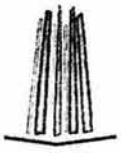
Universidad Nacional Autónoma de México



Escuela Nacional de Estudios Profesionales "ARAGON"

"Integración de Ingeniería de Software con el modelaje Orientado a Objetos utilizando UML"

Tesis profesional que para obtener el título de:

Ingeniero en Computación

Presenta:

Daniel Benjamin Alcántara Zavala

Asesor de Tesis:

Ing. Gladis Fuentes Chávez

San Juan de Aragón, Edo. Mex. 2004

AGRADECIMIENTOS

A mi pareja, cuya fortaleza y capacidad de amar, me asombran y me enamoran siempre.

A mi mamá por su gran amor y ejemplo de lucha implacable, símbolo de mi admiración.

A mi papá, por su infinita paciencia, amor y dedicación a todos nosotros, su familia.

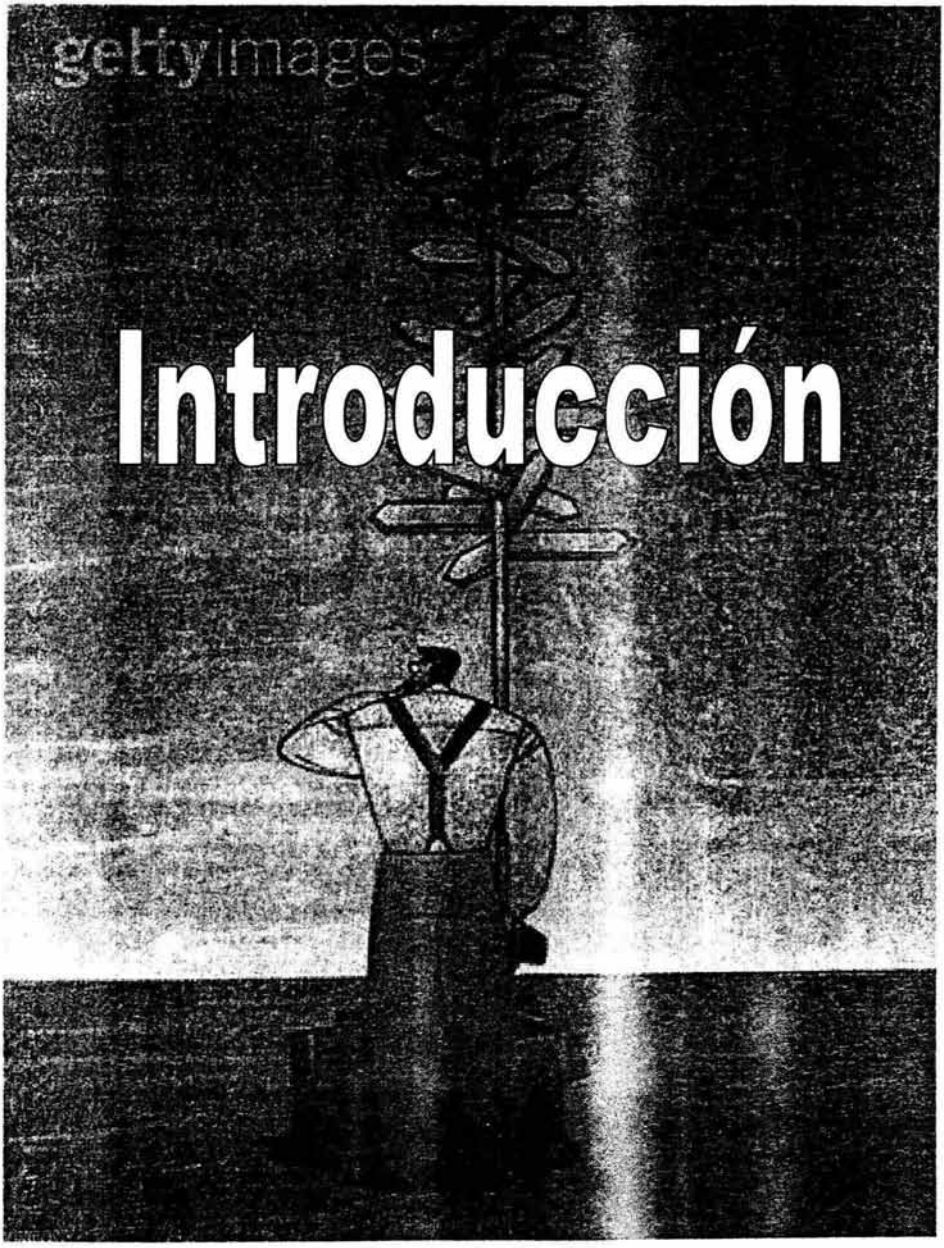
A mi hermano, quien me apoyó y guió durante mi juventud.

A mis profesores, por que de ellos parte soy.

A mi asesor Gladis Fuentes Chávez, por creer y confiar en mí en todo momento.

A mi padres le dedico este esfuerzo diciéndoles que su cachorro siempre estará orgulloso de ustedes. Gracias por su ejemplo.

Integración de Ingeniería de Software con el modelaje Orientado a Objetos utilizando UML


gettyimages
Introducción

La ingeniería de software es todo lo referente a producir lo que el cliente desea dentro del tiempo y costos requeridos. Un producto de calidad necesita una combinación de habilidades técnicas y organizacionales. Por lo que surge una pregunta: ¿Qué tan importante es la calidad del software?.

Existen dos formas de comprender mejor esto:

- Una es en términos de conocer el daño que se puede producir por un software defectuoso, como por desgracia se conocen casos en equipos militares, sistemas de auto-aterrizaje en la aviación, reactores nucleares o los tan comunes sistemas bancarios, entre otros.

- La segunda[1] se basa tan sólo en el costo de producir y mantener el software, pues se sabe que:
  - En promedio los sistemas grandes de software son entregados un año después de la fecha acordada.
  - Solo un 1% de los proyectos principales de software se termina a tiempo y dentro del presupuesto.
  - El 25% de todos lo proyectos intensivos de software nunca se terminan en su totalidad.
  - Más del 60% de los gerentes de producción de software tienen poca o ninguna experiencia en la práctica de ingeniería de software moderna.

Las metodologías convencionales de Ingeniería de Software (IS) tienen mecanismos robustos para hacer un análisis de necesidades y diseño completo. Para hacer uso efectivo de la información recolectada en las fases de análisis y diseño se propone el uso del modelo orientado a objetos en todas las etapas del ciclo de desarrollo y así unificar los términos en los que se habla en cada etapa, estableciendo un modelo del problema y de su comportamiento; de esta forma se hace referencia a objetos en el modelo, de tal manera que al llegar a la implementación, los resultados obtenidos se transcriben al lenguaje de programación deseado, con el simple hecho de interpretar la sintaxis del modelo, que de antemano satisface las necesidades y requerimientos del ser humano.

El objetivo de este trabajo es integrar el modelaje orientado a objetos (OO) con la metodología de Ingeniería de Software (IS) bajo el Lenguaje de Modelaje Unificado (UML), para enriquecer cada uno de los procesos de desarrollo de un sistema.

[1] Aproximaciones a la Fiabilidad del Software.- Enrique A. Chaparro, International Association of Cryptologic Research y Fundación Vía Libre, Argentina (echaparro@vialibre.org.ar)

Como punto de partida se debe identificar las características que deberá poseer un sistema. A partir de allí se hace la adaptación y/o redefinición de los pasos que debe seguir una metodología de IS, en base a los diagramas establecidos en UML.

A Continuación mencionaremos algunos elementos de los cuales hablaremos a detalle en el transcurso de los siguientes capítulos.

El desarrollo del software está fuertemente ligado a la evolución de los sistemas informáticos. Un mejor rendimiento del hardware, una reducción del tamaño y un costo más bajo han dado lugar a sistemas informáticos más sofisticados. Ahora nos enfrentamos al problema de mejorar la calidad del software y de reducir su costo.

En el mundo de los PC's el hardware se ha convertido en un producto estándar, siendo el software suministrado con ese hardware lo que marca la diferencia.

En los inicios de la Informática la mayoría del tiempo se intentaba mejorar el hardware. Actualmente el hardware ha alcanzado niveles tales que al automatizar un proceso lo que más cuenta es el Software (figura 1).

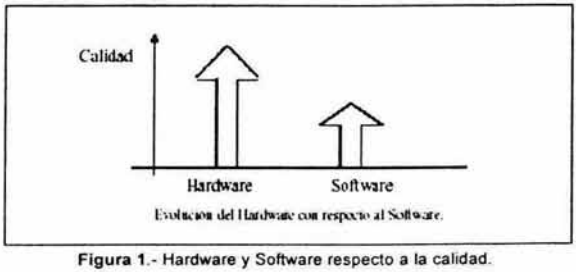


Figura 1.- Hardware y Software respecto a la calidad.

La Ingeniería del Hardware se ha dedicado a desarrollar un hardware de bajo costo y alta calidad y una vez conseguido esto, lo que se intenta ahora es realizar lo mismo con el Software.

En los comienzos de la informática, la programación se veía como un "arte", existían pocos métodos formales y pocas personas los usaban. El programador aprendía mediante la técnica de *"ensayo y error"*. Actualmente, el software es el elemento de principal costo. Por ello la Ingeniería del Software busca conseguir un software de calidad a un precio razonable, a la vez de entregar el producto en los plazos establecidos.

Características del Software.

El proceso creativo del hardware lleva a la obtención de una forma física, el software es un elemento que no es tangible o bien es lógico, ya que no es físico como el hardware. Por ello tiene unas características propias:

- El Software se desarrolla, no se fabrica en un sentido clásico.
- El Software no se estropea, los agentes externos (temperatura, humedad, etc) no le afectan.
- En el Hardware existe una zona de mortalidad infantil hasta llegar a una zona de pocos fallos en la que se estaciona, para luego volver a empezar a fallar por envejecimiento de este. Con el Software en cambio solo existen problemas durante el desarrollo y puesta a punto de este.

Para conseguir un producto de calidad se debe tener un buen diseño. Los costos del software se encuentran en la ingeniería, el software no es un producto de fabricación. Actualmente se tiende a utilizar las herramientas llamadas "Ingeniería del Software Asistida por Computadora" y por sus siglas CASE (Computer Aided Software Engineering).

Es obvio pensar que a medida que pasa el tiempo el software tiende a ser más estable, razón por la cual al realizar un cruce entre el *"índice de fallos"* a través del *"tiempo"* observaríamos una curva como en la figura 2.

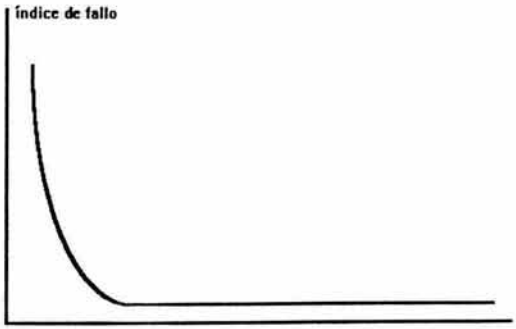


Figura 2.- Comportamiento del software a través del tiempo.

La curva comentada anteriormente sería la ideal, pero normalmente se suelen efectuar cambios en el Software y cuantos más cambios se produzcan más se incrementaran los fallos

hasta que se llegue a un punto en el que sea más rentable comenzar otro programa nuevo que adaptar el que tenemos, como lo observamos en la figura 3.

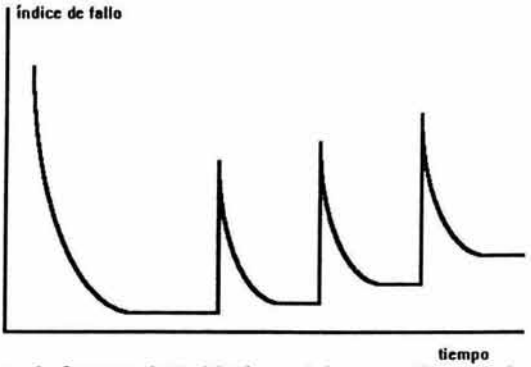


Figura 3.- Comportamiento del software al efectuar cambios en el mismo.

El costo para solucionar fallos en el Software es más elevado que en el Hardware.

- La mayoría del Software se construye a medida en vez de utilizar componentes existentes. Se puede comprar software ya desarrollado para casos muy particulares.

Aplicaciones del Software.

Es difícil establecer categorías genéricas para las aplicaciones del software, ya que conforme aumenta su complejidad es más difícil de encasillar en una u otra categoría. A continuación se enuncian las siguientes categorías con las que podríamos globalizar a éstas:

- Software de Sistemas: Conjunto de programas creado como herramienta para otros programas.(Sistemas Operativos (SO), Compiladores, CASE, Editores, etc.). Se caracteriza por existir una fuerte interacción con el hardware de las computadoras.

- Software de tiempo real: Conjunto de programas que mide, analiza y controla sucesos del mundo real a medida que ocurren. Se caracterizan por los siguientes componentes:

    - Adquisición de Datos: recoge y formatea los datos que le entran.

    - Análisis: transforma la información dependiendo de la aplicación.

    - Control/Salida: responde a la entrada (entorno externo).

  - Monitorización: coordina el resto de los componentes, de forma que pueda mantenerse la respuesta en tiempo real (entre 1 milisegundo y 1 segundo).

- Software de gestión: Realiza tareas de procesamiento de datos y operaciones de cálculo interactivo (aplicaciones de Administración de empresas).

- Software de Ingeniería y Científico: Utiliza algoritmos de manejo de números (Astronomía, Cálculo, Biología molecular, Fabricación automática). La simulación de sistemas están tomando características de software en tiempo real e incluso de software de sistema.

- Software empotrado: reside en memorias ROM y se utiliza para controlar productos y sistemas de los mercados industriales y de consumo (Lavadoras, Microondas).

- Software para Computadoras Personales (PC): (Procesadores de textos, hojas de cálculo, gestores de Bases de Datos, juegos, aplicaciones financieras, etc.).

- Software de inteligencia artificial: Es aquel que hace uso de algoritmos no numéricos para resolver problemas complejos para los que no son adecuados el cálculo o el análisis directo. Forman parte de este grupo, aplicaciones de sistemas expertos, reconocimiento de patrones (OCR), redes neuronales artificiales, prueba de teoremas, juegos, entre otros.

Principales problemas del Software.

Existen 4 problemas principales que afectan al desarrollo del software:

- La planificación y estimación de costos económicos y temporales son frecuentemente muy imprecisos.

- Productividad del Software.

