el diagrama de clases que se muestra en la siguiente figura, se tendría un diagrama de objetos con dos instancias de «Mensaje», más concretamente con una instancia de «MensajeDIR» y otra de «MensajeADM», con todos sus atributos valorados. También se tendría una instancia de cada una de las otras clases que deban tener instancia, como «CanalEnt», «INS», «Distr», y el «Buzón» correspondiente a la instancia de mensaje que se esté instanciando.

En la instancia de la clase «INS» se deberá mostrar en su miembro Estado, que está ocupado realizando una inserción.

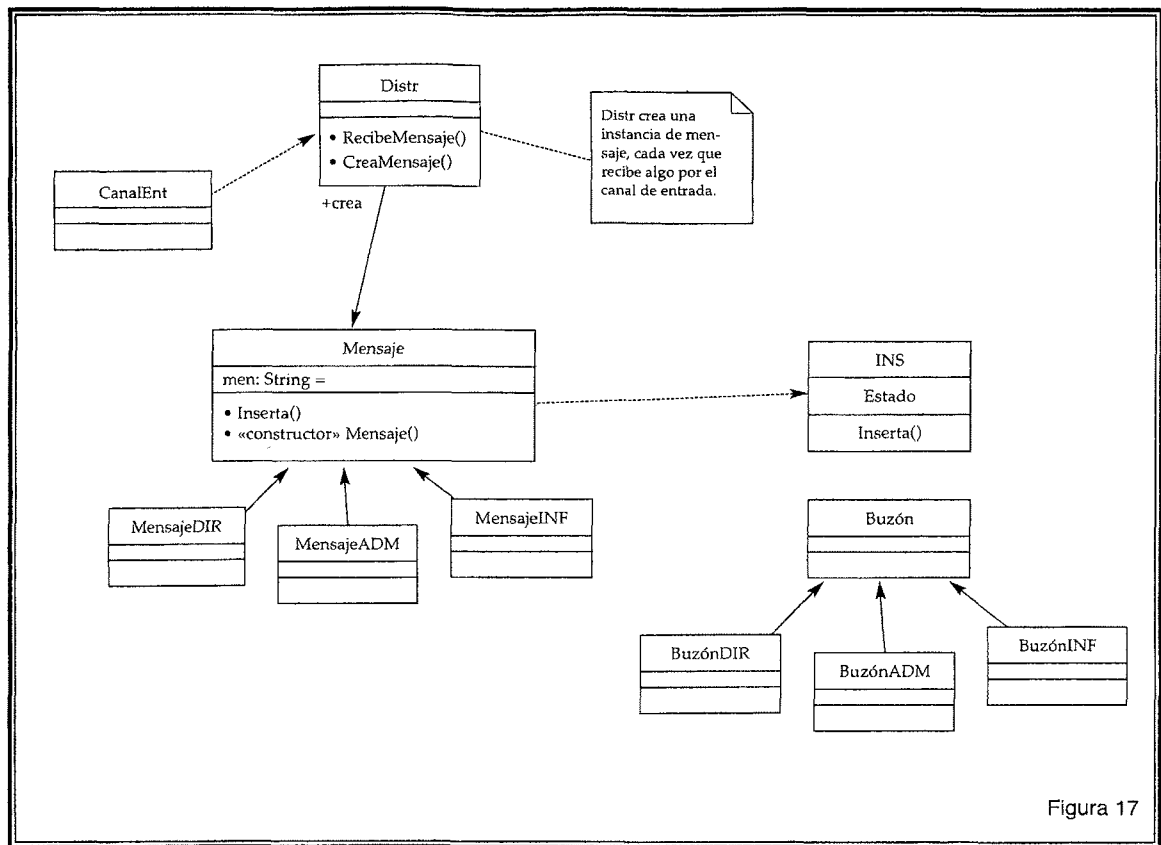


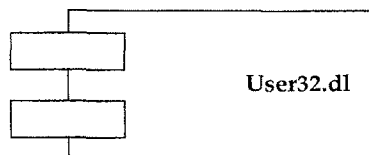
Figura 17

DIAGRAMA DE COMPONENTES.

