FACULTAD DE INGENIERIA U.N.A.M.
DIVISION DE EDUCACION CONTINUA

CENTRO DE INFORMACIÓN Y DOCUMENTACIÓN
" ING. BRUNO MASCANZONI "

El Centro de Información y Documentación Ing. Bruno Mascanzoni tiene por objetivo satisfacer las necesidades de actualización y proporcionar una adecuada información que permita a los ingenieros, profesores y alumnos estar al tanto del estado actual del conocimiento sobre temas específicos, enfatizando las investigaciones de vanguardia de los campos de la ingeniería, tanto nacionales como extranjeras.

Es por ello que se pone a disposición de los asistentes a los cursos de la DECFI, así como del público en general los siguientes servicios:

- **Préstamo interno.**
- **Préstamo externo.**
- **Préstamo interbibliotecario.**
- **Servicio de fotocopiado.**
- **Consulta a los bancos de datos: librunam, seriunam en cd-rom.**

Los materiales a disposición son:

- **Libros.**
- **Tesis de posgrado.**
- **Publicaciones periódicas.**
- **Publicaciones de la Academia Mexicana de Ingeniería.**
- **Notas de los cursos que se han impartido de 1988 a la fecha.**

En las áreas de ingeniería industrial, civil, electrónica, ciencias de la tierra, computación y, mecánica y eléctrica.

El CID se encuentra ubicado en el mezzanine del Palacio de Minería, lado oriente.

El horario de servicio es de 10:00 a 14:30 y 16:00 a 17:30 de lunes a viernes.

Palacio de Minería Calle de Tacuba 5 Primer piso Deleg. Cuauhtémoc 06000 México, D.F. APDO. Postal M-2285
Teléfonos: 512-8955 512-5121 521-7335 521-1987 Fax 510-0573 521-4020 AL 26

FACULTAD DE INGENIERIA U.N.A.M.
DIVISION DE EDUCACION CONTINUA

A LOS ASISTENTES A LOS CURSOS

Las autoridades de la Facultad de Ingeniería, por conducto del jefe de la División de Educación Continua, otorgan una constancia de asistencia a quienes cumplan con los requisitos establecidos para cada curso.

El control de asistencia se llevará a cabo a través de la persona que le entregó las notas. Las inasistencias serán computadas por las autoridades de la División, con el fin de entregarle constancia solamente a los alumnos que tengan un mínimo de 80% de asistencias.

Pedimos a los asistentes recoger su constancia el día de la clausura. Estas se retendrán por el periodo de un año, pasado este tiempo la DECFI no se hará responsable de este documento.

Se recomienda a los asistentes participar activamente con sus ideas y experiencias, pues los cursos que ofrece la División están planeados para que los profesores expongan una tesis, pero sobre todo, para que coordinen las opiniones de todos los interesados, constituyendo verdaderos seminarios.

Es muy importante que todos los asistentes llenen y entreguen su hoja de inscripción al inicio del curso, información que servirá para integrar un directorio de asistentes, que se entregará oportunamente.

Con el objeto de mejorar los servicios que la División de Educación Continua ofrece, al final del curso deberán entregar la evaluación a través de un cuestionario diseñado para emitir juicios anónimos.

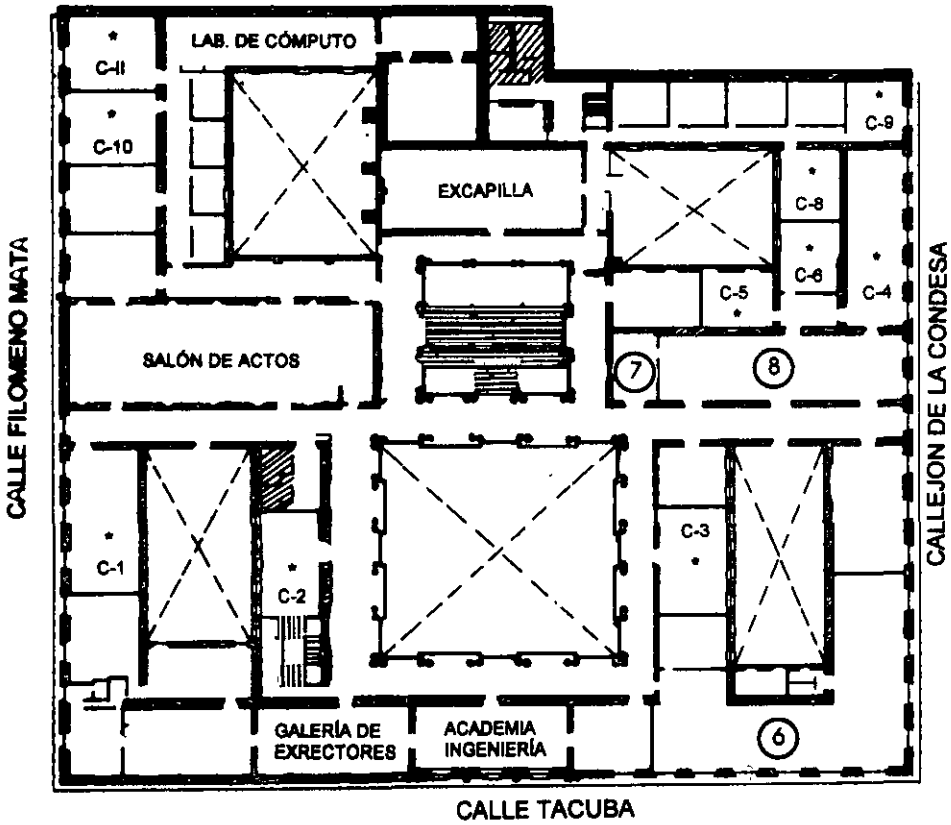
Se recomienda llenar dicha evaluación conforme los profesores impartan sus clases, a efecto de no llenar en la última sesión las evaluaciones y con esto sean más fehacientes sus apreciaciones.

Atentamente

División de Educación Continua.

Palacio de Minería Calle de Tacuba 5 Primer piso Deleg. Cuauhtémoc 06000 México, D.F. APDO. Postal M-2285
Teléfonos: 512-8955 512-5121 521-7335 521-1987 Fax 510-0573 521-4020 AL 26

PALACIO DE MINERIA



1er. PISO

GUÍA DE LOCALIZACIÓN

1. ACCESO
2. BIBLIOTECA HISTÓRICA
3. LIBRERÍA UNAM
4. CENTRO DE INFORMACIÓN Y DOCUMENTACIÓN "ING. BRUNO MASCANZONI"
5. PROGRAMA DE APOYO A LA TITULACIÓN
6. OFICINAS GENERALES
7. ENTREGA DE MATERIAL Y CONTROL DE ASISTENCIA
8. SALA DE DESCANSO