- Demanda del mercado.

- Baja calidad del Software con relativa frecuencia.

Para solucionar estos problemas podemos combinar métodos completos para todas las fases del desarrollo del software; podemos utilizar herramientas para automatizar estos métodos y mejorar la calidad del software, y así conseguir una disciplina para el desarrollo del Software.

Ingeniería del Software.

La Ingeniería del Software es el establecimiento y uso de sólidos principios de ingeniería orientados a obtener Software económico, fiable y que funcione de una manera eficiente sobre maquinas reales.

La Ingeniería del Software abarca tres elementos clave: métodos, herramientas y procedimientos. Todos ellos facilitan el proceso del desarrollo del software y permiten establecer unas bases para construir software de alta calidad de una forma productiva.

Métodos. Estos nos enseñan cómo construir técnicamente el Software. Estos métodos abarcan:

- Planificación y estimación de proyectos.
- Análisis de requisitos del sistema y del software.
- Diseño de estructuras de datos.
- Diseño de programas y procedimientos algorítmicos.
- Codificación.
- Prueba.
- Mantenimiento.

Herramientas. Son el soporte automático o semiautomático de aplicación de los métodos.

Existen herramientas para cada uno de los métodos anteriores. Cuando la información creada por una herramienta puede ser utilizada por otra se establece un sistema para el desarrollo del software llamado Ingeniería del Software Asistida por Computadora (CASE: Computer Aided Software Engineering). Cada método tiene su propia herramienta. Aplicamos varios pasos para crear un producto de software. La salida de uno es la entrada de otro.

Procedimientos. Son la aplicación de los métodos utilizando las herramientas. Son los elementos de unión entre los métodos y las herramientas. Definen la secuencia en la que se aplican los métodos, ayudan a asegurar la calidad y coordinar los cambios, etc.

Proceso de desarrollo del software.

Así como la Ingeniería del Software está compuesta por métodos, herramientas y procedimientos, en el proceso de desarrollo del software se tiene tres etapas genéricas: definición, desarrollo y mantenimiento, mismos que a continuación explicare:

- **Definición (Qué).** se identifica que información ha de ser procesada, que función y rendimiento se desea, que restricciones de diseño existen, que criterios de validación se necesitan, etc.

  Si pudiéramos establecer los puntos básicos para establecer los requerimientos del software estos serían:

  1. Análisis del sistema: analiza el sistema real asignando la parte a informatizar.

  2. Planificación del proyecto: análisis de riesgos, asignación de recursos, estimación de costos, definición de tareas y planificación del trabajo.

  3. Análisis de requisitos: información detallada del ámbito de información y la función del software.

- **Desarrollo (Cómo).** Consta de tres pasos:

  1. Diseño del software: traduce requisitos a un conjunto de representaciones (gráficas, tablas, etc.). Describe estructuras de datos, arquitectura, procedimiento algorítmico y características del interfase.

  2. Codificación: traduce el diseño a lenguaje de programación.

  3. Prueba del Software: prueba para detectar errores.

- **Mantenimiento.** se vuelven a aplicar los pasos de las fases de la definición y desarrollo, pero en el contexto del software ya existente. Puede haber tres tipos de cambios:

  1. Corrección: Errores de software.

  2. Adaptación: Cambios del entorno original.

  3. Mejora: Funciones adicionales solicitadas por el cliente.

En este sentido se han desarrollado una serie de métodos con la finalidad de establecer la ruta optima para el desarrollo de sistemas, de los cuales destacan el **ciclo de vida de un sistema** o ciclo de resolución del problema.

Es muy útil para organizar el gran número de actividades necesarias en la construcción de un sistema y especificar la secuencia en que se deben tratar esas actividades para su desarrollo.

El ciclo de vida también ayuda a los analistas y diseñadores a resolver problemas que surgen durante el desarrollo del sistema.

Ciclo de vida clásico del software.

El ciclo de vida clásico del software, también es conocido como modelo en cascada. En grandes sistemas no tiene por qué ser la misma persona la que realice todos los pasos, por eso es importante que cada una de las partes esté muy clara y debidamente documentada.

En la realidad esto no es lineal y el ciclo no se suele cumplir. Para que se rompa el ciclo pueden suceder muchas cosas. Las tres primeras fases del ciclo son las más importantes para evitarnos problemas en los siguientes pasos:

- **Análisis.** En esta fase se hace un estudio del sistema. Se hará la recopilación de requisitos tanto del sistema como del software, se documenta todo lo que se ha estudiado y se establece lo que se va ha hacer. Todo esto se debe comentar con el cliente antes de continuar.

- **Diseño.** Una vez que se sabe que hay que hacer, en esta fase se determina como se va ha hacer. Se diseña la arquitectura de los datos, la del software, interfase, etc. También se debe documentar todo el diseño realizado, y no se debe olvidar que es en esta fase donde se establece la calidad del producto.

- **Codificación.** Es la traducción del diseño a un lenguaje de programación. A veces, cuando se esta en esta fase surgen problemas que obligan a volver al análisis o al diseño.

- **Prueba.** se trata de probar si el software obtenido se ajusta a lo que queríamos obtener, si no es así se debe volver a fases anteriores.

- **Mantenimiento.** en esta fase se realizarán cambios, bien por errores que no se hayan detectado antes, por cambios en el entorno (por ejemplo que el usuario cambie de computadora, impresora, etc.) o por ampliaciones a petición del cliente.

Este modelo es el más antiguo y el más utilizado, aunque presenta ciertos problemas:

- Es difícil seguir la linealidad del ciclo.
- Normalmente, el cliente no especifica todos los requisitos.
- El cliente no ve una versión del producto hasta que finaliza el ciclo.

A continuación en la figura 4, se observa el diagrama del ciclo de vida clásico del software.

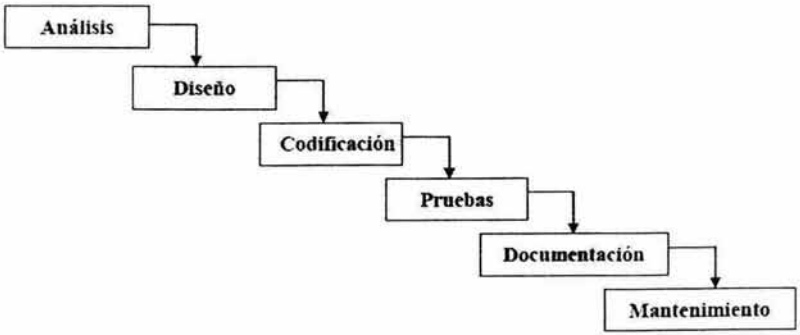


Figura 4.- Ciclo de vida clásico del software.

Como podemos ver éste modelo es la base para crear otros entre los cuales encontramos a:

- Construcción de prototipos.
- Modelo en Espiral.
- Desarrollo Incremental.

La Ingeniería del Software es una disciplina que integra métodos, herramientas y procedimientos para el desarrollo del software.

La difusión de lo "Orientado a Objetos".

En las revistas técnicas abundan las referencias a la "Programación Orientada a Objetos" así como en los "Lenguajes Orientados a Objetos" a tal punto que es común que la gente no preste atención a la palabra "Orientado" o que no note diferencias cuando uno habla de "tecnología de objetos".

Hoy ocurre lo mismo que en los '80 cuando se hablaba de módulos, componentes, y programación estructurada; solo un pequeño grupo de profesionales construían arquitecturas del software.

Hoy se habla de objetos, pero nadie en realidad usa objetos, solo piensan orientado a objetos y en el mejor de los casos logran tener las ventajas del trabajo modular con componentes de software. Debido al costo de implantar una nueva tecnología a nivel masivo, hoy se pueden observar alternativas menos ambiciosas aunque muy difundidas globalmente. La "orientación a objetos" es un conjunto de técnicas de la nueva tecnología de objetos adaptadas a las técnicas tradicionales. Esto permite ir cambiando gradualmente, pero a costos muchos más altos.

La gran ventaja de las "técnicas orientadas a objetos" es que no requieren de experiencia real con objetos; sino que solo requieren una rudimentaria capacitación en nuevas herramientas. Esto permite una propagación horizontal de las herramientas y una gran oportunidad en el diseño.

UML.

Fue originalmente concebido por la Corporación Rational Software y tres de los más prominentes métodologistas en la industria de la tecnología y sistemas de información: Grady Booch, James Rumbaugh, y Ivar Jacobson ("The Three Amigos"). El lenguaje ha ganado un significante soporte de la industria de varias organizaciones vía el consorcio de socios de UML[2] y ha sido presentado al Object Management Group o Grupo de Administración de Objetos (OMG) y aprobado por éste como un estándar (noviembre 17 de 1997).

UML es:

- Un lenguaje de modelamiento unificado, para la especificación, visualización, construcción y documentación dentro del proceso en el desarrollo de sistemas.
- Un lenguaje para modelar que se enfoca a la comprensión de un tema a través de la formulación de un modelo (y su contexto respectivo).

[2] Unified Modeling Language Documentation. UML Resource Center (1999), http://www.rational.com/uml/resources/documentation.

- La integración de las mejores prácticas de la ingeniería de la industria tecnológica y sistemas de información pasando por todos los tipos de sistemas, dominios (negocios versus software) y los procesos de ciclo de vida.

- Un lenguaje que se aplica para especificar sistemas y usarse para comunicar "qué" se requiere de un sistema, "cómo" un sistema puede ser realizado y describir visualmente a un sistema antes de realizarlo.

- Un lenguaje que puede ser usado para guiar la realización de un sistema similar a los "planos". En cuanto a cómo se aplica para documentar sistemas, puede ser usado para capturar conocimiento respecto a un sistema a lo largo de todo el proceso de su ciclo de vida.

UML no es:

- Un lenguaje de programación visual, sino un lenguaje de modelamiento visual.

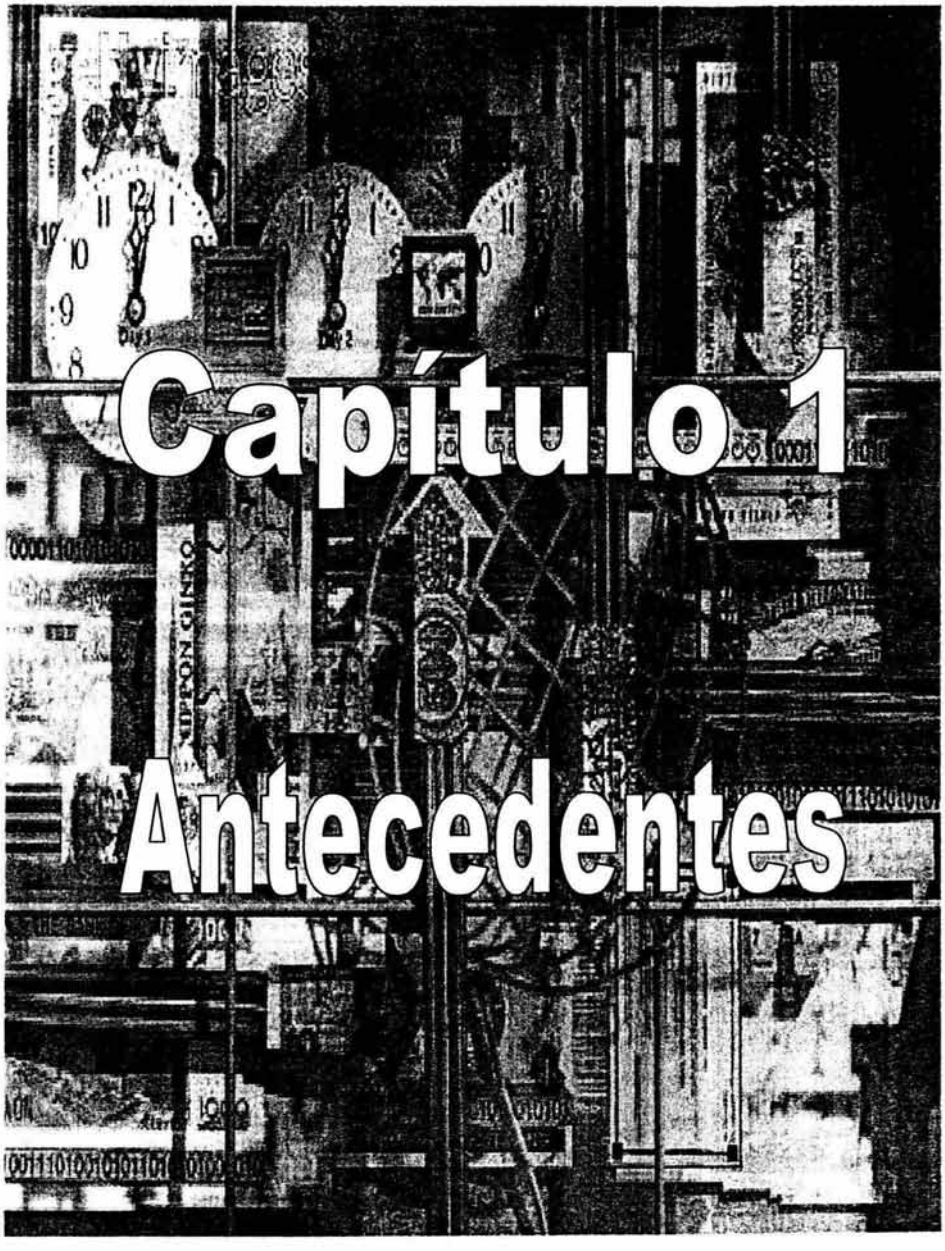
- Una herramienta o un conjunto de especificaciones, sino un lenguaje para modelamiento de especificaciones.

- Un proceso, sino que habilita procesos.

Fundamentalmente, UML está relacionado con la captura, comunicación y nivelación (disgregación en niveles) de conocimientos.

Después de una introducción más bien teórica en la que se ha presentado una declaración de intenciones de la ingeniería del software, en donde se ha planteado los problemas que tiene la ingeniería del software, básicamente por su falta de análisis metódicos y sistemáticos. Hemos visto que el software cuenta con cualidades innatas para introducir métodos que permitan conocer más a fondo todos los parámetros que influyen en la generación del mismo. Por tanto, es probable que el conocimiento del desarrollo de software pueda también dar solución a muchos problemas hasta aquí planteados.

A continuación hablaremos del antecedente y contexto bajo el cual se origino el software. Así como la principal meta del mismo (la calidad) a fin de cumplir con los requerimientos y fines para los que fue construido.