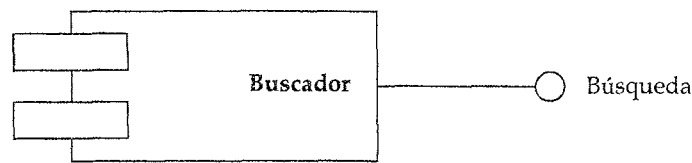
El Diagrama de Componentes también se utiliza para modelar la vista estática de un sistema. Muestra la organización de los componentes software, sus interfaces y las dependencias entre ellos. En este diagrama se sitúan las librerías, tablas archivos, ejecutables y documentos que forman parte del sistema. Uno de los usos principales es que puede servir para ver qué componentes pueden compartirse entre sistemas o entre diferentes partes de un sistema.

Los elementos del diagrama de componentes son: los componentes, las interfaces y las relaciones de dependencia.

Un componente se representa como un rectángulo con dos pequeños rectángulos superpuestos perpendicularmente en el lado izquierdo.



Una interfaz se representa como un pequeño círculo situado junto al componente que lo implementa y unido a él mediante una línea continua.



En esta figura tenemos el mismo componente anterior, pero indicando que dispone de una interfaz (se podría pensar que la representación anterior es incorrecta, pero no es, así sólo corresponde a un nivel diferente de detalle).

Todo objeto UML puede ser modificado mediante estereotipos. En este caso, los estándares que define UML son:

- Executable.
- Library.
- Table.
- File.
- Document.

DIAGRAMA DE INTERACCIÓN: SECUENCIA Y COLABORACIÓN.

El diagrama de secuencia es un tipo de Diagrama de Interacción y forma parte del modelado dinámico del sistema.

Los Diagramas de Interacción tienen por objeto describir el comportamiento dinámico del sistema de información mediante el paso de mensajes entre los objetos del mismo. Además, representan un medio para verificar la coherencia del sistema mediante la validación del modelo de clases.

Los Diagramas de Interacción describen en detalle un determinado escenario de un caso de uso. En él se muestra la interacción entre el conjunto de objetos que cooperan en la realización de dicho escenario.

El Diagrama de Secuencia es un tipo de diagrama de interacción cuyo objeto es describir el comportamiento dinámico del sistema de información haciendo énfasis en la secuencia de los mensajes intercambiados por los objetos.

En el Diagrama de Secuencia se muestra el orden de las llamadas en el sistema. Se utiliza un diagrama para cada llamada a representar.

El diagrama se forma con los objetos que forman parte de la secuencia, los cuales se sitúan en la parte superior de la pantalla. El actor que inicia la acción se sitúa normalmente en la izquierda. De estos objetos sale una línea que indica su vida en el sistema. Esta línea simple se convierte en una línea gruesa cuando representa que el objeto tiene el foco del sistema, es decir, cuando él está activo.