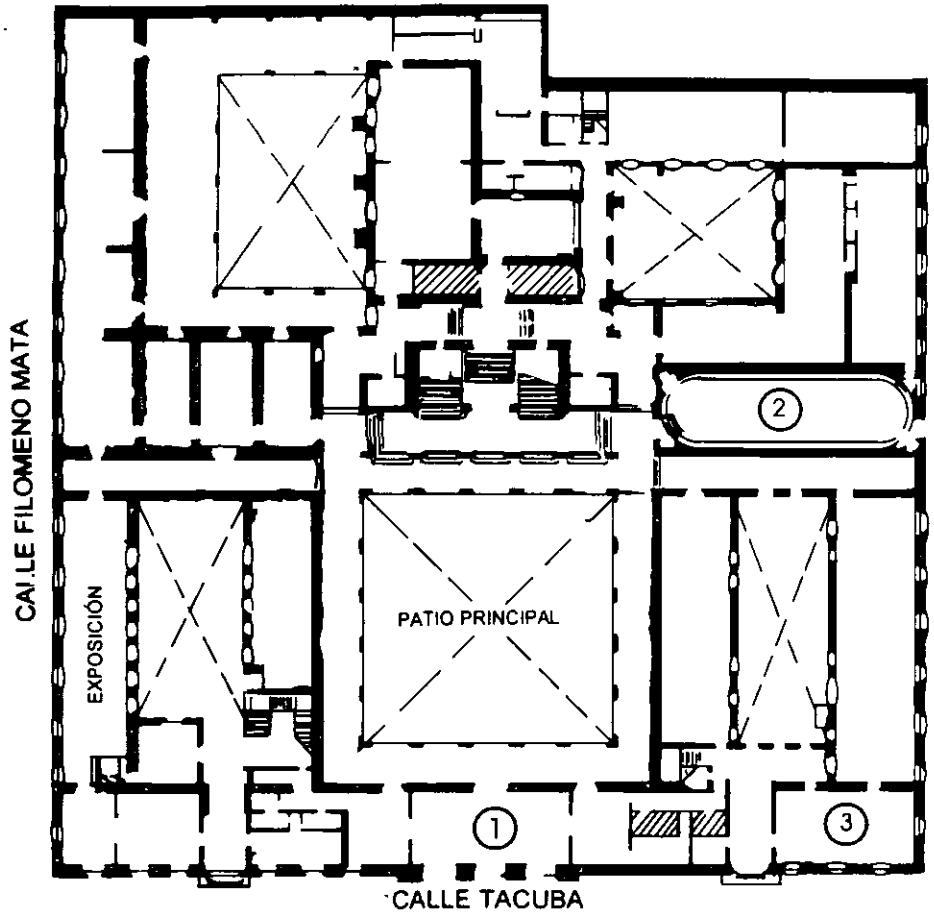
SANITARIOS

* AULAS

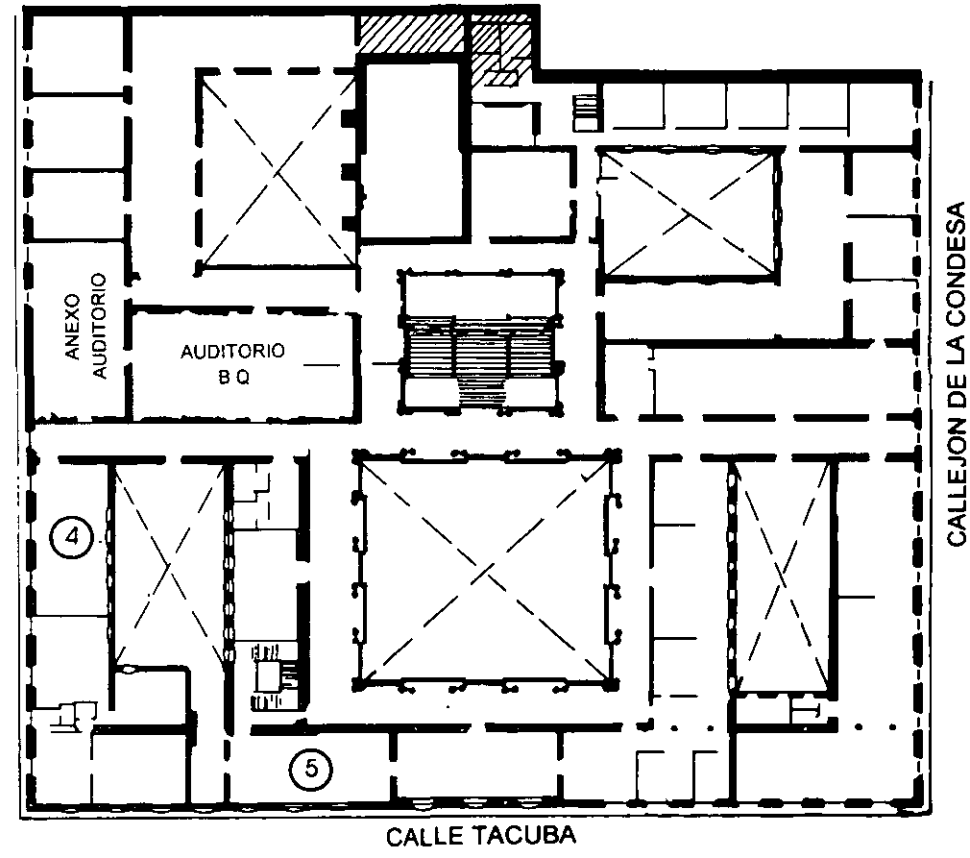
DIVISIÓN DE EDUCACIÓN CONTINUA
FACULTAD DE INGENIERÍA U.N.A.M.
CURSOS ABIERTOS

DIVISIÓN DE EDUCACION CONTINUA
DEC

PALACIO DE MINERIA



PLANTA BAJA



MEZZANINNE

DIVISION DE EDUCACION CONTINUA
FACULTAD DE INGENIERIA, UNAM
CURSOS ABIERTOS

CURSO: cc045 INTRODUCCION AL DISEÑO ORIENTADO A OBJETOS PARA....
FECHA: 23 al 27 DE AGOSTO, 1999

EVALUACIÓN DEL PERSONAL DOCENTE

(ESCALA DE EVALUACIÓN: 1 A 10)

CONFERENCISTA	DOMINIO DEL TEMA	USO DE AYUDAS AUDIOVISUALES	COMUNICACIÓN CON EL ASISTENTE	PUNTUALIDAD
ING. CARLOS ALBERTO ROMAN Z.				

Promedio _____

EVALUACIÓN DE LA ENSEÑANZA

CONCEPTO	CALIF.
ORGANIZACIÓN Y DESARROLLO DEL CURSO	
GRADO DE PROFUNDIDAD DEL CURSO	
ACTUALIZACIÓN DEL CURSO	
APLICACIÓN PRACTICA DEL CURSO	

Promedio _____

EVALUACIÓN DEL CURSO

CONCEPTO	CALIF.
CUMPLIMIENTO DE LOS OBJETIVOS DEL CURSO	
CONTINUIDAD EN LOS TEMAS	
CALIDAD DEL MATERIAL DIDÁCTICO UTILIZADO	

Promedio _____

Evaluación total del curso_____

Continúa...2

1. ¿Le agradó su estancia en la División de Educación Continua?

SI ☐ NO ☐

Si indica que "NO" diga porqué: _____

_____

2. Medio a través del cual se enteró del curso:

Periódico *La Jornada*	
Folleto anual	
Folleto del curso	
Gaceta UNAM	
Revistas técnicas	
Otro medio (Indique cuál)	

3. ¿Qué cambios sugeriría al curso para mejorarlo?

_____

_____

_____

_____

_____

_____

4. ¿Recomendaría el curso a otra(s) persona(s) ?

SI ☐ NO ☐

5.¿Qué cursos sugiere que imparta la División de Educación Continua?