Capítulo 1

Antecedentes

1.1 Introducción.

Muchas veces hemos escuchado el termino *"Software"*, el cual lo relacionamos hacia los programas de computadoras, siendo esto correcto y suficiente para abordar el contexto y la problemática que existe alrededor de él, ya que como hemos visto, existen muchos factores que intervienen en "no tenemos sistema", "el sistema se cayó" o "el error se debe a una pequeña falla del sistema".

Al centrar nuestra atención en el software se entiende que el desarrollo del mismo no se realiza por una sola persona, razón por la que se pensaría que ello lo convertiría en un producto mucho más seguro y sin tendencias a fallas, sin embargo, sí en las fases del desarrollo del software no existe una adecuada coordinación y comunicación, el software se convertirá en un absoluto fracaso con "n mil" errores. Ante esta situación el hombre ha tenido que establecer procedimientos, a fin de darle un orden en la creación del software, mediante metodologías y técnicas que le permitan interpretar fielmente los requisitos exigidos por el usuario, y todo ello se ha concentrado en lo que hoy se le conoce como ingeniería de software, el cual pretende justamente, incrementar esta seguridad durante el proceso de desarrollo hasta alcanzar un nivel de confianza similar al existente en otras ingenierías.

A continuación analizaremos ciertos sucesos ocurridos en nuestra sociedad con la intención de entender el riesgo que se ha sobrellevado en el uso del software, sin un buen grado de calidad.

1.2 El riesgo en la sociedad moderna.

Los accidentes más graves del siglo XX y actuales han ocurrido en los últimos 25 años, como a continuación lo observamos.

- En 1979 ocurrió el accidente de Three Mile Island cerca de Harrisburg (Pensilvania) cuyo reactor estuvo a punto de fundirse, por lo cual no se han encargado nuevas centrales nucleares en Estados Unidos y no se ha permitido el funcionamiento de algunas centrales ya terminadas. En 1990, alrededor del 20% de la energía eléctrica generada en Estados Unidos procedía de centrales nucleares.

- El peor accidente industrial de la historia ocurrió en Bhopal, India en 1984, cuando se produjo un escape de "isocianato de metilo" en la planta química de Union Carbide, provocando la muerte de 3300 personas e hirió a más de 200,000. La India fue el primer país en vías de desarrollo que llegó a ser potencia nuclear.

- El 26 de abril de 1986 ocurre el accidente de Chernóbil en el que se fundió parcialmente un reactor de la central de Chernóbil, situada al norte de Kíev (Ucrania). Este último accidente provocó numerosas muertes y casos de enfermedad ocasionados por la radiación.

- Tres meses antes de Chernóbil, el 28 de enero de 1986 el transbordador espacial Challenger explotó, mientras el mundo contempló con horror cómo el transbordador espacial estadounidense explotó 73 segundos después de despegar (figura 1.1).

Figura 1.1.- Explosión del Challenger (28 enero 1986).

- Muy recientemente el 1 de Febrero del 2003, el transbordador espacial Columbia, con siete astronautas a bordo, se desintegró sobre Texas poco antes de aterrizar en Cabo Cañaveral (figura 1.2).

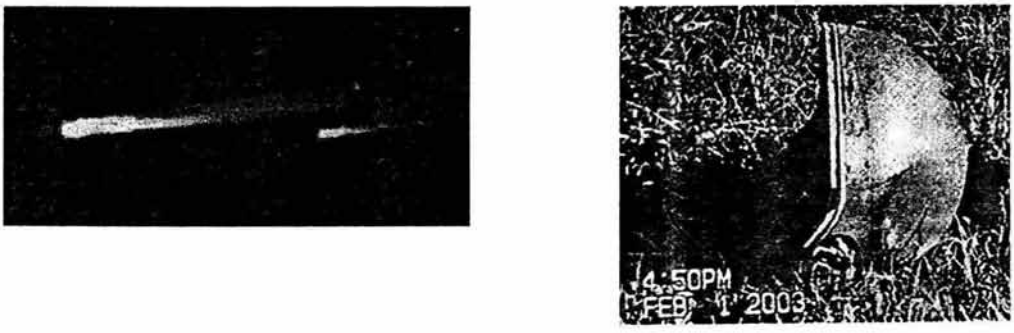


Figura 1.2.- Explosión del Columbia (1 de Febrero 2003).

¿Está convirtiendo la tecnología nuestro mundo en uno más arriesgado?

El riesgo no es un problema nuevo. Siempre ha existido en el medio ambiente. Pero hoy en día la industrialización está reemplazando los peligros de la naturaleza en nuevos riesgos que se basan en la transformación o afectación de la naturaleza mediante la tecnología.

Sin embargo, hemos utilizado tanto a la tecnología, que prácticamente ya es parte de nuestra vida, a tal grado que los problemas que surgen nos es muy difícil identificar si fueron originados por su uso o por problemas de la naturaleza.

A resumidas cuentas, la modificación del medio ambiente realizada por los seres humanos ha agravado los desastres naturales.

1.3 Cambios en la actitud hacia el riesgo.

Todas las actividades de los seres humanos involucran el riesgo. El progreso demanda arriesgarse, pero ahora la sociedad demanda que se sepan y controlen los riesgos hasta la extensión máxima posible. El cambio de la responsabilidad desde algo personal a algo público o de la organización es un fenómeno reciente. Hasta la primera parte de este siglo los trabajadores contaban con sus propias herramientas, entendían su oficio y asumían la responsabilidad para su seguridad. Ahora los trabajadores no tienen mucho control sobre la manera en que trabajan y por lo tanto dependen más de sus empleadores para proveer un ambiente seguro.

La complejidad de la sociedad tecnológica requiere que el público confíe en el conocimiento de los expertos. La responsabilidad se ha transferido del público al estado, la gerencia corporativa, los expertos de seguridad, y otros profesionales. A la vez, los accidentes como Bhopal han creados nuevas demandas en el público. La gente quiere saber los riesgos de las tecnologías peligrosas, los escenarios del peor caso, los peligros de contaminación, etc.

Ante estos hechos vemos que la reducción del riesgo puede ser caro, sin embargo, es necesario identificar los principales.

1.4 Factores de riesgo en la sociedad industrializada.

1.4.1. Nuevos Peligros.

Antes de la Revolución Industrial los accidentes eran los resultados de causas naturales o involucraban pocos elementos tecnológicos sencillos. Los avances de la tecnología han creados muchos beneficios pero también peligros nuevos. Ahora tenemos microbios resistentes, productos químicos en la comida, los peligros del átomo, entre otros, que son difíciles de encontrar.

1.4.2. Más complejidad.

Muchos peligros son relacionados a la complejidad de los nuevos sistemas. La complejidad también dificulta la identificación de las causas de los accidentes. Los sistemas consisten en

redes de subsistemas con acoplamiento estrecho. Las condiciones que producen peligros surgen en las interfaces y los problemas se propagan de un componente a otro. En algunos sistemas la operación puede ser entendida solamente por pocos expertos.

1.4.3. Más energía.

El uso de fuentes de energía alta, tal como sistemas de alta presión o la fisión atómica, ha aumentado la magnitud de las pérdidas potenciales. Otros sistemas usan energía convencional pero en cantidades mayores.

1.4.4. Más automatización de operaciones manuales.

Aunque puede parecer que la automatización disminuiría el riesgo de errores de operación, la verdad es que no elimina a la gente que opera a los sistemas. Los operadores en los sistemas automatizados están a menudo en salas de control central, donde tienen que usar información indirecta sobre el estado del sistema. Esta información puede ser errónea, como en el apagón de Nueva York en 1977 (la falla de un relé oscureció la falla de una segunda, y la línea parecía normal).

1.4.5. Más centralización y mayor escala.

La automatización ha sido acompañada por la centralización de procesos en plantas muy grandes. Se han extrapolado dispositivos y procesos a áreas no probadas. La fuerza nuclear ha sufrido de este problema.

1.4.6. El paso más rápido del cambio tecnológico.

El tiempo promedio para desarrollar un descubrimiento básico en un producto ha disminuido, llegando a ser de 30 años a principios de siglo hasta en cinco años en la actualidad. El número de productos y procesos nuevos está creciendo en una manera exponencial.

Hay menos posibilidades de aprender de la experiencia porque el paso de cambio es demasiado rápido y las consecuencias de fallas son demasiado graves. Los diseños y procedimientos de operación tienen que ser correctos la primera vez. Como resultado se reemplazan las reglas de diseño empíricas por depender de la identificación de peligros y del control. No es conocido cuánta protección ofrece estos métodos por contraste con los métodos antiguos de experiencia y estándares.

1.5 ¿Cuán seguro es suficientemente seguro?.

Parece que el riesgo está creciendo. El enfoque de seguridad del sistema trata de reducir el riesgo anticipando los accidentes y sus causas en análisis del peligro. Pero no es posible eliminar el riesgo completamente. Hay un problema básico que últimamente se encuentra sobre la mesa, en el cual existen mutuas complicidades entre la seguridad y el desempeño. Dada la existencia del riesgo hay que decidir qué sistemas se deben construir y qué tecnología se les debe introducir.

Una posibilidad es el análisis de riesgos y beneficios. La idea es medir el riesgo y elegir un nivel apropiado para hacer decisiones. El problema es que es imposible medir el riesgo con precisión, especialmente antes de construir un sistema. En la valoración del riesgo se tratan de cuantificar antes de que los datos estén disponibles, sin embargo, se calcula la probabilidad de accidentes graves construyendo modelos de interacción de eventos que pueden producir un accidente. En la práctica se incluyen solamente los eventos que se pueden medir. A la vez, los factores causales involucrados en la mayoría de los accidentes mayores no son medibles.

Un segundo problema es determinar el nivel aceptable de riesgo. ¿Quién determina qué nivel de riesgo es aceptable en comparación con los beneficios?.

"FORD" una empresa automotriz de gran prestigio, ha tenido graves problemas en algunos de sus modelos, como en el caso de los tanques de bencina de los "Ford Pinto" o el sistema de frenos sobre el "Ford Focus". Ford sabía el peligro existente, sin embargo, después de un análisis de riesgo mostró que sería más barato arreglar las disputas solicitando a los conductores que llevasen las unidades a las agencias a que los conductores asumieran el riesgo y tener que indemnizar posteriormente a éstos.

1.6 Las computadoras y el riesgo.

Hoy en día hay pocos sistemas que no usan computadoras para proveer control o apoyar el diseño. Las computadoras ahora controlan muchos dispositivos críticos de la seguridad y frecuentemente reemplazan los medios físicos de seguridad.

- En 1979 en los Estados Unidos el descubrimiento de un error en el software usado en el diseño de reactores nucleares y sus sistemas refrescantes resultó en que la Comisión de Regulación Nuclear cerró cinco plantas que no satisfacían los estándares para los terremotos.

- Frecuentemente se usa el argumento que el software genera datos pero no hace decisiones, lo cual es crítico a la seguridad. Más frecuentemente el operador tiene que confiar en datos para los cuales no tiene ninguna manera de verificarlos en forma independiente.

1.7 Mitos de software.

A continuación se observan algunos de los mitos más frecuentes en el software:

- **El costo de computadoras es menos que el de los dispositivos analógicos o electromecánicos.**

La verdad es que el hardware es barato, pero el costo de escribir y certificar software muy seguro y confiable más el costo del mantenimiento sin poner en peligro la confiabilidad y la seguridad, puede ser enorme.

Por ejemplo, el software del transbordador espacial (400,000 palabras de código) cuesta más de $10,000,000 de dólares por año. Un sistema electromecánico es frecuentemente más barato, particularmente si se pueden usar diseños estándares.

- **Es fácil cambiar el software.**

Los cambios son fáciles, pero hacer cambios sin introducir errores es muy difícil. Hay que verificar el software de nuevo con cada cambio. También, con cambios el software se pone frágil.

- **Las computadoras proveen más confiabilidad que los dispositivos que reemplazan.**

Es verdad que el software no falla como dispositivos normales, pero hay poca evidencia que indica que el comportamiento erróneo de software no es un problema significativo.

Estudios de sistemas muy críticos en el entorno de la seguridad han mostrado que hasta 10% de los módulos se desviaron de la especificación en un modo o más de operación. Muchos errores eran pequeños, pero aproximadamente 1 en 20 producía efectos directos y observables en el sistema controlado.

Parece que la solución es sencillamente implementar el software correctamente. Pero esto es mucho más difícil de lo esperado. Por ejemplo, el software del transbordador espacial ha sido usado desde 1980 y la NASA ha invertido recursos enormes en la verificación y

mantenimiento de este software. Sin embargo, desde la operación del transbordador se han encontrado 16 errores de grado de severidad 1 (la severidad 1 pueden producir una pérdida del transbordador y su tripulación) en el software liberado, siendo que 8 de estos errores se encontraban en el código que normalmente ya había sido utilizado en forma "confiable".

Otros errores de menor severidad han ocurrido durante misiones (tres amenazaron el cumplimiento de la misión) a pesar que la NASA tiene uno de los procesos de desarrollo y verificación de software más completos existentes.

No existen técnicas coma la redundancia para aumentar sencillamente la confiabilidad del software. Y aun cuando sea posible escribir software sin errores, las condiciones ideales para desarrollar software (dinero y tiempo sin límites) nunca existen.

- **La prueba o verificación formal del software puede eliminar todos los errores.**

Las limitaciones de la prueba de software son bien conocidas. Básicamente hay demasiados estados en el software real para probarlo completamente.

En el futuro la verificación puede validar la consistencia entre las especificaciones y la implementación, pero esto requiere que las especificaciones (escritas en notación formal) se encuentren libres de errores. También, muchos errores importantes son debidos a cosas que no están en el código. Por ejemplo, muchos accidentes relacionados a software han involucrado sobrecarga. Un sistema para los servicios de emergencia dejó de funcionar cuando recibió demasiadas llamadas.

- **El reuso de software aumenta la seguridad.**

Aunque el reuso puede aumentar la confiabilidad, puede disminuir la seguridad. La razón es porque engendra el falso sentido de seguridad y porque no se consideraron los peligros específicos del sistema cuando éste fue diseñado originalmente bajo otras especificaciones. Ejemplos:

  - El software de control del tráfico aéreo usado por muchos años en los Estados Unidos no se pudo reutilizar en Gran Bretaña. Los desarrolladores norteamericanos han ignorado el problema de cero grados de longitud.

  - El software de aviación escrito para uso en el hemisferio boreal frecuentemente crea problemas cuando es usado en el hemisferio austral. También el software

para cazas F-16 ha causado accidentes cuando éste se uso en Israel en aviones volados sobre el Mar Muerto, donde la altitud es menor que el nivel del mar.