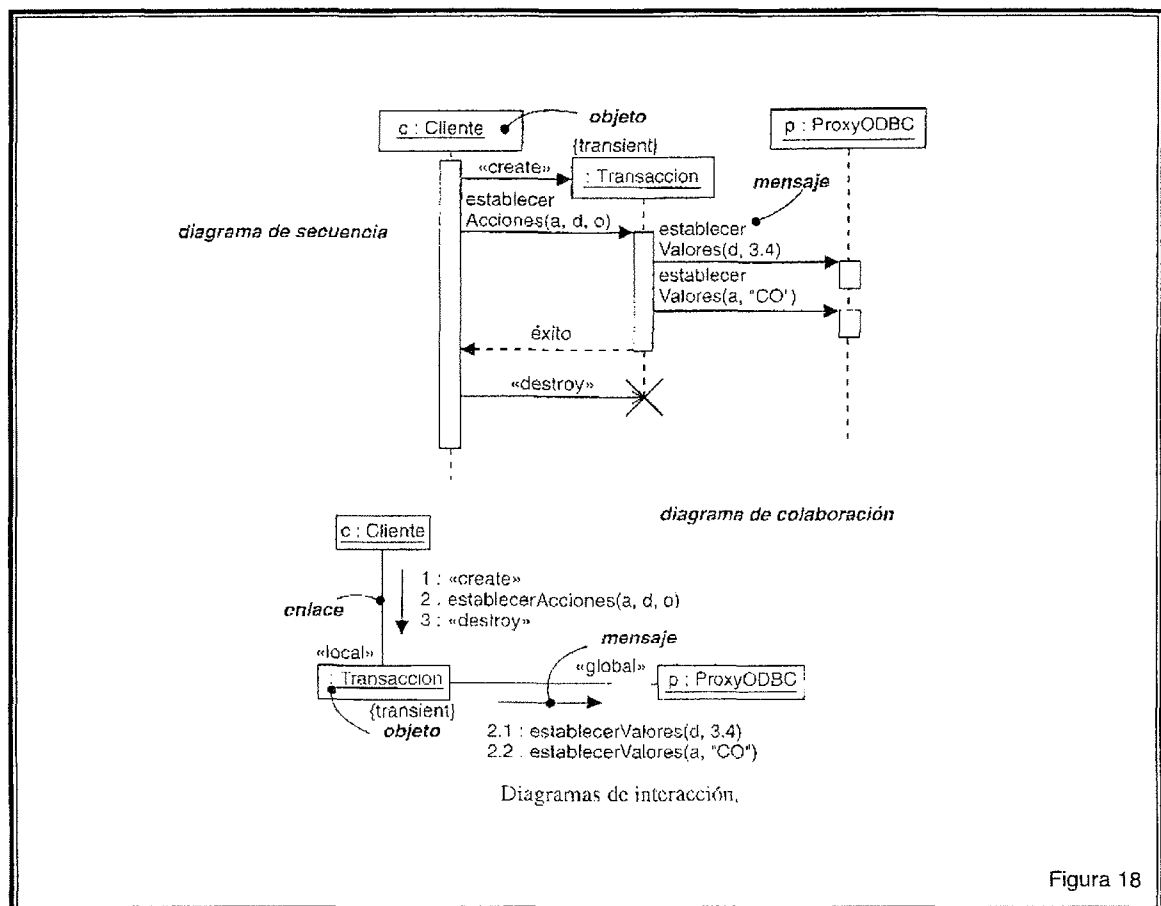


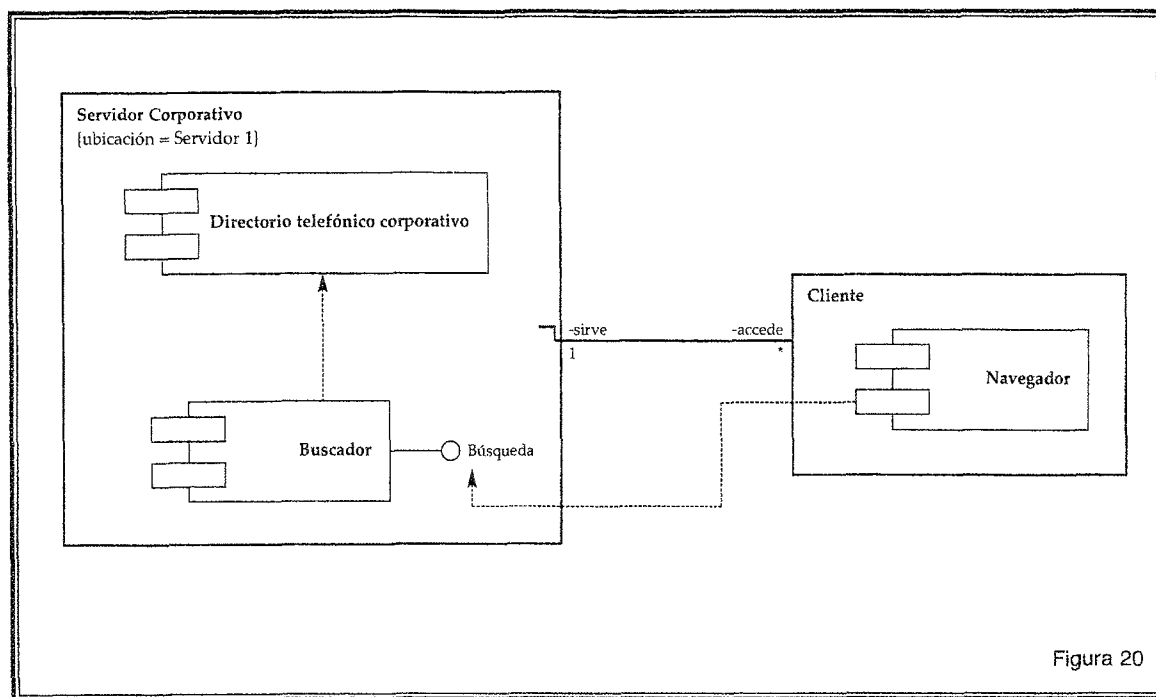
Figura 18

El diagrama de secuencia hace énfasis en la ordenación temporal de los mensajes mientras que el de colaboración hace énfasis en la organización estructural de los objetos que envían y reciben mensajes. El cambio entre uno y otro no supone pérdida de información.