_____

_____

_____

_____

_____

_____

6. Otras sugerencias:

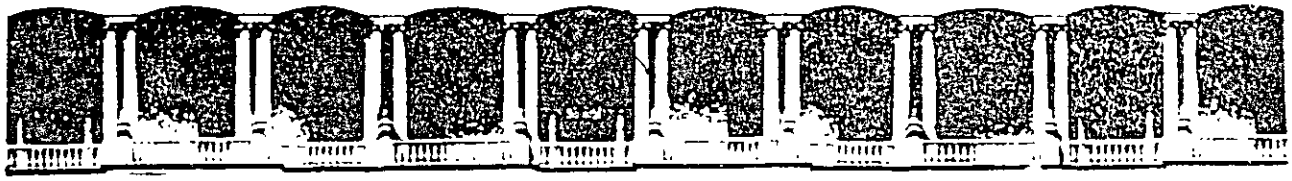
_____

_____

_____

_____

_____



FACULTAD DE INGENIERIA U.N.A.M.
DIVISION DE EDUCACION CONTINUA

MATERIAL DIDACTICO DEL CURSO

INTRODUCCION AL DISEÑO ORIENTADO

A OBJETOS PARA PROGRAMAR EN JAVA

O.C++

AGOSTO, 1999

Palacio de Mineria Calle de Tacuba 5 Primer piso Deleg. Cuauhtemoc 06000 Mexico D.F. APDO. Postal M-2285
Telefonos: 512-8955 512-5121 521-7335 521-1987 Fax 510-0573 521-4020 AL 26

1. INTRODUCCIÓN

La velocidad a la que avanza la tecnología de hardware de computadoras es sorprendente; año con año se logran avances que conducen a la construcción de computadoras más veloces, más compactas y más baratas. Sin embargo, en el aspecto de software no parece haber un desarrollo similar. Mientras los costos de hardware han disminuido continuamente, los de software han hecho lo contrario. La construcción de software no es una tarea fácil y en muchas ocasiones los proyectos de programación sobregiran los presupuestos de tiempo y dinero, lo que da lugar a la crisis del software.

El software es solicitado para ejecutar las tareas demandantes de hoy y está presente en todos los sistemas que van desde los más sencillos hasta los de misión crítica. Las aplicaciones de software son complejas porque modelan la complejidad del mundo real. En estos días, las aplicaciones típicas son muy grandes y complejas para que un individuo las entienda y, por ello, lleva gran tiempo implementar software.

1.1 El papel de la descomposición

"*Divide et impera*", "*Divide y vencerás*".

Cuando se diseña un sistema de software complejo, es esencial dividirlo en componentes más pequeños, cada uno de los cuales podemos analizar detallada e independientemente. De esta manera, satisfacemos la limitación real que existe sobre la percepción humana: para entender algún nivel dado de un sistema, necesitamos solamente comprender unas cuantas partes, en lugar de todas las partes del sistema.

La *modularidad* es una de las herramientas de diseño más poderosas para facilitar el desarrollo y mantenimiento de sistemas de software. La modularidad permite definir un sistema complejo en términos de unidades más pequeñas y manejables; cada una de esas unidades (o módulos) se encarga de manejar un aspecto local de todo el sistema, interactuando con otros módulos para cumplir con el objetivo global.

1.2 El problema de mantenimiento de software

La construcción de software ha recibido la atención de los expertos desde hace mucho tiempo; en la década de los 70's se consiguieron avances significativos hacia el desarrollo de metodologías de forma sistemática y a bajo costo. Como resultado de esos esfuerzos surgieron algunas técnicas que, como el diseño estructurado y el desarrollo descendente (top-down), han sido las herramientas utilizadas por los programadores para construir software. Aunque dichas técnicas han sido empleadas durante mucho tiempo en proyectos realmente complejos, la mayoría de los ingenieros de software coinciden en afirmar que sufren de tres grandes deficiencias:

- Los productos que resultan al emplear éstas técnicas son poco flexibles.
- Los programadores que las usan tienden a concentrarse en el diseño y la implementación del sistema, sin tomar en cuenta su vida posterior.
- No alientan al programador a aprovechar el trabajo de proyectos anteriores.

La desventaja de utilizar una metodología que se concentra en el diseño inicial del sistema se hace evidente si se toma en cuenta que la vida útil de un producto de software puede ser cinco o seis veces más grande que el lapso en que se desarrolla; por ejemplo, un sistema

que se desarrolla en uno o dos años puede mantenerse trabajando durante un período que va de cinco a quince años. Los gastos que se hacen durante el último período (gastos de mantenimiento) representan alrededor del 70% del costo total del sistema. La siguiente ilustración (Figura 1) muestra la forma en que se distribuyen estos gastos.

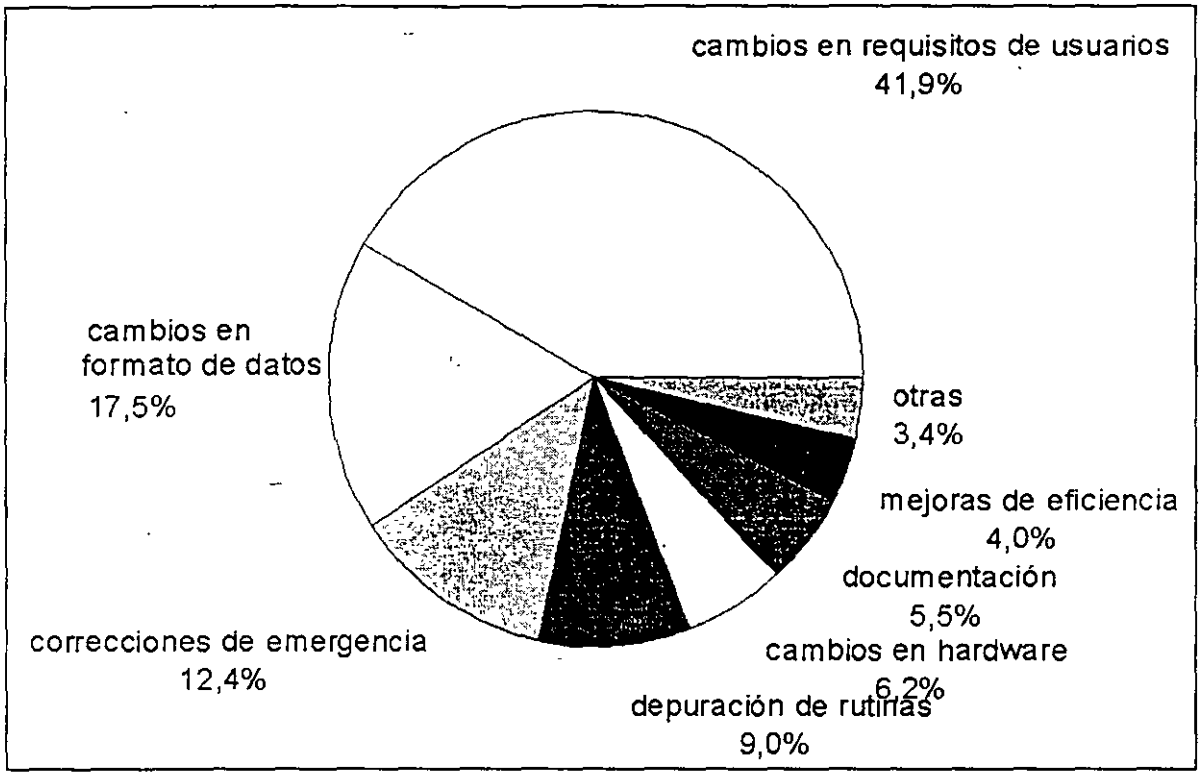


Figura 1

La gráfica muestra que cerca del 60% de los costos de mantenimiento de un sistema (alrededor del 42% del costo total) se tiene que hacer por cambios en las especificaciones del usuario o en los formatos de los datos. Es por ello que la *extensibilidad*, la facilidad con que se modifica un sistema para que se realicen nuevas funciones, debe ser uno de los objetivos primarios de la etapa de diseño.