La seguridad no es una propiedad del software, sino es una combinación del diseño del software y del ambiente en que es usado.

- **Las computadoras disminuyen el riesgo por contraste con los sistemas mecánicos.**

Las computadoras tienen este potencial y pueden automatizar tareas peligrosas como el pintar con spray, construcción de armamento o artefactos militares, pruebas con bacterias de peligro letal, entre otros, sin embargo, existen ciertos argumentos que son discutibles, ya sea a favor o en contra:

  - Las computadoras permiten un control más fino ya que pueden revisar parámetros más frecuentemente, hacer cálculos en tiempo real, y tomar acción rápidamente.

  Es verdad. Pero el control más fino permite que el proceso se pueda operar más cerca a lo óptimo, y se pueden reducir los margines de seguridad. Los sistemas que resultan tienen beneficios económicos, porque en teoría se cerrarán menos y la productividad se puede aumentar con control más óptimo.

  - Los sistemas automatizados permiten a los operadores trabajar más lejos de áreas peligrosas.

  Con la eliminación de operadores se eliminan los errores humanos. Los errores de operadores se reemplazan con errores de diseño y mantenimiento; los diseñadores hacen los mismos tipos de errores como los operadores. También, cuando se elimina el contacto directo con el sistema, los humanos pierden la información necesaria para hacer decisiones.

  - Las computadoras pueden proveer mejor información a los operadores.

  En teoría es verdad, pero esto es muy difícil lograr. Básicamente las computadoras permiten que se provee demasiada información y en una forma menos útil para algunos propósitos que la instrumentación tradicional.

1.8 Una perspectiva histórica.

El término Ingeniería del Software (ampliamente utilizado hoy día, hemos preferido el de ingeniería de software por recalcar el aspecto de software) fue acuñado en 1968 en el transcurso de un curso de verano de la OTAN en Garmisch, Alemania.

Si bien es verdad que la Ingeniería del software no ha solventado todas las deficiencias que se desean, ha evolucionado de gran forma, por lo que entraremos a una perspectiva historia, tomada desde tres puntos de vista:

- **Programación.** En éste se vislumbran cambios directos en la metodología y técnicas de creación del software.

- **Industrial y social.** Aquí se observan los cambios en el entorno de software, como lo son los equipos de cómputo y los cambios sociales, que contrajo esta evolución.

- **El software en México.** En México la evolución del software respecto al resto del mundo no fue mucho tiempo después, son solamente tres años los que nos separan del despegue del resto de los demás y para orgullo propio se da en la UNAM.

1.8.1. Evolución de la programación.

Centrándonos en la ingeniería de software, su consolidación ha sufrido una evolución en etapas en paralelo con la propia evolución de la programación. En este sentido se han destacado cuatro etapas[3]:

1. **La programación como base del desarrollo (1955-1965).** Énfasis absoluto en la tarea de escribir el código en un lenguaje de programación. Alrededor de los nuevos lenguajes de alto nivel, los programadores se alejan de la estructura de los ordenadores y comienzan a acercarse a la complejidad de las aplicaciones de usuario.

2. **La génesis (1965-1975).** Ligada a la crisis del software se plantea la necesidad de controlar el proceso de desarrollo. Se definen modelos de ciclo de vida como una referencia en la que enmarca las actividades requeridas. El concepto de ciclo de vida en cascada surge de la necesidad del Departamento de Defensa de EE.UU. de disponer de una documentación normalizada para todas las etapas del desarrollo y poder controlar en base a ella a los suministradores de productos software.

[3] Ingeniería de sistemas de software.- Dr. Gonzalo León Serrano, 1996

3. **La consolidación (1975-1985).** El control de las actividades de desarrollo debería permitir gestionar el proceso. Durante esta etapa aparecen métricas para estimar a priori el coste o el tamaño del software; se difunde el uso de métodos de desarrollo. Con ello, el programador se convierte en analista, diseñador o gestor. Se vislumbra la idea del Ingeniero del software.

4. **Hacia una ingeniería (1985-1995).** Aceptando una consolidación de las tecnologías de software, la mejora viene de la mano de un mejor conocimiento de los procesos con el fin de incrementar la calidad de los productos. Aparece una gestión sofisticada del proceso de desarrollo ligada al control de riesgos y a la madurez de los procesos.

1.8.2. Evolución industrial y social.

Glass[4] divide este periodo en tres etapas:

1. **La era Pionera (1955-1965).** El desarrollo más importante fue que las nuevas computadoras surgían casi cada año o dos. El personal del Software tuvo que reescribir todos sus programas para correr estos sobre los nuevos equipos, adicionalmente los programadores no contaban con computadoras sobre sus escritorios y tenían que ir al "cuarto de máquina" (laboratorio).

   Este campo era tan nuevo que la idea de administrar en forma programada no existía. Hacer predicciones sobre la fechas de termino del proyecto era casi imposible.

   Las computadoras realizaban tareas específicas. Tareas científicas y financieras necesitaban diferentes máquinas.

   Debido a la necesidad para trasladar el viejo Software a las características de nuevas máquinas, surgen lenguajes de alto nivel como: FORTRAN, COBOL, y ALGOL.

   Vendedores de Hardware obsequiaban los sistemas (software), dado que los equipos no podían ser vendidos sin éste. Pocas compañías vendían servicios para construir software que fuese a la medida del cliente, más no existían compañías que vendieran software empacado.

[4] En el comienzo: Recolección de los Pioneros de Software.- Robert L. Glass, 2001

La noción del reuso floreció debido a que el software era gratuito y los usuarios de organizaciones comúnmente lo obsequiaban. De igual manera grupos de científicos de IBM compartían componentes reutilizables mediante catálogos de componentes.

Las universidades aun no enseñaban los principios de la computación y la programación modular, así como la abstracción de datos ya empezaban a ser usadas en la programación.

2. **La Era de la Estabilización (1965-1980).** El conjunto de trabajos habían sido institucionalizados y algunos programadores perdieron sus trabajos, excepto por algunas aplicaciones. La burocracia crecía en torno de la computadora central.

   El mayor problema resultado de esta burocracia era el tiempo de espera, el tiempo entre la captura de la información, procesamiento y los resultados, lo que llego a medirse en días.

   Entonces llega el IBM 360 (7 Abril 1964), el cual es señalado como el principio de la era de la estabilización. Esto puso fin a la era de un rápido y barato surgimiento de computadoras cada año, a pesar de que el tamaño de estos equipos no era pequeño, como se observa en la figura 1.3.

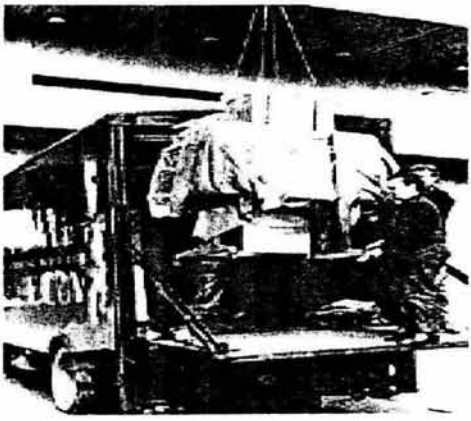


Figura 1.3.- Impresora de alta velocidad para la IBM 360.

La gente del Software pudo finalmente utilizar el tiempo escribiendo nuevo software en lugar de rescribir el anterior. La familia IBM 360, 370 (1970) también combino aplicaciones científicas y financieras sobre una máquina. Esto ofrecía ambas

aritméticas: binaria y decimal. Con la ventaja de que la distancia organizacional entre las personas que usaban aplicaciones científicas y financieras disminuyó, lo que creo un impacto, entre estas áreas, ya que los programadores científicos quienes normalmente tenía títulos universitarios se sentían superiores a los programadores financieros quienes por lo regular tenían títulos de bachillerato según el sistema americano.

La gran variedad de los sistemas operativos, continuaban siendo gratuitos, en la compra de las computadoras, sin embargo la mayoría de los programas no tenían un control sobre los recursos de las máquinas, razón por la cual podía una aplicación consumir todos los recursos de una máquina.

El lenguaje de control de trabajos (JCL) trajo consigo una nueva serie de problemas. El programador tenia que escribir el programa en un nuevo lenguaje para decirle a la computadora y al sistema operativo que hacer. El JCL fue la característica más popular de las 360.

El compilador PL/1, introducido por IBM (Marzo 1968) para unir a todos los lenguajes de programación en uno.

La demanda de programadores excedía la existencia de ellos.

Como el campo del software se estaba estabilizado, el software llegó a incorporar un valor enorme.

Firmemente emergió en las universidades las disciplinas computacionales en los finales de los 60's. Sin embargo la carrera de ingeniería de software aun no existía.

Muchas carreras "exageradas" como la Inteligencia Artificial se originan en base al pensamiento del ser humano, su razón de ser y existir, sin embargo, como estos nuevos conceptos aún no podían ser aterrizados en beneficios reales, la credibilidad del campo de la computación empezó a disminuir.

La "Programación Estructurada " estalló sobre la escena de la mitad de ésta era.

Los vendedores de equipos quienes definían el estándar podían nuevamente competir con la ventaja de hacer los estándares más parecidos a sus propias tecnologías.

Aunque los vendedores de hardware trataban de hacer una pausa sobre la industria del software por mantener sus precios bajos, los vendedores de software surgen en poco tiempo.

La mayoría de las aplicaciones continuaban siendo hechas en casa.

3. **La Micro Era (1980-Hasta la fecha).** Los precios de las computadoras han caído dramáticamente haciéndolas casi accesibles para todos. Ahora todos los programadores pueden tener una computador en su escritorio.

   El viejo JCL ha sido reemplazado por la amigable Interfaz Grafica de Usuario (GUI).

   La mayoría de programadores utilizan los lenguajes de programación y la idea esta más limitada a la generación de reportes desde las Bases de Datos.

   Hay un clamor por la búsqueda de más y mejor software, pues el actual esta entre 15 y 40 años atrasado.

   Los lenguajes de cuarta generación nunca lograron el sueño de "Sin Programación"

   A lo largo de estas etapas, han existido avances puntuales significativos tanto en la tecnología empleada como en la propia percepción del proceso de desarrollo.

   El progreso hacia la ingeniería de software ha sido acumulativo en los años cubiertos por las etapas mencionadas y no se puede entender ningún progreso sin la experiencia obtenida de éxitos y fracasos anteriores.

1.8.3. El Software en México.

En México, la historia del cómputo empezó apenas en 1958, al llegar a la UNAM una IBM 650, siendo la primera computadora de América Latina, como se muestra en la figura 1.4.

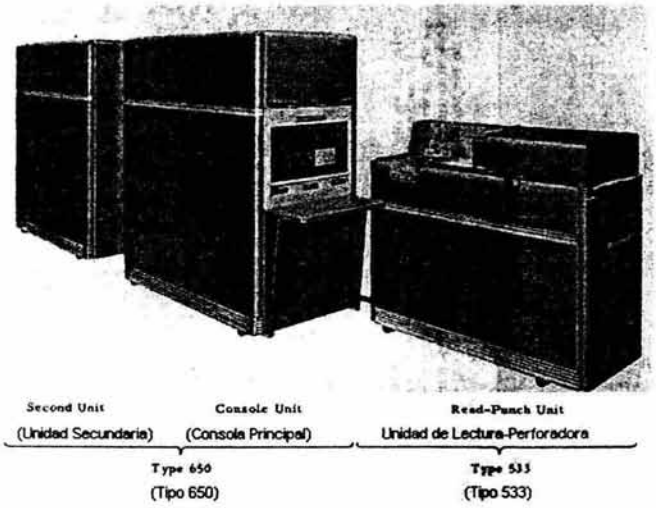


Figura 1.4.- IBM 650, la cual fue liberada en Diciembre de 1954.

Si analizados globalmente los puntos significativos en el desarrollo del software, observamos que se han ido intercalando el énfasis sobre las tecnologías de desarrollo con el énfasis en la gestión del proceso. En cada una de estas etapas se consiguió un incremento de la calidad del proceso de desarrollo.

Si inicialmente la esperanza de una mejora de calidad del producto se centraba en el empleo de nuevos lenguajes de programación, después se hizo necesario una mejor comprensión del ciclo de vida. Posteriormente, fue necesario robustecer ese ciclo de vida en cascada con tecnologías de software que facilitasen las primeras fases del ciclo de vida. Pero el empleo de métodos y herramientas software de ayuda no hacía más predecible y eficiente el desarrollo de un gran software: el énfasis se situó de nuevo sobre los aspectos de gestión con la mejora del proceso.

1.9 Por qué la ingeniería de software es difícil.

Hay varias razones por qué hay tantos problemas en la ingeniería de software.

- **La flexibilidad.** El costo bajo aparente de cambiar software es engañoso. También, la flexibilidad refuerza la tardanza en la redefinición de tareas bajo el proceso de desarrollo del sistema. Los cambios son mucho más difíciles para las máquinas físicas porque las propiedades de los materiales restringen las modificaciones. Tales frenos no existen para el software, pero es tan frágil como hardware.

- **La planificación y la estimación de costos son muy imprecisas.** A la hora de abordar un proyecto de una cierta complejidad, cualquiera que sea el ámbito, es frecuente que surjan imprevistos que no estaban en la planificación inicial, y como consecuencia de estos imprevistos se producirá una desviación en los costos del proyecto. Sin embargo, en el desarrollo de software lo más frecuente es que la planificación sea prácticamente inexistente, y que nunca se revise durante el desarrollo del proyecto. Sin una planificación detallada, es totalmente imposible hacer una estimación de costos que tenga alguna posibilidad de cumplirse, y tampoco se pueden identificar las tareas conflictivas que pueden desviarnos de los costos previstos.

- **La productividad es baja.** Los proyectos software tienen, por lo general, una duración mucho mayor a la esperada. Como consecuencia de esto los costos se disparan y la productividad y los beneficios disminuyen. Uno de los factores que influyen en esto, es la falta de unos propósitos claros o realistas a la hora de comenzar el proyecto. La mayoría del software se desarrolla a partir de especificaciones ambiguas o incorrectas, y no existe apenas comunicación con el cliente hasta la entrega del producto. Debido a esto son muy frecuentes las modificaciones de las especificaciones sobre la marcha o los cambios de última hora. No se realiza un estudio detallado del impacto de estos cambios y la complejidad interna de las aplicaciones crece hasta que se hacen virtualmente imposibles de mantener y cada nueva modificación, por pequeña que sea, es más costosa, y puede provocar el fallo de todo el sistema.