DIAGRAMA DE DESPLIEGUE.

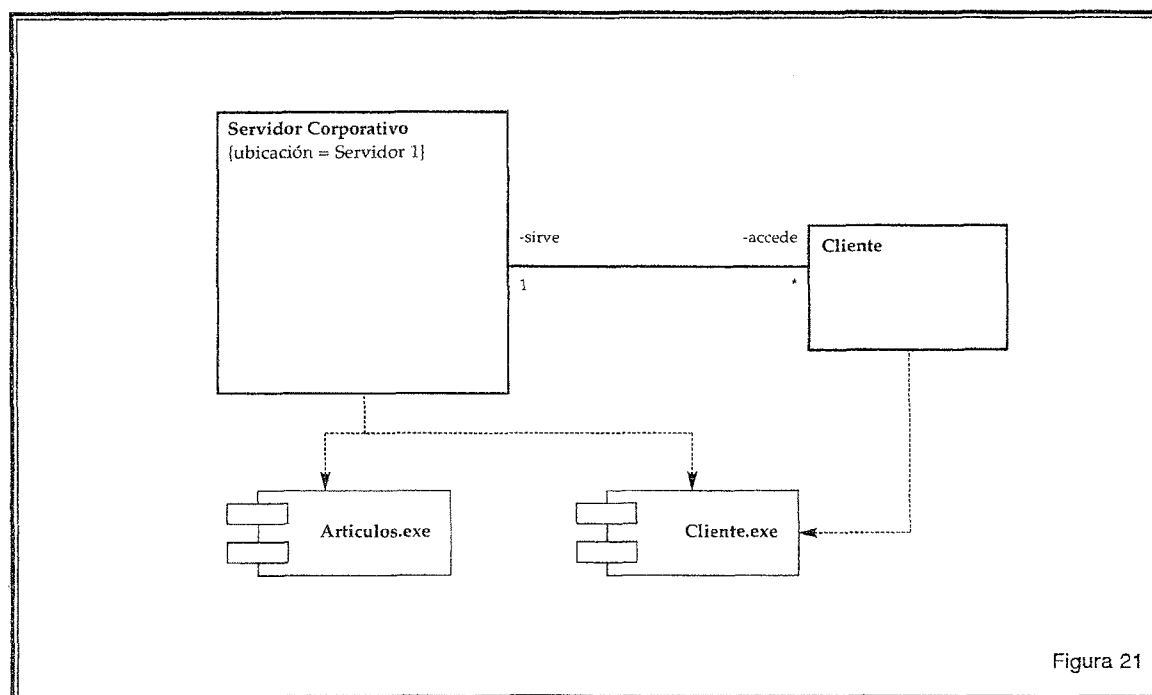
El diagrama de despliegue muestra las relaciones físicas entre los componentes software y hardware del sistema. En él se indica la situación física de los componentes lógicos desarrollados, es decir, se sitúa el software en el hardware que lo contiene. Cada hardware se representa como un nodo.

Un nodo es un elemento donde se ejecutan los componentes, y representa el despliegue físico de estos componentes. Gráficamente, un nodo se representa como un cubo.



En la siguiente figura se tienen dos nodos, el cliente y el servidor. Cada uno de ellos contiene componentes. El componente del cliente utiliza una interfaz de uno de los componentes del servidor. Se muestra la relación existente entre los dos nodos. A esta relación se le podría asociar un estereotipo para indicar el tipo de conexión de que se dispone entre el cliente y el servidor.

Como los componentes pueden residir en más de un nodo, se puede situar el componente de forma independiente, sin que pertenezca a ningún nodo, y relacionarlo con los nodos en los que se sitúa.



6. EL MODELO CORBA.

6.1. INTRODUCCIÓN.

CORBA (Common Object Request Broker Architecture) define la infraestructura para la arquitectura OMA (Object Management Architecture) del OMG (Object Management Group), especificando los estándares necesarios para la invocación de métodos sobre objetos en entornos heterogéneos.

Los entornos heterogéneos son aquellos en los que las arquitecturas que los constituyen pueden ser sistemas de distintos fabricantes, incluyendo los protocolos de comunicaciones y los lenguajes de programación utilizados en las diferentes arquitecturas.

La aparición de CORBA está muy ligada a los sistemas distribuidos. Un sistema distribuido es aquel en el que todas las funciones de la aplicación se exponen como objetos, de forma que cada objeto puede usar todos los servicios o métodos proporcionados por otros objetos del mismo o de diferente sistema.