La fase de mantenimiento es tan importante que cualquier método de diseño debe tener como objetivo principal producir sistemas que faciliten su propio mantenimiento. Pero, ¿cómo alcanzar éste objetivo?. Los expertos recomiendan una serie de actividades y heurísticas que pueden servir como guía durante el desarrollo del sistema. La **Tabla 1** agrupa tales actividades de acuerdo a la etapa de desarrollo en que se deben realizar; más tarde nos preocuparemos con mayor cuidado de la *modularidad*, el *ocultamiento de la información* y la *abstracción de datos.*

Actividades de análisis Establecimiento de estándares (formatos de documentos, codificación, etc.). Especificar procedimientos de control de calidad. Identificar probables mejoras del producto. Estimar recursos y costos de mantenimiento.
Actividades de diseño Establecer la claridad y modularidad como criterios de diseño. Diseñar para facilitar probables mejoras. Usar notaciones estandarizadas para la documentación, algoritmos, etc. Seguir los principios de ocultamiento de información, abstracción de datos y descomposición jerárquica de arriba hacia abajo. Especificar efectos colaterales de cada módulo.
Actividades de implementación Usar estructuras de una sola entrada y una sola salida. Usar sangrado estándar en las diferentes estructuras. Usar un estilo de codificación simple y claro. Usar constantes simbólicas para asignar parámetros a las rutinas. Proporcionar prólogos estándares de documentación en cada módulo.
Otras actividades Desarrollar una guía de mantenimiento. Desarrollar un juego de pruebas. Proporcionar la documentación del juego de pruebas.

Tabla 1. Actividades que facilitan el mantenimiento de un sistema

La *reutilización de código* es otro de los factores que no se toma en cuenta para los métodos de diseño tradicionales. Cualquier programador que desarrollla un sistema nuevo debe escribir una buena cantidad de código que ha escrito anteriormente una y otra vez. Las rutinas de búsqueda, ordenamiento, manejo de menús y despliegue de ventanas, entre otras, se repiten continuamente en proyectos diferentes. Los métodos de desarrollo de software deben alentar al programador a utilizar el código escrito previamente ya sea por él mismo o por otros programadores.

Tradicionalmente, se han empleado tres tipos de reutilización: de personal, de diseño y de código fuente. Sin embargo, estas tres formas de reutilización se han empleado en forma muy limitada, por lo que hay que buscar técnicas más generales, entre las que destaca el *Paradigma Orientado a Objetos.*

1.3 El Paradigma Orientado a Objetos

A medida que se acercaban los años 80, el Paradigma Orientado a Objetos para la ingeniería del software comenzaba a madurar como un enfoque de desarrollo de software. Empezamos a crear diseños de aplicaciones de todo tipo utilizando la forma de pensar orientada a los objetos e implementamos (codificamos) programas utilizando lenguajes y técnicas orientadas a los objetos.Sin embargo, el análisis de requisitos, es decir, la relación entre la asignación de software a nivel de sistema y el diseño del software, se quedó atras.

El Paradigma Orientado a Objetos presenta características que lo hacen idóneo para el análisis, diseño y programación de sistemas. Algunas de las razones para utilizar dicho paradigma se explican en los siguientes párrafos.

El mundo real es un lugar complejo en el sentido de que todas las cosas, tangibles e intangibles, son complejas si las analizamos a partir de sus componentes más básicos. Se dice que el Paradigma Orientado a Objetos es más natural que el tradicional, el Paradigma Estructurado, y es cierto en dos sentidos. En un sentido, dicho paradigma es más natural porque nos permite organizar la información de una forma similar para nosotros. En el otro sentido es más natural porque refleja las técnicas de la naturaleza para manejar la complejidad. Para entender mejor lo anterior, veamos la manera, de forma más general, en que la naturaleza le da poder al concepto de objeto.

El bloque básico de construcción en la naturaleza, y de la cual está compuesta toda la variedad de seres vivos que existimos sobre la Tierra, es la célula. Si observamos la estructura interna de la misma (Figura 2), podemos hacer las siguientes conclusiones:

1. La célula está protegida por una membrana que controla el acceso a su interior. De hecho, la membrana encapsula a la célula, protegiendo su forma interna de trabajar de la intromisión externa.

2. La célula contiene formas de manejar información y comportamiento. La mayoría de la información que define su comportamiento se encuentra en el núcleo, en el centro de la célula (DNA y RNA). El comportamiento, por ejemplo la síntesis de proteinas y la conversión de energía, se lleva a cabo en el cuerpo medio de la célula.