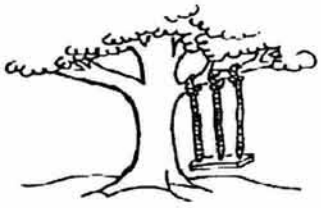
- **La calidad es mala.** Como consecuencia de que las especificaciones son ambiguas o incluso incorrectas, y de que no se realizan pruebas exhaustivas, el software contiene numerosos errores cuando se entrega al cliente. Estos errores producen un fuerte incremento de costos durante el mantenimiento del producto, cuando ya se esperaba que el proyecto estuviese acabado. Sólo recientemente se ha empezado a tener en cuenta la importancia de la prueba sistemática y completa, y han empezado a surgir conceptos como la fiabilidad y la garantía de calidad.

- **La complejidad y las interfaces invisibles.** Frecuentemente se trata de manejar la complejidad partiendo el objeto en trozos o módulos. En el ámbito del software el resultado de este proceso son más interfaces, y estas introducen una nueva fuente de la complejidad.

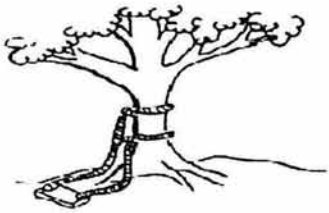
- **No hay una clara idea de lo que el cliente desea.** Debido al poco tiempo e interés que se dedican al análisis de requisitos y a la especificación del proyecto, a la falta de comunicación durante el desarrollo y a la existencia de numerosos errores en el producto que se entrega, los clientes suelen quedar muy poco satisfechos de los resultados. Consecuencia de esto, es que las aplicaciones tengan que ser diseñadas y desarrolladas de nuevo, que nunca lleguen a utilizarse o que se produzca con frecuencia un cambio de proveedor a la hora de abordar un nuevo proyecto.

A continuación observaremos lo que ocurre cuando "No hay una clara idea de lo que el cliente desea" (figura 1.5).

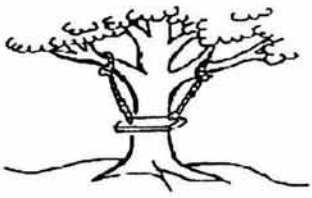
Como el usuario lo ordene



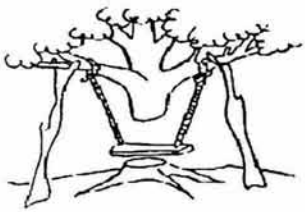
Como su jefe lo interpretó



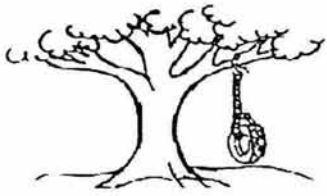
Como el analista lo interpretó



Como el programador lo diseño



Como se puso en operación



Como el usuario quería

Figura 1.5.- Insatisfacción del cliente.

- **Proveedores incumplidos.** Es común hallar a proveedores de software con floreados discursos sobre buenas prácticas de gestión de proyectos pero que a la hora de implementar, caen en saco roto. Los gestores de los proyectos no están especializados en la producción de software. Tradicionalmente, los responsables del desarrollo del software han sido ejecutivos de nivel medio y alto sin conocimientos de informática, siguiendo el principio de 'Un buen gestor puede gestionar cualquier proyecto'. Esto es cierto, pero no cabe duda que también es necesario conocer las características específicas del software, aprender las técnicas que se aplican en su desarrollo y conocer una tecnología que evoluciona continuamente.

Ahora bien, desde el punto de vista del diseñador y operador humano se conjugan, dos propiedades que actúan durante toda la vida útil del software: la incertidumbre de lo que realmente hará y la ignorancia en cómo lo conseguiría. Reducirlas y dominarlas ha constituido la meta básica en la evolución de la Ingeniería Software durante el último cuarto de siglo.

En el siguiente capítulo veremos la manera mediante el cual la Ingeniería del Software ha conseguido reducir los riesgos aquí mencionados.


Capítulo 2
Ingeniería
de Software

2.1 Introducción.

Hemos visto que establecer una metodología que permita la creación del software con calidad, es la meta de la ingeniería de software, sin embargo, a fin de tener un punto de referencia a continuación se verán sus definiciones tal como aparecen en el diccionario[5] y que a continuación se enlistan:

- **Ingeniería.** Término aplicado a la profesión en la que se aplican los conocimientos, la experiencia, la práctica y estudio de las matemáticas, la física principalmente alcanzado con ello la utilización eficaz de los materiales y las fuerzas de la naturaleza.

- **Software.** Comúnmente conocido como sistema, conjunto de instrucciones que permiten interactuar con las máquinas (PC), el software puede dividirse en dos categorías: Los sistemas operativos y el software de aplicación. El software del sistema operativo procesa tareas tan esenciales, que parecerían invisibles, como el acceso a la información dentro del disco duro, la administración de periféricos, etc., mientras que el software de aplicación lleva a cabo tareas de procesamiento de textos, gestión de bases de datos, entretenimiento, en fin una serie de tareas especificas dependiendo al área en la cual se apliquen estos.

- **Ingeniería de Software.** Disciplina que ofrece métodos y técnicas para desarrollar y mantener la calidad de software.

Ahora analizaremos la definición de dos personajes que han sido pilares dentro de la ingeniería de software:

Blanchard[6] define al software como "una combinación de recursos (seres humanos, materiales, equipos, instalaciones, datos, etc.) integrados de forma tal que cumplen una función específica en respuesta a una necesidad designada por el usuario". Esta es una definición genérica que incluye todo tipo de software.

Ahora veremos una definición clásica dada por Pressman[7] en donde enfatiza aspectos concretos de calidad, enunciando a la ingeniería del software como: "El establecimiento y uso de principios robustos de la ingeniería, orientados a obtener software económico que sea confiable y funcione de manera eficiente sobre máquinas reales".

Esta definición enfatiza los principios y metodologías, encaminados a obtener un producto con calidad. Sabiendo que los principios se refieren a aceptar leyes y reglas de acuerdo a

[5] Diccionario de la Real Academia Española, http://www.rae.es/
[6] Blanchard, B.S., Ingeniería de Sistemas, Serie de monografías de ingeniería de sistemas, Isdefe, Madrid, 1995.
[7] Pressman, R.S., Ingeniería del Software: un enfoque práctico, 5ª edición, Mc Graw-Hill, 2002.

la forma de trabajar bajo cierta área, mientras que la metodología trata con el "cómo trabajar" de acuerdo a un plan.

2.2 Clasificación del Software.

En este sentido entendemos que la complejidad del software, se debe a que éste trata de hacer una extracción de algún proceso del mundo real, procurando hacer una copia fiel del mismo. En éste sentido podríamos clasificar el software por su utilidad, distinguiendo tres tipos de software:

- **Software Base.** Es todo aquel que proporciona una plataforma de ejecución (como un sistema operativo o una interfaz de usuario o un compilador) para que otros programas puedan desarrollarse o ejecutarse. Su utilidad no está ligada directamente a una aplicación definida por un usuario sino a otro programa. Suelen formar parte de la infraestructura de ejecución. Un ejemplo típico es el sistema operativo.

- **Software de aplicación.** Responde a una necesidad de un usuario en un determinado contexto que se resuelve mediante un paquete de aplicación. El usuario interacciona directamente con el software de aplicación del que obtiene el servicio deseado. Se distingue entre software de aplicación horizontal cuando responde a necesidades genéricas manifestadas por multitud de usuarios (por ejemplo, un procesador de textos) y software de aplicación vertical cuando la necesidad responde a un tipo de usuario concreto o a un mercado sectorial restringido (por ejemplo, un sistema de gestión de hospitales).

- **Software empotrado.** Hasta ahora hemos considerado el software ejecutándose potencialmente sobre una máquina genérica, sin embargo, en muchos casos no es posible aislar el sistema de la máquina sobre la que se ejecuta. De hecho, no hay interacción externa sino que se realiza sobre un hardware en el que está "empotrado" o "embebido". Un ejemplo de software empotrado lo tenemos en lo hoy en día llamamos "cajeros" o en los "mostradores automáticos" en los aeropuertos establecido recientemente por Aeromexico, otro ejemplo son los despachadores de tickets (figura 2.1).

Figura 2.1.- Cajero (ATM).

La generación de modelos de ingeniería de software consta de 4 fases las cuales se mencionan a continuación

2.3 Fases Genéricas del modelo de la Ingeniería de Software.

La generación de modelos en la ingeniería de software se ha creado a fin de representar la funcionalidad, el comportamiento de lo que se desea realice un sistema, y en este sentido lograr entender lo que el cliente desea, sin embargo, sea cual sea el área en la cual se enfoque la modelación. Se identifican cuatro fases en las que podemos dividir la creación de un software:

Fase de definición (¿qué hacer?)

- Estudia la **factibilidad** o **viabilidad** de realizar el sistema.
- **Conoce los requisitos** que debe satisfacer el sistema (funciones y limitaciones de contexto).
- Asegura que los **requisitos son alcanzables.**
- Formaliza el **acuerdo** con los usuarios.
- Realiza una **planificación** detallada.

Fase de diseño (¿cómo hacerlo? Soluciones en coste, tiempo y calidad)

- Identifica **soluciones tecnológicas** para cada una de las funciones del sistema.
- Asigna **recursos** materiales para cada una de las funciones.
- Propone (identificar y seleccionar) **subcontratas.**
- Establece métodos de **validación** del diseño.
- **Ajusta las especificaciones** del producto.

Fase de construcción

- Genera el producto o servicio pretendido con el proyecto.
- Integra los elementos subcontratados o adquiridos externamente.
- Valida que el producto obtenido satisface los requisitos de diseño previamente definidos y realizar, si es necesario, los ajustes necesarios en dicho diseño para corregir posibles lagunas, errores o inconsistencias.

Fase de mantenimiento y operación

- **Operación.** asegurar que el uso del proyecto es el pretendido.
- **Mantenimiento.** Se basa en el cambio que va asociado a la corrección de errores, a las adaptaciones en base a la evolución del entorno del software y a cambios o mejoras solicitados por el cliente. Durante esta fase se encuentran cuatro tipos de mantenimiento:
- **Correctivo.** El mantenimiento correctivo cambia el software para corregir los defectos.
    - Adaptativo. Este mantenimiento se produce a fin de acoplar el software a los cambios de su entorno externo.
    - Perfectivo o de Mejora. En este sentido el mantenimiento lleva al software más allá de sus requisitos funcionales originales. Ya que conforme se utilice el software, el cliente puede descubrir funciones adicionales que van a producir beneficios.
    - Preventivo. En esencia, este mantenimiento hace cambios en los programas a fin de que se puedan corregir, adaptar y mejorar más fácilmente, ante los cambios que se vislumbran (Ejemplo: Y2K).

2.4 Modelos de Ingeniería de Software.

Todo proyecto de ingeniería tiene fines ligados a la obtención de un producto que es necesario generar a través de diversas actividades. Algunas de estas actividades pueden agruparse en fases porque globalmente contribuyen a obtener un producto intermedio necesario para continuar hacia el producto final y facilitar la gestión del proyecto. De esta manera se han creado modelos que representen el ciclo de vida del software.

Un modelo de ciclo de vida de software es una vista de las actividades que ocurren durante el desarrollo de un sistema, intentando determinar el orden de las etapas involucradas y los criterios de transición asociadas entre estas etapas. Cada fase o etapa de desarrollo tiene un conjunto de metas bien definidas y las actividades dentro de una fase contribuye a la satisfacción de metas de esa fase. Las flechas muestran el flujo de información entre las fases. La flecha de avance muestra el flujo normal. Las flechas hacia atrás representan la retroalimentación.

La modelación del ciclo de vida de software brinda una guía para los ingenieros de software con el fin de ordenar las diversas actividades técnicas en el proyecto, por otra parte suministran un marco para la administración del desarrollo y el mantenimiento, en el sentido en que permiten estimar recursos, definir puntos de control intermedios, monitorear el avance, etc.

A lo largo de los años se han creado diversos modelos de entre los cuales destacan:

2.4.1 Modelo Lineal Secuencial o Cascada.

Este es el modelo básico y sirve como base para los demás modelos de ciclo de vida. Es el más utilizado, precisamente por ser el más sencillo. Consiste en descomponer la actividad global del proyecto en fases que se suceden de manera lineal, es decir, cada una se realiza una sola vez, cada una se realiza tras la anterior y antes que la siguiente. Con un ciclo lineal es fácil dividir las tareas entre equipos sucesivos y prever los tiempos (sumando los de cada fase).

Este modelo requiere que la actividad del proyecto pueda descomponerse de manera que una fase no necesite resultados de las siguientes (realimentación), aunque pueden admitirse ciertos supuestos de realimentación correctiva.

Desde el punto de vista de la gestión (para decisiones de planificación), requiere también que se sepa bien de antemano lo que va a ocurrir en cada fase antes de empezarla (figura 2.2).

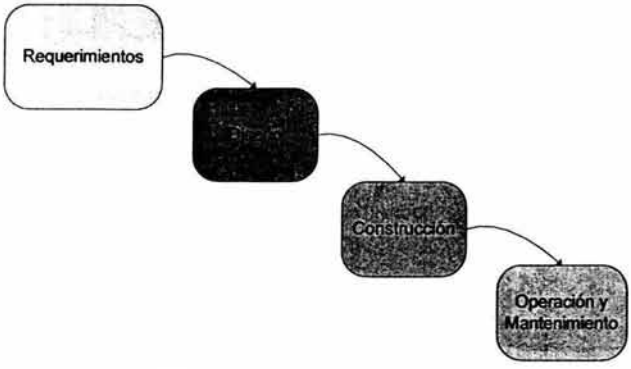


Figura 2.2.- Modelo Lineal Secuencial.

El modelo comprende las siguientes actividades:

- **Análisis de los requerimientos del software.** El proceso de reunión de requerimientos se intensifica y se centra especialmente en el software. Para comprender la naturaleza del programa a construirse, el "analista" debe comprender la información que utilizara el sistema, así como la función requerida, el rendimiento e interconexión.

- **Diseño.** El proceso del diseño traduce requerimientos en una representación del software donde se pueda evaluar su calidad antes de que comience la codificación.