En los sistemas distribuidos es muy importante la definición de la interfaz entre módulos (objetos), esto es del protocolo de comunicación entre módulos, ya que dicha interfaz especifica a los demás módulos qué servicios ofrece y cómo se usan. CORBA proporciona un estándar para poder definir estas interfaces entre módulos (objetos), así como algunas herramientas para facilitar la implementación de dichas interfaces en el lenguaje de programación escogido.

CORBA es un estándar creado con la idea de una distribución de los sistemas basada en objetos. Es una arquitectura orientada a objetos, por lo que exhibe muchas de las características de estos sistemas, incluyendo la herencia de interfaces y el polimorfismo. Con CORBA se pretende definir una arquitectura que especifique cómo se crean los objetos y cómo se accede a sus funcionalidades.

Las especificaciones de CORBA las define el OMG, consorcio de compañías entre las que figuran SUN. HP. DEC, IBM, etc.

Las dos principales características de CORBA son:

- Es independiente de la plataforma, es decir, los objetos CORBA se pueden utilizar en cualquier plataforma que tenga una aplicación CORBA ORB.
- Es independiente del lenguaje de la aplicación, esto es, los objetos CORBA y los clientes se pueden implementar en cualquier lenguaje de programación. (A un objeto CORBA no le hace falta saber el lenguaje en que ha sido escrito otro objeto con el que se está comunicando).

6.2. LA ARQUITECTURA CORBA.

Los dos componentes fundamentales de CORBA son:

- El ORB (Object Request Broker), que dirige la comunicación entre objetos CORBA.
- El IDL (Interface Definition Language), que define las interfaces de los componentes de la aplicación sobre los que se construyen las aplicaciones CORBA.

Un ORB es un componente software cuyo propósito es facilitar la comunicación entre objetos proporcionando una serie de facilidades tales como: la localización de un objeto remoto dada una referencia a ese objeto, y la ordenación de los parámetros y valores de retorno a y desde invocaciones a métodos remotos.

El IDL es un lenguaje estándar de definición de interfaces que especifica interfaces entre objetos CORBA.

CORBA define su propio modelo de objetos, basado en la definición de las interfaces de los objetos mediante el lenguaje IDL. De esta forma se logra una abstracción de la heterogeneidad que permite que el uso de CORBA no sea complejo.

CORBA ha logrado parte de su éxito en la clara separación entre la interfaz de los objetos y la implementación de los mismos. Las interfaces se definen utilizando el lenguaje IDL, cuya principal característica es su alto nivel de abstracción, lo que le separa de cualquier entorno de desarrollo específico (interoperabilidad). Para la implementación de objetos se puede usar cualquier lenguaje de programación que proporcione enlaces con IDL. De esta forma y a partir de una especificación de las interfaces en IDL, se generan unos proxies en el lenguaje elegido que permiten el acceso a la implementación de los objetos desde la arquitectura COREA.

Otros componentes de la especificación CORBA son:

- El SII (Static Invocation Interface), que es el método de invocación más utilizado. Todos los métodos se especifican de antemano, y tanto cliente como servidor acceden a ellos a través de sus proxies.
- El DII (Dynamic Invocation Interface), que es un método de invocación que proporciona una API que permite que los objetos y sus métodos se puedan especificar e invocar en tiempo de ejecución.
- El DSI (Dynamic Skeleton Interface), que proporciona en el servidor una funcionalidad análoga a la que proporciona el DII en el cliente, y permite que el código acepte peticiones arbitrarias de objetos CORBA.

7. COMPLEMENTOS AL DISEÑO ORIENTADO A OBJETOS.

7.1. PATRONES DE DISEÑO.

Un patrón de diseño es una solución a un problema de diseño que aparece con frecuencia. Muchos de los patrones de diseño están documentados en libros, que presentan los patrones utilizando plantillas estándar. Estas plantillas asignan un nombre a un patrón y presentan un resumen de los problemas y las ventajas que lo hacen surgir, una solución en términos de las clases participantes e interacción entre objetos de esas clases. Las plantillas también proporcionan ejemplos de cómo se utiliza el patrón en algunos lenguajes de programación. Existen patrones de diseño como Façade, Decorador, Proxy Observer, Strategy y Visitor ampliamente citados y utilizados.