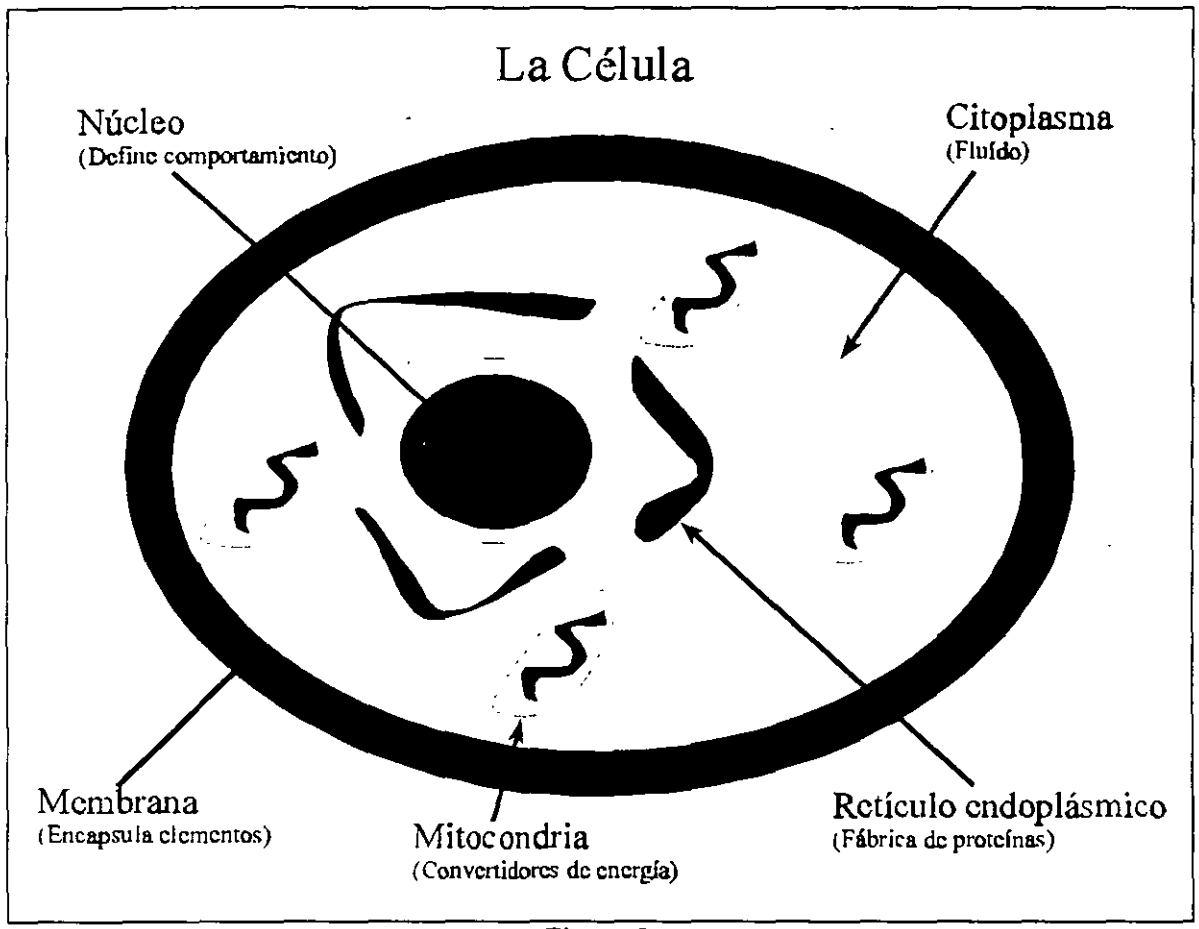


Figura 2

La interacción de las células se realiza a través de la membrana ya que es ésta la que permite intercambiar mensajes químicos con otras células. Una célula puede o no permitir el intercambio del mensaje químico. La membrana representa, entonces, su interfaz con el exterior. Esta comunicación basada en mensajes simplifica enormemente la forma en que las células realizan su función. No es necesario que una célula conozca el funcionamiento interno de otra; lo único que tiene que hacer es enviarle el mensaje apropiado para que la célula receptora ejecute esa acción. Cuando una célula muere, por ejemplo, se liberan sustancias (mensajes) con las que otras células se dan cuenta; entonces una de ellas se divide en dos nuevas células para sustituir a la que murió, regenerándose así los tejidos.

Como podemos apreciar, la interacción entre células está basada en mensajes. Esto quiere decir que cuando una célula quiere afectar el comportamiento de otras células, aquélla les envía un mensaje químico que provoca que éstas modifiquen su comportamiento al realizar cierta acción. Cuando una célula recibe un mensaje que tiene significado para ella, responde al estímulo de acuerdo con la información que tiene codificada en su núcleo.

En pocas palabras, las células no interactúan entre sí interfiriendo con el DNA de las otras. Las células se ocupan de sus propias actividades y hacen requerimientos de una manera ordenada, en lugar de tratar de controlar la operación interna de las otras células directamente. Esta situación ofrece dos formas diferentes de protección:

1. Protege el interior de cada célula de ser modificado por otras. Cualquier célula viviente es mucho más compleja que la mayoría de los sistemas de cómputo jamás creados. Al proteger el interior de la célula del mundo externo, mantiene su integridad.

2. Protege a otras células al no conocer el manejo interno de dichas células.

La segunda ventaja es un ejemplo vivo de lo que es el ocultamiento de la información. De hecho, las células funcionan tan bien como módulos en sistemas complejos como los seres vivos, gracias a que protegen al resto del sistema del tener que lidiar con la complejidad interna. Cuando no se respeta esta independencia entre células y la implementación interna, la célula enferma y muere. Veamos el problema con un virus; el virus se interna en la célula y la ataca desde adentro. Dado que el virus está arraigado en la célula, no puede ser destruido por los glóbulos blanco porque destruiría también a la célula.

Así como la célula del ejemplo anterior, de igual manera es de complejo el mundo que rodea al hombre. Típicamente, el ser humano entiende el mundo que lo rodea por la construcción de modelos mentales, de porciones o fragmentos de ese mundo, para tratar de entender las cosas con las que él interactúa. En esencia, este proceso de construcción de modelos es semejante al de diseño de software.

Además, el modelo mental debe ser más simple que el sistema al que está modelando o dicho modelo será inútil. El ser humano debe abstraerse para poder entender el sistema y realizar una migración, de éste, al modelo mental considerando sólo información cuidadosamente seleccionada e ignorando características irrelevantes. Este proceso de *abstracción* es psicológicamente natural y necesario para que nosotros entendamos el mundo que nos rodea.

1.4 La Abstracción como herramienta en el Paradigma Orientado a Objetos

La abstracción es esencial para el funcionamiento de la mente humana, una poderosa herramienta para tratar la complejidad y la llave para diseñar buen software.

Históricamente, los programadores enviaban instrucciones binarias a una computadora agrupando interrupciones en su panel frontal. Los mnemínicos del lenguaje ensamblador eran abstracciones diseñadas para eliminar la necesidad de recordar las secuencias de bits de las cuales estaban compuestas las instrucciones. En cambio, los programadores realizaron una abstracción de esas secuencias de bits y les agregaron un nombre, llegando a un nivel de abstracción muy alto.

El siguiente uso de la abstracción se dió en la habilidad para crear instrucciones definidas por el usuario. El conjunto de instrucciones de una máquina dada podía ser extendido agrupando secuencias de esas instrucciones primitivas en macro-instrucciones y agregándoles un nombre. Un conjunto de instrucciones optimizado podía, entonces, ser invocado por una macro-instrucción. Una sola de las macro-instrucciones puede lograr que una máquina haga muchas cosas.

Los lenguajes de programación de alto nivel permitieron a los programadores separarse de la arquitectura específica de una computadora dada. Ciertas secuencias de instrucciones fueron reconocidas como universalmente útiles y los programas fueron escritos en términos de éstas. Cada instrucción podía invocar una variedad de instrucciones-máquina, dependiendo de la máquina específica para la cual los programas fueron compilados. Esta abstracción permitió a los programadores escribir software para un propósito genérico, sin preocuparse por la máquina que ejecutaría el programa.

La programación estructurada abarcó el uso de abstracciones de control tales como los ciclos o sentencias if-else que fueron incorporados a lenguajes de alto nivel. Ésto permitió a los programadores abstraer condiciones comunes para cambiar la secuencia de la ejecución.

Más recientemente, los tipos de datos abstractos han permitido a los programadores escribir código sin precuparse por la forma específica en que los datos serán representados. En cambio, los programadores pueden codificar al nivel abstracto en el que pudieran estar realizados los datos. Los detalles de su representación están escondidos; como la estructura

de datos es implementada puede considerarse a un detalle de bajo nivel en el que no es necesario preocuparse por quién diseña y estructura el programa. Por ejemplo, un conjunto puede ser definido como una colección no ordenada de elementos entre los cuales no hay duplicados. Utilizando esta definición, nosotros podemos especificar las operaciones que pueden ser ejecutadas en un conjunto sin especificar si sus elementos están almacenados en un arreglo, una lista ligada o alguna otra estructura de datos.

El *Diseño Orientado a Objetos* (DOO) descompone un sistema en objetos, el componente básico del diseño. Este uso de objetos permite la adición de otros mecanismos de abstracción.

1.5 Algunos conceptos del Paradigma Orientado a Objetos

De todo lo anterior se puede concluir que el Paradigma Orientado a Objetos ha derivado de los paradigmas anteriores a éste. Así como los métodos de diseño estructurado realizados guían a los desarrolladores que tratan de construir sistemas complejos utilizando algoritmos como sus bloques fundamentales de construcción, similarmente los métodos de diseño orientado a objetos han evolucionado para ayudar a los desarrolladores a explotar el poder de los lenguajes de programación basados en objetos y orientados a objetos, utilizando las clases y objetos como bloques de construcción básicos.