- **Construcción.** El diseño se debe traducir en una forma legible por la máquina, lo que se conoce como la programación. Si se lleva a cabo el diseño de una forma detallada, la generación de código se realiza casi en forma automática. Una vez que se ha generado el código, comienzan las pruebas del programa, las cuales se realizan para la detección de errores y asegurar que la entrada definida produce resultados reales de acuerdo con los resultados esperados.

- **Operación y Mantenimiento.** Finalmente se obtendrá la aceptación del usuario para proceder con la implantación del sistema bajo la arquitectura solicitada, en algunos casos será necesario establecer un plan de capacitación, para el personal que operará el usuario.

El software sufrirá cambios después de ser entregado al cliente (a menos que sea un software empotrado como los cajeros), ya sea porque se han encontrado errores, porque el software debe adaptarse a los cambios de su entorno externo (por ejemplo: un cambio debido a un

dispositivo periférico nuevo), o porque el cliente requiere mejoras funcionales o de rendimiento. El soporte y mantenimiento del software vuelve a aplicar cada una de las fases precedentes a un programa ya existente y no a uno nuevo.

Entre los problemas que se encuentran:

- Los proyectos reales raras veces siguen el modelo secuencial en todas sus fases, lo que produce que los cambios causen confusión.
- Es común que el cliente no exponga claramente todos los requerimientos. El modelo necesita de ello y es difícil incorporar estos a último momento.
- El cliente debe tener paciencia. Una versión del programa no estará disponible hasta que el proyecto esté muy avanzado.

El primer ciclo de vida del software, "Cascada", fue definido por Winston Royce a fines del 70. Desde entonces muchos equipos de desarrollo han seguido este modelo. Sin embargo, ya desde 10 a 15 años atrás, el modelo cascada ha sido sujeto a numerosas críticas, debido a que es restrictivo y rígido, lo cual dificulta el desarrollo de proyectos de software moderno. En su lugar, muchos modelos nuevos de ciclo de vida han sido propuestos, incluyendo modelos que pretenden desarrollar software más rápidamente, o más incrementalmente o de una forma más evolutiva, o precediendo el desarrollo a escala total con algún conjunto de prototipos rápidos.

2.4.2 Modelo de Construcción de Prototipos de Requerimientos.

Un prototipo es construido de una manera tan rápida como sea posible. Esto es con el fin de que los usuarios, clientes o representantes de ellos, experimenten con el prototipo. Estos individuos luego proveen la retroalimentación sobre lo que a ellos les gustó y no les gustó acerca del prototipo proporcionado.

A continuación se representa en la figura 2.3, en forma muy concreta los pasos establecidos para seguir este modelo.

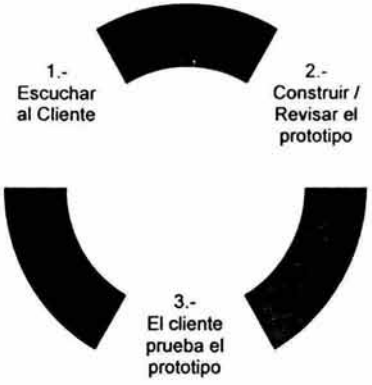


Figura 2.3.- Modelo de Prototipos.

El paradigma de construcción de prototipos comienza con la recolección de requisitos. Entonces aparece un "diseño rápido". El diseño rápido lleva a la construcción de un prototipo. El prototipo se pone a punto para satisfacer las necesidades del cliente, permitiendo al mismo tiempo que el desarrollador comprenda mejor lo que se desea hacer.

El Prototipo ha sido usado frecuentemente en los 90`s, porque la especificación de requerimientos para sistemas complejos tienden a ser relativamente difíciles de entender. Muchos usuarios y clientes encuentran que es mucho más fácil proveer retroalimentación basándose en la manipulación desde un prototipo, en vez de leer una especificación de requerimientos potencialmente ambiguos y extensos.

Sin embargo, un prototipo podría no ser suficiente para la culminación del software, por lo que se tendría que crear una versión rediseñada en la que se resuelvan todas las observaciones y aparentemente desechar la primera versión. Por lo que se muestra una clara deficiencia.

Básicamente el modelo de prototipos puede adecuarse a los pasos establecidos en el modelo cascada, he inclusive con retroalimentación como lo observamos en la figura 2.4

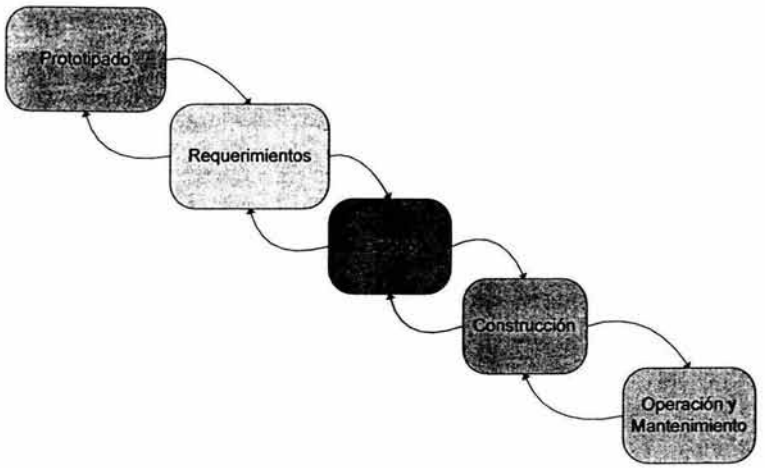


Figura 2.4.- Prototipo de Requerimientos basado en el modelo Cascada.

2.4.3 Modelo de Desarrollo Incremental.

Los riesgos asociados con el desarrollo de sistemas largos y complejos son enormes. Una forma de reducir los riesgos es construir sólo una parte del sistema, reservando otros aspectos para niveles posteriores. El modelo incremental combina elementos de modelo lineal secuencial (aplicados repetidamente) con la filosofía interactiva de construcción de prototipos. El modelo incremental aplica secuencias lineales de forma escalonada mientras el software va madurando en el paso del tiempo. Cada secuencia lineal produce un "incremento" del software[8].

[8] McDermid, J. y P. Rook, "Software Development Process Models", CRC Press 1993

Note que el desarrollo incremental es 100% compatible con el modelo en cascada. El desarrollo incremental no demanda una forma específica de observar el desarrollo de algún otro incremento. Así el modelo en cascada puede ser usado para administrar cada esfuerzo de desarrollo, como se muestra en la figura 2.5

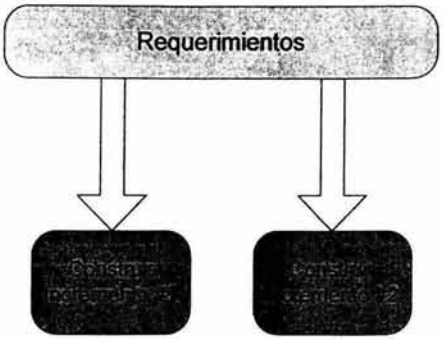


Figura 2.5.- Modelo de Desarrollo Incremental.

O bien puede adecuarse al modelo de construcción de prototipos (figura 2.6).

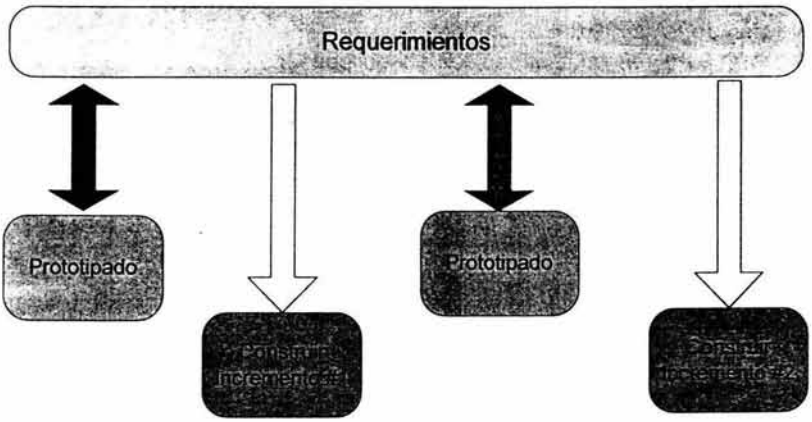


Figura 2.6 - Prototipo de Requerimientos basado en el modelo de desarrollo incremental.

A continuación observamos el modelo incremental en base al modelo cascada (figura 2.7).

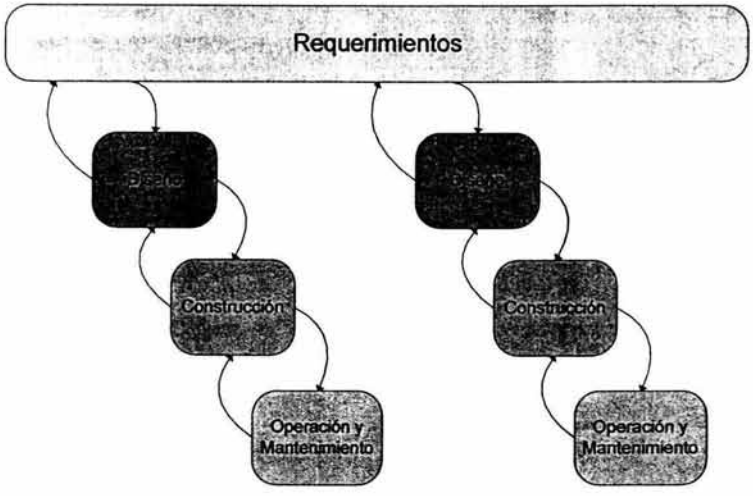


Figura 2.7.- Modelo de Desarrollo Incremental con desarrollo en cascada de los incrementos.

El modelo de desarrollo incremental provee algunos beneficios significativos para los proyectos:

- Construir un sistema pequeño es siempre menos riesgoso que construir un sistema grande.
- Al ir desarrollando parte de las funcionalidades, es más fácil determinar si los requerimientos planeados para los niveles subsiguientes son correctos.
- Si un error importante es realizado, sólo la última iteración necesita ser descartada.
- Reduciendo el tiempo de desarrollo de un sistema (en este caso en incremento del sistema) decrecen las probabilidades que esos requerimientos de usuarios puedan cambiar durante el desarrollo.
- Si un error importante es realizado, el incremento previo puede ser usado.
- Los errores de desarrollo realizados en un incremento, pueden ser arreglados antes del comienzo del próximo incremento.

Cuando se utiliza un modelo incremental el primer incremento a menudo es un producto esencial. El cliente utiliza el producto, como consecuencia de la utilización y/o de evaluación, se desarrolla un plan para el incremento siguiente. Este proceso se repite siguiendo la entrega de cada incremento, hasta que se elabore el producto completo.

2.5 Modelos Evolutivos de Software.

Los modelos evolutivos son interactivos. Se caracterizan por la forma en que permite a los ingenieros del software desarrollar versiones cada vez mas completas del software.

2.5.1 El modelo Espiral.

El modelo espiral, propuesto originalmente por Boehm[9], es un modelo de software evolutivo que conjuga la naturaleza iterativa de construcción de prototipos con los aspectos controlados y sistemáticos del modelo lineal secuencial.

En este modelo, el esfuerzo de desarrollo es iterativo. Tan pronto como uno completa un esfuerzo de desarrollo, otro comienza. Además, en cada desarrollo ejecutado, puedes seguir estos cuatros pasos:

1. Determinar qué quieres lograr.
2. Determinar las rutas alternativas que puedes tomar para lograr estas metas. Por cada una, analizar los riesgos y resultados finales, y seleccionar la mejor.
3. Seguir la alternativa seleccionada en el paso 2.
4. Establecer qué tienes terminado.

[9] Boehm, B. "A Spiral Model for Software Developement and Enhancement", Computer vol. 21, no. 5 Mayo 1988

Para poder explicar en forma clara, observemos la figura 2.8.

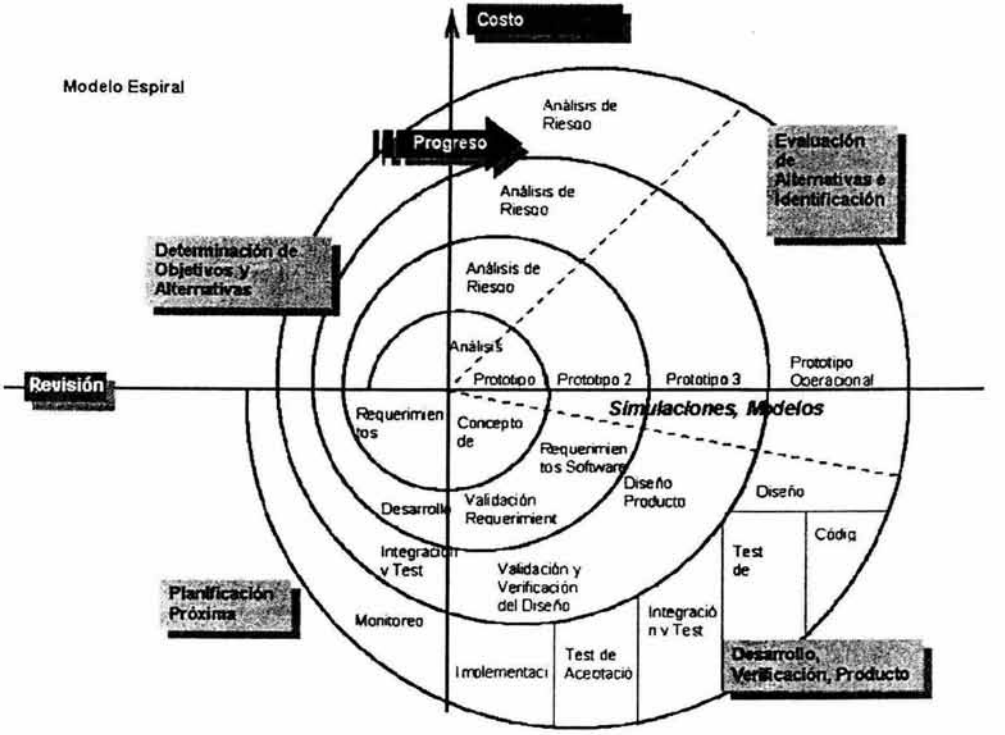


Figura 2.8.- Modelo en espiral.

La dimensión radial en la figura 2.8 refleja costos acumulativos incurridos en el proyecto. Observemos un escenario particular. Digamos que en este proyecto, nosotros viajaremos a resolver un conjunto particular de problemas del cliente. Durante el primer viaje alrededor de la espiral, analizamos la situación y determinamos que los mayores riesgos son la interfaz del usuario. Después de un cuidadoso análisis se construye un prototipo. Lo realizamos. Luego proveemos el prototipo al cliente quien nos provee con retroalimentación útil. Ahora, comenzamos el segundo viaje alrededor de la espiral.