También se ha utilizado esta idea para recoger soluciones estándar a problemas de la arquitectura que ocurren frecuentemente. Algunos de estos patrones incluyen Layers, Pipes and Filtres, Broker Blackboard, Horizontal-Vertical Metadata, Chain of responsibility, Singleton y MVC (Model-View-Controller).

Queda fuera del tema el estudio de estos patrones pero se detallan dos apuntes bibliográficos para conocerlos en detalle [Gamma et al] y [Cooper].

7.2. PROGRAMACIÓN ORIENTADA AL ASPECTO.

Es un paradigma de programación que se centra en construcciones llamadas aspectos que describen funcionalidades diferentes a las propias de un conjunto de objetos, clases o funciones. Un ejemplo de estos aspectos que son generales a muchos tipos de objetos diferentes son el control de seguridad, el logging, el control de errores y excepciones, etc. Cuando se desarrolla una clase dada, su funcionalidad debe ser simple y específica. Sin embargo, gran parte del código de la clase está destinado a controlar los aspectos anteriormente comentados.

Por ejemplo, en una aplicación de una tarjeta de crédito, la facturación sería su principal cometido (su principal funcionalidad o concern). El logging y la persistencia de sus objetos serían funcionalidades que probablemente fuesen generales a una gran parte de la jerarquía de clases (crosscut behaviour). La separación de esas funciones de la función principal es el principal objetivo de la programación orientada a aspectos. El código principal ya no contendrá llamadas a esos concerns porque se mantiene separado en aspectos (aspects) que facilitan el mantenimiento y los cambios.

La programación orientada a aspectos complementa principalmente a la programación orientada a objetos pero no se limita a ella únicamente. El lenguaje más actual que soporta este paradigma de programación es AspectJ.

7.3. DISEÑO POR CONTRATO.

Es una metodología de diseño de software que obliga a que los diseñadores definan interfaces precisos y verificables para los componentes software basándose en los Tipos Abstractos de Datos.

El contrato se extiende hasta el nivel de procedimiento/método y contendrá información para los siguientes puntos:

- Parámetros de entrada aceptables e inaceptables.
- Valores de retorno posibles y su semántica.
- Condiciones de error y excepción que puedan ocurrir.
- Efectos laterales.
- Pre-condiciones.
- Post-condiciones.
- Invariantes.
- Rendimiento (opcional).

BIBLIOGRAFÍA

- El Proceso Unificado de Desarrollo del Software. Ivar Jacobson, Grady Booch, James Rumbaugh. Ed. Addison-Wesley.
- El Lenguaje Unificado de Modelado. Ivar Jacobson, Grady Booch, James Rumbaugh. Addison-Wesley.
- An introduction to OMG's UML. Object Management Group. http://www.omg.org/getting-started/what_is_uml.htm
- The C++ Programming Language. Special Edition. Bjarne Stroustrup. Addison-Wesley.
- Java in a nutshell. David Flanagan. O'Reilly.
- Design Patterns: elements of Reusable Object-Oriented Software. Gamma, Helm, Johnson, and Vlissides. Addison-Wesley.
- The Design Patterns. Java companion. James W. Cooper. Addison-Wesley.
- Object-oriented programming. Wikipedia. <http://www.wikipedia.com>
- Introducción a la Orientación a Objetos. Fernando Bellas Permuy. Departamento de Tecnologías de la Información y las Comunicaciones. Universidad de A Coruña. <http://www.tic.udc.es/~fbellas>
- Introducción: conceptos de Programación Orientada a Objetos. Pablo Castells. Escuela Técnica Superior de Informática. Universidad Autónoma de Madrid.
- A History of Object-Oriented Programming Languages and their Impact on Program Design and Software Development. Jeff Sutherland.
- Documentación técnica de Métrica v3. MAP. <http://www.csi.map.es/csi/metrica3>

Guía de técnicas y prácticas.

- Temario de las pruebas selectivas para ingreso en el Cuerpo Superior de Sistemas y Tecnologías de la Información de la Administración del Estado. ASTIC.
 - 01.25. Elementos conceptuales y arquitectura de sistemas abiertos. Sistemas abiertos y normalización de facto.
 - 03.08. El análisis orientado a objetos. Objetos, clases, herencia, métodos. Arquitectura de objetos distribuidos.
 - 03.11. El diseño orientado a objetos.
- Apuntes sobre CORBA.
- Temario del Máster en Ingeniería del Software. Facultad de Informática. Universidad Politécnica de Madrid. Centro de Estudios Financieros.