Actualmente el modelo de objetos ha sido influenciado por un número de factores no sólo de la Programación Orientada a Objetos, POO (Object Oriented Programming, OOP por sus siglas en inglés). Además, el modelo de objetos ha probado ser un concepto uniforme en las ciencias de la computación, aplicable no sólo a los lenguajes de programación sino también al diseño de interfaces de usuario, bases de datos y arquitecturas de computadoras por completo. La razón de ésto es, simplemente, que una orientación a objetos nos ayuda a hacer frente a la inherente complejidad de muchos tipos de sistemas.

El Paradigma Orientado a Objetos representa, entonces, un desarrollo evolucionario y no revolucionario. No rompe con los avances del pasado pero construye sobre éstos. Desafortunadamente, muchos programadores hoy en día están utilizando, formalmente e informalmente, los principios de diseño estructurado por lo que, hasta la fecha, ambos tipos de diseño de sistemas han coexistido.

Debido a que el modelo de objetos deriva de diversas fuentes, ha sido acompañado de una confusión de terminologías, por lo aquí se tratará de unificar conceptos.

De manera informal, se define a un objeto como "*una entidad tangible que muestra alguna conducta bien definida*" [1].

Los objetos sirven para unificar ideas de abstracción logarítmica y de datos; además tienen una cierta "integridad" la cual no deberá ser violada. En particular, un objeto puede solamente cambiar estado, conducta, ser manipulado o estar en relación con otros objetos de manera apropiada a este objeto.

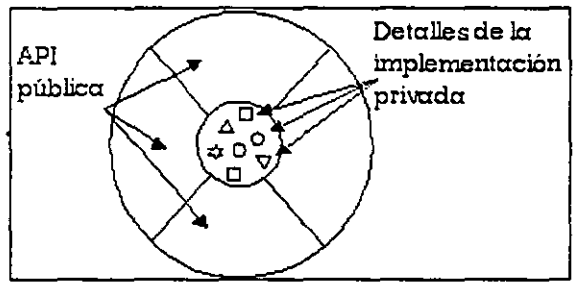
Actualmente, el *Análisis Orientado a Objetos* (AOO) va progresando poco a poco como método de análisis de requisitos por derecho propio y como complemento de otros métodos de análisis. En lugar de examinar un problema mediante el modelo clásico de entrada-proceso-salida (flujo de información) o mediante un modelo derivado exclusivamente de estructuras jerárquicas de información, el AOO introduce varios conceptos nuevos. Estos conceptos nuevos le parecen inusuales a mucha gente, pero son bastante naturales. Considerando este hecho, Coad y Yourdon escriben: "*El AOO —Análisis Orientado a Objetos— se basa en conceptos que una vez aprendimos en la guardería: objetos y atributos, clases y miembros, todo y parte. Nadie sabe por qué hemos tardado tanto tiempo en aplicar estos conceptos en el análisis y la especificación de los sistemas de información —quizás hemos estado demasiado ocupados 'siguiendo el flujo' durante nuestros análisis estructurados diarios, para ponernos a considerar otras alternativas*" [2].

Algunos conceptos que se utilizan en el Paradigma Orientado a Objetos son los siguientes:

Una *clase* es un plantilla para objetos multiples con características similares. Las clases comprenden todas esas características de un conjunto particular de objetos. Cuando se escribe un programa en lenguaje orientado a objetos, no se definen objetos verdaderos sino se definen clases de objetos.

Una *instancia* de un objeto es otro término para un objeto real. Si la clase es la representación general de un objeto, una instancia es su representación concreta.

En los lenguajes orientados a objetos, cada clase está compuesta de dos cualidades: *atributos* (estado) y *métodos* (comportamiento o conducta) como se observa en la figura 3.

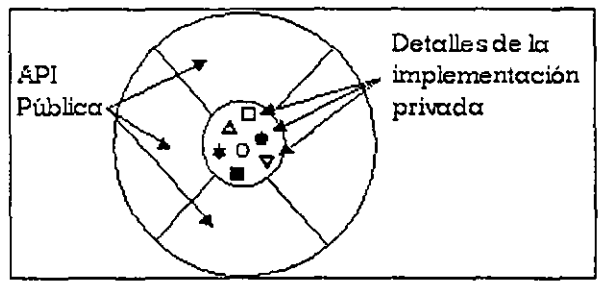


Representación visual de una clase como componente de software

Figura 3

Los atributos son las caracteristicas individuales que diferencian a un objeto de otro y determinan la apariencia, estado u otras cualidades de ese objeto. Los atributos de un objeto incluyen información sobre su estado.

Los métodos de una clase determinan el comportamiento o conducta que requiere esa clase para que sus instancias puedan cambiar su estado interno o cuando dichas instancias son llamadas para realizar algo por otra clase o instancia. El comportamiento es la única manera en que las instancias pueden hacerce algo a sí mismas o tener que hacerles algo (Figura 4).



Representación visual de un objeto como componente de software

Figura 4

Para definir el comportamiento de un objeto, se crean métodos, los cuales tienen una apariencia y un comportamiento igual al de las funciones en otros lenguajes de programación, los lenguajes estructurados, pero se definen dentro de una clase.

Los métodos no siempre afectan a un solo objeto; los objetos también se comunican entre sí mediante el uso de métodos. Una clase u objeto puede llamar métodos en otra clase u objeto para avisar sobre los cambios en el ambiente o para solicitarle a ese objeto que cambie su estado.

Cualquier cosa que un objeto no sabe o no puede hacer es excluída del objeto. Además, como se puede observar de los diagramas, las variables del objeto se localizan en el centro o núcleo del objeto. Los métodos rodean y esconden el núcleo del objeto de otros objetos en el programa. Al empaquetamiento de las variables de un objeto con la protección de sus métodos se le llama *encapsulación*. Típicamente, la encapsulación es utilizada para esconder detalles de implementación no importantes de otros objetos. Entonces, los detalles de implementación pueden cambiar en cualquier tiempo sin afectar otras partes del programa.

Esta imagen conceptual de un objeto —un núcleo de variables empaquetadas en una membrana protectora de métodos— es una representación ideal de un objeto y es el ideal por el que los diseñadores de sistemas orientados a objetos luchan. Sin embargo, no lo es todo: a menudo, por razones de eficiencia o implementación, un objeto puede querer exponer algunas de sus variables o esconder algunos de sus métodos.

La encapsulación de variables y métodos en un componente de software ordenado es, todavía, una simple idea poderosa que provee dos principales beneficios a los desarrolladores de software:

- *Modularidad*, esto es, el codigo fuente de un objeto puede ser escrito, así como darle mantenimiento, independientemente del código fuente de otros objetos. Así mismo, un objeto puede ser transferido alrededor del sistema sin alterar su estado y conducta.

- *Ocultamiento de la información*, es decir, un objeto tiene una "interfaz publica" que otros objetos pueden utilizar para comunicarse con él. Pero el objeto puede mantener información y métodos privados que pueden ser cambiados en cualquier tiempo sin afectar a los otros objetos que dependan de ello.

En las ilustraciones anteriores de una clase y un objeto, figuras 3 y 4, se observa que son muy similares una de la otra y puede prestarse a una seria confusión. En el mundo real, es obvio que las clases no son, ellas mismas, los objetos que ellas describen o contienen. Sin embargo, es un poco más difícil diferenciar clases y objetos en software porque los objetos de software son, simplemente, modelos electrónicos de objetos del mundo real o conceptos abstractos en primer lugar. Lo anterior sucede porque mucha gente utiliza el término "objeto" irregularmente y lo hace para referirse tanto a clases como instancias.