El modelo espiral captura algunos principios básicos:

- Decidir qué problema se quiere resolver antes de viajar a resolverlo.
- Examinar las múltiples alternativas de acción y elegir una de las más convenientes.
- Evaluar qué tienes hecho y qué tienes que haber aprendido después de hacer algo.
- No ser tan ingenuo para pensar que el sistema que estás construyendo será "EL" sistema que el cliente necesita, y
- Conocer y comprender los niveles de riesgo, que tendrás que tolerar.

A diferencia del modelo de proceso clásico que termina cuando se entrega el software, el modelo en espiral puede adaptarse y aplicarse a lo largo de la vida del software de computadora.

Pero al igual que otros paradigmas, el modelo en espiral no es la panacea. Puede resultar difícil convencer a grandes clientes (particularmente en situaciones bajo contrato) de que el enfoque evolutivo es controlable.

2.6 Metodología de la Ingeniería de Software.

Hasta este punto hemos visto una definición de la Ingeniería de software, su clasificación, el ciclo de vida de un software, así como el modelo que trata de representar la manera de crear un software (dividiendo éste en etapas de desarrollo), sin embargo, es necesario entender "que es" y "como" se tienen que realizar cada una de estas etapas, en otras palabras es necesaria la utilización de una metodología.

Una metodología es la integración entre un método y una técnica, la cual nos indica el qué y como se tienen que realizar las cosas a fin de lograr una meta (figura 2.9).

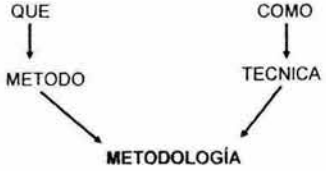


Figura 2.9.- Integración de una metodología.

2.7 Análisis y Diseño Estructurado.

La metodología clásica que se ha utilizado en la modelación de sistemas es sin duda el Análisis y Diseño estructurado, el cual surge a finales de los años 60´s e inicios de los 70´s como resultado de la necesidad de buscar una forma interpretativa más rápida y eficiente, de tal manera que se pudiesen definir los requerimientos del usuario y las especificaciones funcionales del sistema.

Sin embargo en aquel momento, no había un entorno favorable ya que existían grandes volúmenes de información que había que leer por completo y que acarreaban una serie de problemas de: redundancia, ambigüedad e imposibilidad de mantener. Es por ello que surge una amplia variedad de diagramas que permiten representar las especificaciones funcionales en forma sencilla y rápida, aumentando por ende el grado de comunicación entre las especificaciones funcionales y el usuario final.

El objetivo que persigue el análisis y diseño estructurado es organizar las tareas asociadas con la determinación de requerimientos para obtener la comprensión completa y exacta de una situación dada.

Los componentes del análisis estructurado son los siguientes: símbolos gráficos, diccionarios de datos, reglas, descripciones de procesos y procedimientos, los cuales nos entregan los siguientes productos:

- Diagrama de Flujo de Datos (DFD)
- Diccionario de Datos (DD)
- Mini Especificaciones (ME)

2.7.1 Diagrama de Flujo de Datos (DFD).

La técnica de diagrama de flujo de datos (DFD) tiene por objetivo representar gráficamente el sistema a nivel lógico y conceptual, ilustrando los componentes esenciales de un proceso y la forma en que interactúan, permitiendo así comprender los procedimientos existentes con la finalidad de optimizarlos, reflejándolos en el sistema propuesto.

Los elementos que componen a un DFD son los siguientes, representado en la figura 2.10.

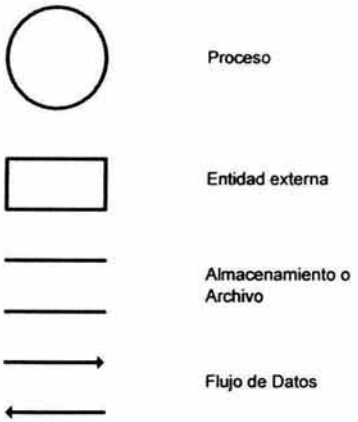


Figura 2.10.- Elementos del DFD.

2.7.1.1 Niveles del DFD.

El DFD posee niveles de desagregación o explosión o apertura de burbujas.

- El Nivel 0 ó Diagrama de Contexto.

Es aquel que muestra una sola burbuja y las entidades externas o terminadores con los que interactúa el sistema.

- No existirán almacenes o archivos.
- Se representarán las entidades externas que son fuente y destino de los datos.
- El sistema será representado como un proceso simple.
- Se dibujarán sólo los flujos de datos de comunicación exterior-sistema.

En la figura 2.11 se observa el DFD nivel 0 (Diagrama de contexto) del proceso: "Registro de un cliente por Internet"

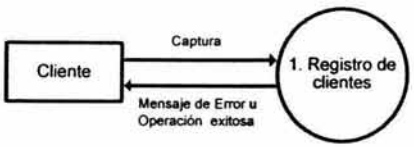


Figura 2.11.- DFD Nivel 0 o Diagrama de contexto.

- Nivel 1 y subsiguientes.
  - Deberá haber igual cantidad de archivos. Aunque podrá existir mayor cantidad de almacenamientos en el nivel 2 debido a la explosión de algún proceso.
  - En el último nivel, cada proceso realizará una función específica y concreta.

A continuación se muestra el nivel 1 del proceso: "Registro de un cliente por Internet" figura 2.12

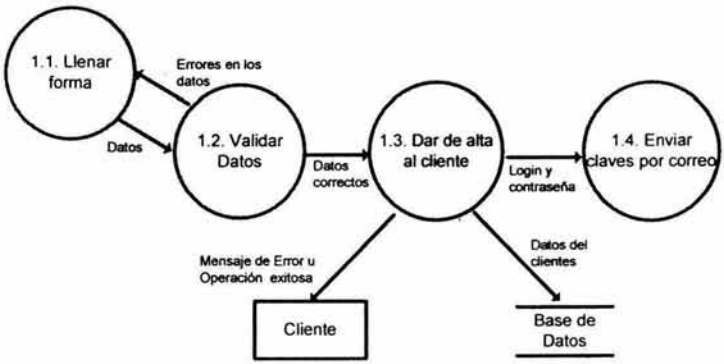


Figura 2.12.- DFD Nivel 1.

En síntesis, el Diagrama de Flujo de Datos describe:

- Los lugares de origen y destino de los datos (los límites del sistema).
- Las transformaciones a las que son sometidos los datos (los procesos internos).
- Los lugares en los que se almacenan los datos dentro del sistema, y
- Los canales por donde circulan los datos.

2.7.2 Diccionario de Datos (DD).

Como ya se ha observado, realizar cada una de las etapas del análisis estructurado es complejo y requiere de un enfoque organizado para representar las características de cada proceso, así como la información que se maneja en cada c/u de ellos. Esto se realiza con el diccionario de datos.

El diccionario de datos es un listado organizado de todos los elementos de datos que son necesarios para el sistema, con definiciones precisas y claras que permiten que el usuario y el analista del sistema tengan una misma comprensión de la entradas, salidas, así como de los procesos intermedios[10].

[10] Yourdon, E.N., Modern Structured Analysis, Prentice Hall, 1990

Aunque el formato del diccionario varía entre las distintas herramientas, la mayoría utiliza la siguiente nomenclatura (figura 2.13).

Construcción de datos	Notación	Significado
Agregación	=	Igual o equivalencia
Secuencia	+	Concatenación
Selección	[|]	Uno u otro
Repetición	{ }n	n repeticiones de
	( )	Datos opcionales
	* ... *	Delimitadores de comentarios

Figura 2.13.- Nomenclatura utilizada en un Diccionario de datos.

Tomando como ejemplo la descripción de una ficha bibliográfica, utilizaremos el siguiente formato, únicamente para describir la ficha bibliográfica, Autor y el Título de cortesía (figura 2.14).

Nombre	Descripción	Estructura
Ficha bibliográfica	Proporciona los datos sobre un libro, artículo, revista o periódico.	Ficha bibliográfica = Información Básica + Información de resumen
Autor	Nombre de la persona que escribió libro	Nombre =Título de cortesía + Apellido Paterno + (Apellido Materno) + Nombre
Título de cortesía	Lo que establece el derecho de uno a una distinción honorífica o bien a ejercer un empleo o profesión.	Título de cortesía = [Sr. | Sra. | Srita. | Dr. | Profesor]

Figura 2.14.- Diccionario de datos, representando una ficha bibliográfica.

2.7.3 Mini Especificaciones (ME).

2.7.3.1 La Descripción Narrativa.

Dado que las especificación de requerimientos sólo indican que es lo que se desea obtener del sistema, es necesario contar con el detalle de los procesos el cual exprese el funcionamiento interno de los procesos, ante esta situación la descripción narrativa, ofrece conocer a fondo los procesos, evitando así posibles ambigüedades y diferencias entre procesos.

2.7.3.2 El Pseudocódigo.

El pseudocódigo es una herramienta algorítmica que permite escribir la imitación de un programa real utilizando un lenguaje de muy similar a los lenguajes de programación de alto nivel. Así, un pseudocódigo es una combinación de símbolos (+, -, *, /, %, >, >=, <, <=, !=, ==, y, o, no), términos (Leer, Imprimir, Abrir, Cerrar, Hacer...Mientras, Mientras...Hacer, Para...Mientras, etc.) y otras características comúnmente utilizadas en uno o más lenguajes de alto nivel.

No existen reglas que determinen que es o no es un pseudocódigo, sino que varía de un programador a otro. El objetivo del pseudocódigo es permitir al programador centrarse en los aspectos lógicos de la solución evitando las reglas de sintaxis de un lenguaje de programación. Posteriormente el pseudocódigo debe ser traducido a un programa usando un lenguaje de programación de alto nivel como Java, Visual Basic, C++, C, etc.

A continuación veremos un ejemplo:

Algoritmo para preparar una limonada.

```
INICIO
    Llenar una jarra con un litro de agua
    Echar el jugo de tres limones
    Echar cuatro cucharadas de azúcar
    Remover el agua hasta disolver completamente el azúcar
FIN
```

Algoritmo que permite hallar la suma y el promedio de tres números.

```
INICIO
    LEER numero1, numero2, numero3
    suma = numero1 + numero2 + numero3
    promedio = suma / 3
    IMPRIMIR suma, promedio
FIN
```

A este último algoritmo, el cual se explico mediante el pseudocódigo, es necesario utilizar la descripción narrativa, a fin explicar los siguientes términos:

- El término **LEER** significa obtener un dato de algún dispositivo de entrada, como el teclado, y almacenarlo en una variable.

- Una variable es una localización en la memoria que tiene un nombre y cuyo contenido puede cambiar a lo largo de la ejecución de un programa. Así **numero1, numero2 y numero3** son variables.
- El término **IMPRIMIR** significa mostrar el valor de una variable en algún dispositivo de salida, como la pantalla.

2.7.3.3 Árboles de Decisiones.

Uno de los mejores métodos para el análisis de una decisión es el llamado árbol de la decisión. En este se describen gráficamente en forma de árbol, los puntos de decisión, hechos aleatorios y probabilidades de los diversos cursos de acción que podrían seguirse.

Tomemos por ejemplo uno de los problemas más comunes de las empresas al introducir un nuevo producto. Los administradores deben decidir en este caso, si instalar un costoso equipo permanente para garantizar la producción al menor costo posible o si efectuar un montaje temporal más económico que suponga mayores costos de manufacturaron por menores inversiones de capital y resulte en menores perdidas, en el caso de que las ventas del producto no respondan a las estimaciones.

A continuación se muestra los el árbol de decisión ante esta situación (figura 2.15).

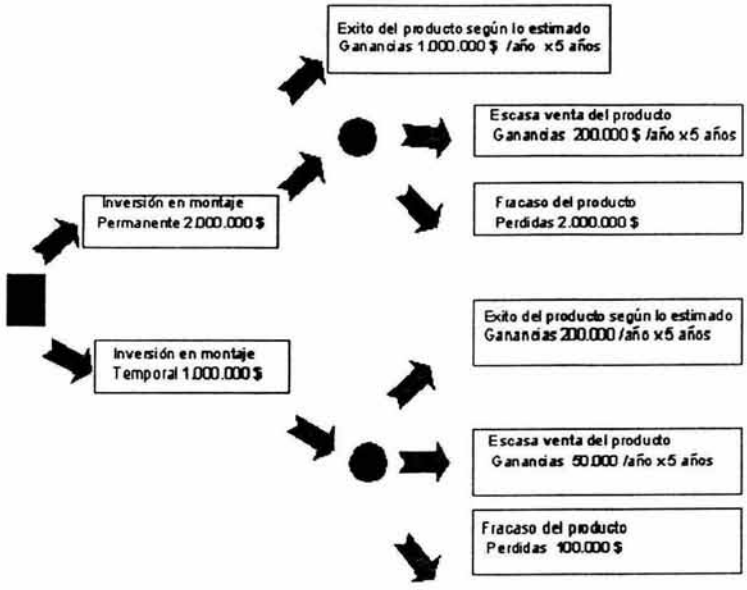


Figura 2.15.- Árbol de decisión ante la introducción de un nuevo producto.

En donde la simbología utilizada refleja: (figura 2.16).

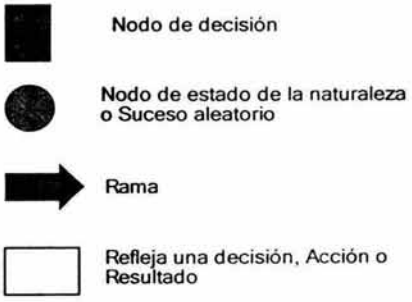


Figura 2.16.- Simbología utilizada en un Árbol de decisión.

Los árboles de decisión ilustran la manera en que se pueden desglosar los problemas y la secuencia del proceso de decisión. A continuación explicaremos el uso del de los árboles de decisión:

- **Rama.** es un arco conector.
- **Nodo de decisión.** Representa un punto en el que se debe tomar una decisión. Se representa con un cuadrado. De este salen ramas de decisión que representan las decisiones posibles.
- **Nodo de estado de la naturaleza.** representa el momento en que se produce un evento incierto. Se representa con un círculo. De el salen ramas de estado de la naturaleza que representan los posibles resultados provenientes de eventos inciertos sobre los cuales no se tiene control.
- **Secuencia temporal.** Se desarrolla de izquierda a derecha.
- **Ramas que llegan a un nodo desde la izquierda**. Estas ya ocurrieron.
- **Ramas que salen hacia la derecha**. Ellas aún no han ocurrido.