La figura anterior de la clase, figura 3, no está "sombreada" porque representa un "proyecto", general y detallado, de un objeto en lugar de un objeto mismo. En comparación, un objeto está sombreado indicando que ese objeto actualmente existe y puede ser utilizado.

Los objetos proveen el beneficio de la modularidad y el ocultamiento de la información. Las clases proveen el beneficio de la reusabilidad. Los programadores de software utilizan la misma clase, y por lo tanto el mismo código, una y otra vez para crear muchos objetos.

Otro concepto muy importante en el Paradigma Orientado a Objetos es el de *herencia*. La herencia es un mecanismo poderoso con el cual se puede definir una clase en términos de otra clase; lo que significa que cuando se escribe una clase, sólo se tiene que especificar la diferencia de esa clase con otra, con lo cual, la herencia dará acceso automático a la información contenida en esa otra clase.

Con la herencia, todas las clases están arregladas dentro de una jerarquía estricta. Cada clase tiene una superclase (la clase superior en la jerarquía) y puede tener una o más subclases (las clases que se encuentran debajo de esa clase en la jerarquía). Se dice que las clases inferiores en la jerarquía, las clases hijas, heredan de las clases más altas, las clases padres.

Las subclases heredan todos los métodos y variables de las superclases; es decir, en alguna clase, si la superclase define un comportamiento que la clase hija necesita, no se tendrá que redefinir o copiar ese código de la clase padre (Figura 5).

De esta manera, se puede pensar en una jerarquía de clase como la definición de conceptos demasiado abstractos en lo alto de la jerarquía y esas ideas se convierten en algo más concreto conforme se desciende por la cadena de la superclase.

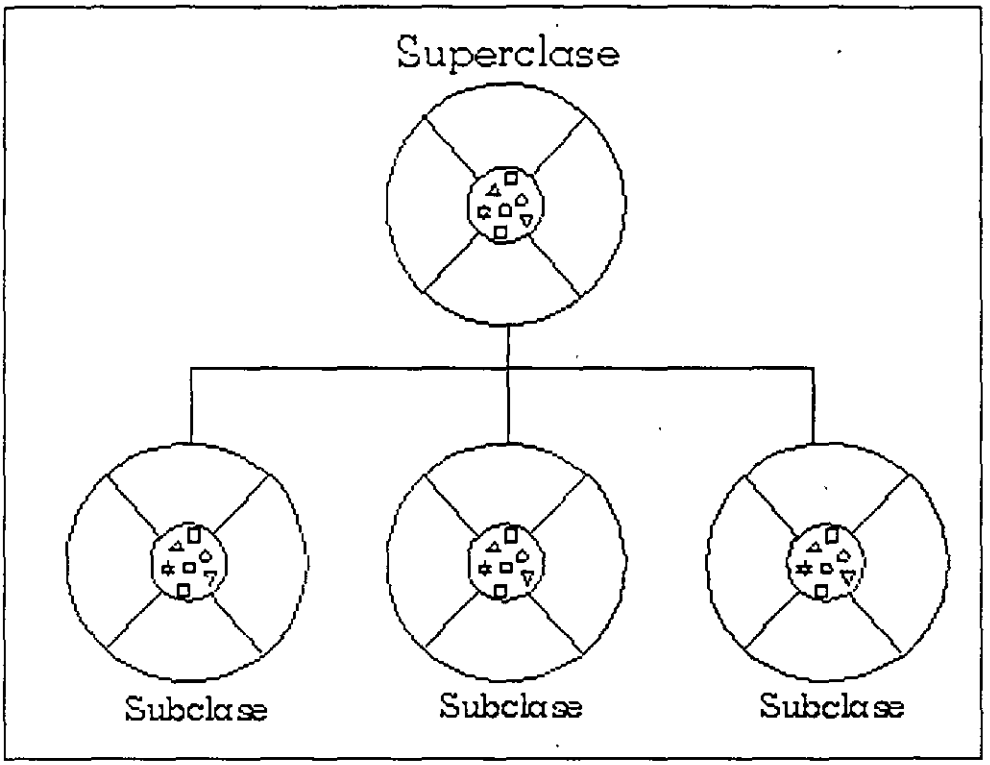


Figura 5

Sin embargo, las clases hijas no están limitadas al estado y conducta provistos por sus superclases; pueden agregar variables y métodos además de los que ya heredan de sus clases padres.

Las clases hijas pueden, también, sobreescribir los métodos que heredan por implementaciones especializadas para esos métodos. De igual manera, no hay limitación a un sólo nivel de herencia por lo que se tiene un árbol de herencia en el que se puede heredar varios niveles hacia abajo y mientras más niveles descienda una clase, más especializada será su conducta.

La herencia presenta los siguientes beneficios:

Las subclases proveen conductas especializadas en base a elementos comunes provistos por la superclase. A través del uso de herencia, los programadores pueden reusar el código de la superclase muchas veces.

Los programadores pueden implementar superclases llamadas clases abstractas que definen conductas "genéricas". Las superclases abstractas definen, y pueden implementar

parcialmente, la conducta pero gran parte de la clase no está definida ni implementada. Otros programadores concluirán esos detalles con subclases especializadas.

1.6 Conclusiones

Se puede, entonces, concluir lo siguiente:
Los programadores trabajan en un lenguaje y utilizan un estilo de programación. Ellos trabajan en un paradigma dependiendo del lenguaje que utilicen. Un estilo de programación, o paradigma, se define como "*una manera de organización de programas de acuerdo a las bases de un modelo de programación conceptual y un lenguaje apropiado para realizar programas escritos en un estilo claro*". Los siguientes, son ejemplos de paradigmas y el tipo de abstracción que cada uno de ellos utiliza:

Paradigmas	**Abstracción utilizada**
Orientado a procedimientos	Algoritmos
Orientado a objetos	Clases y objetos
Orientado a lógicas	Metas, a menudo expresadas en un cálculo de predicado
Orientado a reglas	Reglas if-then
Orientado a constraints	Relaciones invariantes

- **Análisis Orientado a Objetos (OOA)**

Es un método de análisis que examina los requerimientos desde la perspectiva de las clases y objetos encontrados en el vocabulario de la definición del problema.

- **Diseño Orientado a Objetos (OOD)**

Es un método de diseño que abarca el proceso de descomposición orientada a objetos y una notación de representación tanto lógica y física, además de modelos estáticos y dinámicos del sistema a diseñar.

- **Programación Orientada a Objetos (OOP)**

Es un método de implementación en el cual los programas están organizados como colecciones cooperativas de objetos, cada uno de los cuales representa una instancia de alguna clase, y todas esas clases son miembros de una herencia de clases a través de relaciones de herencia.

En la vida real hay objetos, no clases.

Para el Paradigma Orientado a Objetos, existen varias metodologías, entre las que destaca la metodología OMT (Object Modeling Technique), la cual fue diseñada por General Electric.

Características fundamentales que distinguen, o identifican un enfoque OO, según OMT.

1. **Identidad**. Al estar en el dominio del problema los objetos definibles tienen su propia identidad. Datos discretos, únicos y distinguibles.

2. **Clasificación**. Se realiza una distinción entre semejanzas y diferencias; se agrupan, y es el paso para definir clases. Objetos con atributos y comportamientos iguales, que se agrupan en clases.

3. **Polimorfismo**. Se operan objetos de diferentes clases, empleando los mismos métodos. Aplicar un mismo comportamiento de forma diferente a distintas clases de objetos.

4. **Herencia**. Agrupa en una jerarquía de clases el problema para dar solución a un problema de una manera más fácil. Compartir atributos y operaciones de una clase base en una jerarquía de clases.

CICLO DE VIDA DE UN SISTEMA DURANTE SU DESARROLLO, SEGÚN OMT

- **Requerimientos**
  - Reingeniería de procesos
  - Reestructuración
- **Contrato**, en el que se vean reflejadas las condiciones y los requerimientos.
- **Propuesta**
- **Diagnóstico**
  - Explicación, y el diagrama de procesos de la empresa.
  - Manuales de Procedimientos.
  - Diagramas de E/S de información.
  - Cómo se trabajan los procesos
  - Situación actual de la empresa para saber qué se debe modificar.
  - Conclusión.
  - Posibles puntos a automatizar.
- **Análisis**

- **Diseño**
- **Diseño de Objetos**
- **Implementación**
- **Pruebas**
- **Mantenimiento**

2. EL ANÁLISIS

Es un método de análisis que examina los requerimientos desde la perspectiva de las clases y objetos encontrados en el vocabulario de la definición del problema. En el análisis se toman decisiones de alto nivel.

En teoría, el análisis y el diseño abarcan el 70 % de la construcción de un sistema. El 30 % restante se emplea en la implementación del mismo.

En el análisis se llevan a cabo los siguientes pasos:

1. Describir el problema, tomando en cuenta lo siguiente:
   1.1. Abstracción del problema.
   1.2. ¿Cuál es el problema?
   1.3. ¿Dónde está la necesidad?
   1.4. ¿Qué se debe hacer? y no ¿Cómo se debe hacer? ni ¿Cómo se va a hacer?
   1.5. Una vez realizado el sistema, ¿Qué debe hacer?

2. Realizar un resumen escrito del problema, en el cual se pueden utilizar imágenes para que se entienda mejor por terceras personas.

3. Un buen análisis lo entienden los expertos en el dominio de la aplicación, sin necesidad de que sean analistas-programadores.

4. No se deben incluir **nunca** estructuras de datos ni artificios de programación.

En la abstracción del problema, se deben identificar los objetos lo cual resulta complicado cuando se observa el "espacio" del problema de un sistema de software.

Podemos empezar a identificar objetos examinando la descripción del problema o llevando a cabo un "análisis gramatical" de la narrativa del procesamiento del sistema a construir. Determinamos los objetos subrayando cada nombre o claúsula nominal y añadiéndolo en una tabla. Se deben anotar los sinónimos. Si un objeto es necesario para implementar una solución, entonces es parte del "espacio de la solución"; en caso contrario, cuando el objeto sólo es necesario para describir la solución, entonces es parte del "espacio del problema". Ahora bien, de la descripción del problema ¿Cuáles son los objetos?

Los objetos pueden ser:

- *Entidades externas*, por ejemplo: otros sistemas, dispositivos, gente, etc., que producen o consumen información a ser utilizada en el sistema basado en computadora.
- *Cosas*, por ejemplo: informes, visualizaciones, cartas, señales, etc., que son parte del dominio de información del problema.
- *Ocurrencias* o *sucesos*, por ejemplo, una transferencia de una propiedad o la terminación de una serie de movimientos de un robot, etc., que ocurren en el contexto de operación del sistema.
- *Papeles* que juegan las personas que interactúan con el sistema. Por ejemplo: gestor, ingeniero, vendedor, etc.
- *Unidades organizativas*, por ejemplo: división, grupo, equipo, etc., que son relevantes para la aplicación.
- *Lugares*, por ejemplo: sala de facturación, muelle de descarga, etc., que establecen el contexto del problema y del funcionamiento general del sistema.
- *Estructuras*, por ejemplo, sensores, vehículos de cuatro ruedas, computadoras, etc., que definen clases de objetos o, en casos extremos, clases de objetos relacionadas.

En realidad, lo que se definen son clases de las que se pueden instanciar objetos. Por lo tanto, cuando identificamos posibles objetos, también estamos identificando posibles clases, a las cuales se les llama *clases candidato*.

Una vez identificadas las clases candidato, o los posibles objetos, se elabora una lista o tabla con 2 columnas, la primera se coloca la clase u objeto y en la segunda columna se define su clasificación general o nombres relacionados.

Por ejemplo, tengamos la siguiente descripción de un sistema de software

> El software HogarSeguro permite al propietario de la vivienda configurar el sistema de seguridad al instalarlo, controla todos los sensores conectados al sistema de seguridad e interactúa con el propietario a través de un teclado numérico y unas teclas de función que se encuentran en el panel de control de HogarSeguro.
> Durante la instalación se utiliza el panel de control para "programar" y configurar el sistema . Cada sensor tiene asignado un número y un tipo; existe una constraseña maestra para activar y desactivar el sistema, y se introduce(n) un(os) teléfono(s) para contactar cuando se produce un suceso detectado por un sensor.
> Cuando el software detecta la sensorización de un suceso, hace que suene una alarma audible que está incorporada en el sistema. Tras un retardo, especificado por el propietario durante la

configuración del sistema, el programa marca un número de teléfono de un servicio de monitorización, proporciona información sobre la situación e informa sobre la naturaleza del suceso detectado. Cada 20 segundos se volverá a marcar el número de teléfono hasta que se consiga establecer la comunicación.
Toda la interacción con HogarSeguro está gestionada por un subsistema de interacción con el ususario que lee la información introducida a través del teclado numéricoy las teclas de función; muestra mensajes de petición en un monitor y muestra información sobre el estado del sistema en el mismo monitor. La iteacción por teclado toma la siguiente forma...

En base al enunciado anterior, se debe realizar la tabla de objetos y clases candidato. Coad y Yourdon sugieren seis características selectivas que debe utilizar el analista para considerar la inclusión o no de cada objeto potencial en el modelo de análisis:

- *Información retenida.* El objeto potencialserá útil durante el análisis sólo si la información sobre el mismo ha de ser recordada para que el sistema pueda funcionar.
- *Servicios necesarios.* El objeto potencial debe tener un conjunto de operaciones identificables que puedan cambiar de alguna forma el valor de sus atributos.
- *Múltiples atributos.* Durante el análisis de requisitos, se debe centrar la atención a la información "principal"; un objeto con un sólo atributo puede resultar útil durante el diseño, pero probablemente se represente mejor como atributo de otro objeto en la fase del análisis.
- *Atributos comunes.* Se pueden definir conuntos de atributos para los objetos potenciales y aplicar así esos atributos a todas las ocurrencias del objeto.
- *Operaciones comunes.* Se pueden definir conjuntos de operaciones para los objetos potenciales y aplicar así esas operaciones a todas las ocurrencias del objeto.
- *Requisitos esenciales.* En el modelo de requisitos casi siempre se definirán como objetos las entidades externas que aparecen en el espacion del problemay que producen o consumen información esencial para el funcionamiento de cualquier solución que se desarrolle en el sistema.

Para que un objeto sea considerado como legítimo para ser incluído en el modelo de requisitos, el objeto potencial debe satisfacer todas, o casi todas, las características anteriores. La decisión de incluir o no un objeto potencial en el modelo de análisis puede resultar algo subjetiva, haciendo que un posterior evaluación pueda descartar o reinstanciar a un objeto determinado. Sin embargo uno de los primeros pasos en el AOO debe consistir en la definición de los objetos, debiendo llevar a cabo decisiones, aunque sean subjetivas.