Posiblemente una limitación en los árboles de decisión, es que a pesar de dar una buena descripción visual en problemas relativamente simples, su complejidad aumenta exponencialmente a medida que se agregan etapas adicionales.

2.7.3.4 Tablas de Decisiones.

Las tablas son muy parecidas a los árboles de decisión. En la primera columna, que esta divida por 2 partes, se ponen en la primera parte todos los condicionales. En esa misma columna, en la segunda sección, se colocan todas las decisiones posibles que se pueden llegar a alcanzar.

Después, a la derecha, van N columnas, siendo N el resultado de la productora de la cantidad de respuestas posibles para cada condicional. (Me explico: si hacen una condicional que tiene tres posibles estados y otra con dos posibles estados, deberían hacer 3*2=6 columnas para cubrir todas las posibilidades.)

Una ventaja de esta tabla es que hace prácticamente evidente si hay una forma de simplificar las condiciones y son muy fáciles de verificar su validez lógica.

Muchos procesos de toma de decisiones pueden ser tratados por medio de **tablas de decisión**, en las que se representan los elementos característicos de estos problemas:

- Los diferentes estados que puede presentar la naturaleza: $e_1, e_2, ..., e_n$.
- Las **acciones o alternativas** entre las que seleccionará el decisor: $a_1, a_2, ..., a_m$.
- Las **consecuencias o resultados** x_{ij} de la elección de la alternativa a_i cuando la naturaleza presenta el estado e_j.

Se supone, por simplicidad, la existencia de un número finito de estados y alternativas. El formato general de una tabla de decisión es el siguiente (figura 2.17).

		Estados de la Naturaleza			
Alternativas		e_1	e_2	...	e_n
	a_1	x_{11}	x_{12}	...	x_{1n}
	a_2	x_{21}	x_{22}	...	x_{2n}
	...	...	...	...	...
	a_m	x_{m1}	x_{m2}	...	x_{mn}

Figura 2.17.- Formato utilizado generalmente para una Tabla de decisión.

Finalmente y a fin de dejar más claro este concepto analizaremos el siguiente ejemplo:

Un ama de casa acaba de echar cinco huevos en un tazón con la intención de hacer una tortilla. Dispone, además, de un sexto huevo del que no conoce su estado, aunque es de esperar que en caso de encontrarse en buen estado y no ser utilizado, se estropeará. Al ama de casa se le presentan tres posibles alternativas:

- Romper el huevo dentro del tazón donde se encuentran los cinco anteriores.
- Romperlo en otro tazón diferente.

- Tirarlo directamente.

Dependiendo del estado del huevo, las consecuencias o resultados que pueden presentarse para cada posible alternativa se describen en la figura 2.18.

 Alternativas	Estado del 6º huevo	
	Bueno (e_1)	Malo (e_2)
Romperlo dentro del tazón (a_1)	Tortilla de 6 huevos	5 huevos desperdiciados y no hay tortilla
Romperlo en otro tazón (a_2)	Tortilla de 6 huevos y un tazón más que lavar	Tortilla de 5 huevos y un tazón más que lavar
Tirarlo (a_3)	Tortilla de 5 huevos y un huevo bueno desperdiciado	Tortilla de 5 huevos

Figura 2.18.- Tabla de decisión.

2.8 Los modelos de ciclo de vida y la metodología utilizada son complementarios.

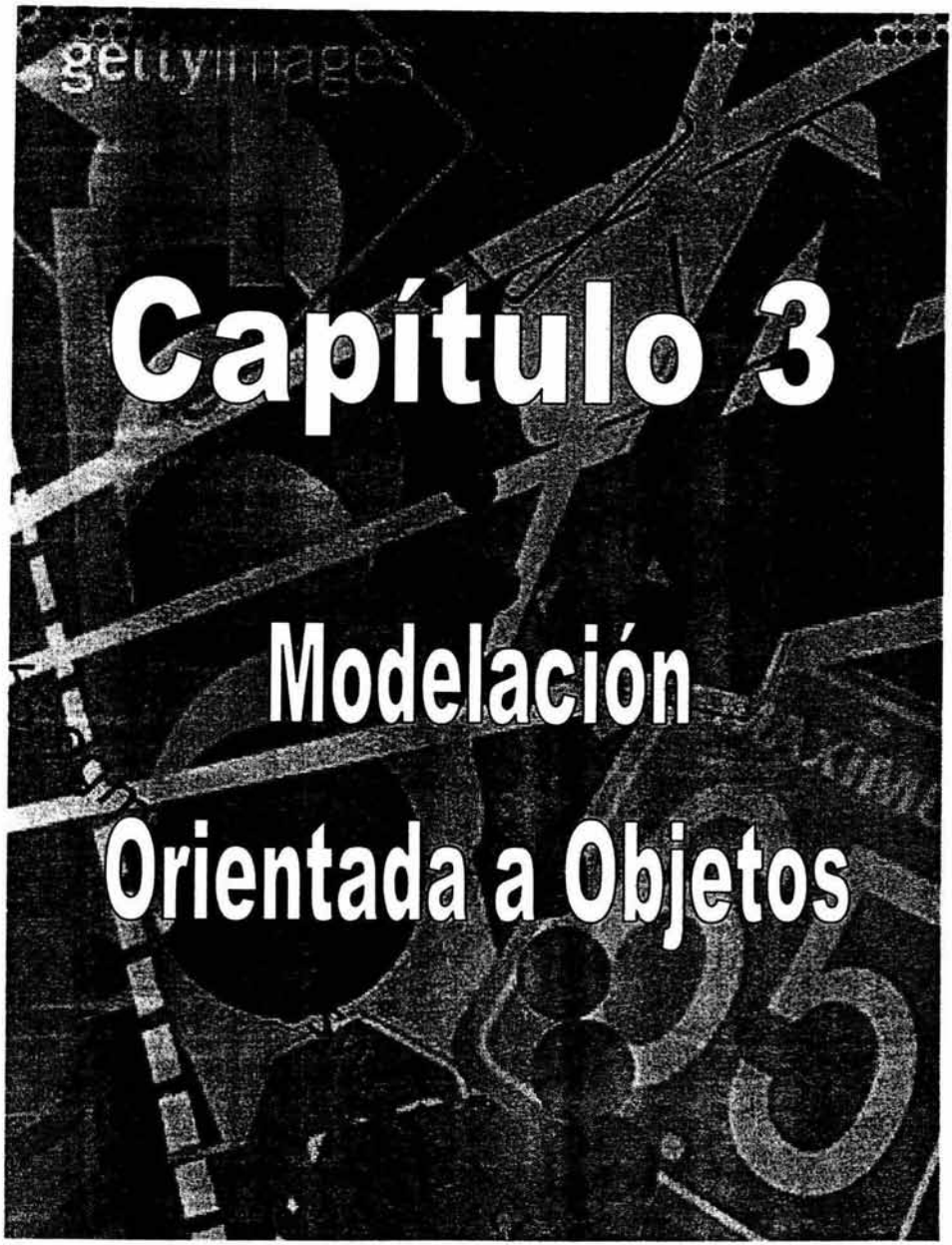
El cambio es una propiedad intrínseca del software. Hoy en día el software debe poseer un enfoque evolutivo. El software cambia constantemente, debido a la necesidad de repararlo (eliminando errores no detectados anteriormente) como a la necesidad de apoyar la evolución de los sistemas a medida que aparecen nuevos requerimientos o cambian los antiguos.

Por lo cual es importante enfatizar que, no tiene sentido entonces que un proyecto tome estrictamente una decisión concerniente con cual modelo se adherirá. Los modelos de ciclo de vida presentados, son complementarios en vez de excluyentes.

A pesar de la importancia que tiene la Ingeniería de Software (IS) ha costado mucho que se le preste la atención adecuada a esta actividad. Aún quedan muchos desafíos que

deben ser mejorados, cada actividad y técnica de la IS utilizada individualmente, dará diferentes soluciones para diferentes proyectos. Por esta razón, se considera que no existe un modelo de proceso ideal para la IS; encontrar el método o la técnica perfecta es una ilusión, pues cada método y técnica ofrece diferentes soluciones ante un problema.

En muchos casos, deben de combinarse los modelos con la intención de tomar las ventajas de cada uno de ellos en un proyecto. Por lo tanto no hay necesidad de ser dogmático en la elección de los paradigmas, de esta forma para la ingeniería de software: la naturaleza de la aplicación debe dictar el modelo a elegir, con la mejor metodología.



Capítulo 3

Modelación Orientada a Objetos

3.1 Historia de la Orientación a Objetos.

La ola con tendencia hacia la Orientación a Objetos (OO) surge en Noruega en 1967 con un lenguaje llamado Simula 67, desarrollado por Krinsten Nygaard y Ole-Johan Dahl, en el centro de cálculo noruego. Simula 67 introdujo por primera vez los conceptos de clases, corrutinas y subclases (conceptos muy similares a los lenguajes Orientados a Objetos de hoy en día). Uno de los problemas de inicio de los años setentas consistía en cómo adaptar el software a nuevos requerimientos imposibles de haber sido planificados inicialmente.

En los 70's científicos del centro de investigación en Palo Alto Xerox (Xerox park) inventaron el lenguaje Smalltalk que dio respuesta al problema anterior (investigar no planificar). Smalltalk fue el primer lenguaje Orientado a Objetos puro de los lenguajes Orientados a Objetos, es decir, únicamente utiliza clases y objetos.

Hasta este momento uno de los defectos más graves de la programación es que las variables eran visibles desde cualquier parte del código y podían ser modificadas incluyendo la posibilidad de cambiar su contenido.

Quien tuvo la idea fue Parnas[1] D. L. cuando propuso la disciplina de ocultar la información. Su idea era encapsular cada una de las variables globales de la aplicación en un solo módulo junto con sus operaciones asociadas, sólo mediante las cuales se podía tener acceso a esas variables. El resto de los módulos (objetos) podían acceder a las variables sólo de forma indirecta mediante las operaciones diseñadas para tal efecto.

En los años 80's Stroustrup[2] de AT&T Labs., amplió el lenguaje C para crear C++ que soporta la programación Orientada a Objetos.

En esta misma década se desarrollaron otros lenguajes Orientados a Objetos como Objective C, Common Lisp Object System (CLOS), Pascal, Ada y otros.

[1] Dr. David Lorge Parnas 1976, "tipos abstractos definidos como clases de variables", procedimientos de la conferencia sobre datos: Abstracción, definición, y estructura , ciudad del lago salt, marcha de 1976, pp. 22-24. También En: Informe 7998, laboratorio de investigación naval de Estados Unidos, C.C. de Washington, abril de 1976, pp. 1-10 del memorándum de NRL.

[2] Bjarne Stroustrup 1985, es el diseñador y el ejecutor original de C++ y del autor del "el lenguaje de programación de C++" (1ra edición 1985, 2dos 3ro edición 1997 de la edición 1991 "" edición especial 2000 1997) y el diseño y la evolución de C++ . Sus intereses de la investigación incluyen sistemas distribuidos, sistemas operativos, la simulación, el diseño, y la programación.

Posteriores mejoras en herramientas y lanzamientos comerciales de C++ por distintos fabricantes, justificaron la mayor atención hacia la programación Orientada a Objetos en la comunidad de desarrollo de software. El desarrollo técnico del hardware y su disminución del costo fue el detonante final.

En el inicio de los 90's se consolida la Orientación a Objetos como una de las mejores maneras para resolver problemas. Aumenta la necesidad de generar prototipos más rápidamente, surgiendo el concepto RAD (Rapid Aplication Developments). Sin esperar a que los requerimientos iniciales estén totalmente precisos.

En 1996 surge un desarrollo llamado JAVA (extensión de C++). Su filosofía es aprovechar el software existente, el cual facilita la adaptación del mismo a otros usos diferentes a los originales sin necesidad de modificar el código ya existente.

En 1997-1998 se desarrollan herramientas "CASE" orientadas a objetos como el diseño asistido por computadora (CAD).

De 1998 a la fecha se desarrolla la arquitectura de objetos distribuidos **RMI, Corba, COM, DCOM.**

Actualmente la orientación a objetos parece ser el mejor paradigma, no obstante, no es una solución a todos los problemas. Trata de eliminar la crisis del software.

Entre los creadores de metodologías orientadas a objetos se encuentran: G. Booch, Rambaught, Ivar Jacobson y Peter Cheng.

3.2 ¿Qué es la Orientación a Objetos?.

La Orientación a Objetos es un paradigma, es decir, es un modelo para aclarar algo o para explicarlo. La Orientación a Objetos es el paradigma que mejora el diseño, desarrollo y mantenimiento del software ofreciendo una solución a largo plazo a los problemas y preocupaciones que han existido desde el comienzo del desarrollo del software.

Es una manera de pensar, otra manera de resolver un problema; de lo más reciente en metodologías de desarrollo de software. Es un proceso mental humano aterrizado en una

computadora. Antes se adecuaba el usuario al entendimiento de la computadora. Actualmente, se le enseña a la computadora a entender el problema.

Entre los principales problemas que han existido para el desarrollo del software destacan:

- La falta de portabilidad del código.
- Su reusabilidad.
- La modificación (que antes era difícil de lograr).
- Ciclos de desarrollo largo.
- Técnicas de programación no intuitivas.

La Orientación a Objetos está basada en los tres métodos de organización que os desde la infancia:

- Entre un objeto y sus atributos (automóvil > marca, color, número de llanta
- Entre un objeto y sus componentes donde incluso otros objetos pueden fo e parte o otros objetos (agregación) (camión > motor, parabrisas, llantas).
- Entre un objeto y su relación con otros objetos (camión > vehículos automotores; una bicicleta no entraría en esta relación).

La metodología del software orientado a objetos consiste en:

- Saber el espacio del problema.
- Realizar una abstracción.
- Crear los objetos (espacio de la solución).
- Instanciarlos (esto es, traerlos a la vida).

- Dejarlos vivir (ellos ya saben lo que tienen que hacer).

El enfoque Orientado a Objetos (OO) busca reducir las deficiencias que se presentan en cada una de las etapas del ciclo de vida de la Ingeniería de Software convencional, permitiendo obtener una mejor representación del mundo y de los requerimientos particulares de una aplicación en dicho mundo. Este enfoque puede ser aplicado indistintamente al análisis, diseño o desarrollo de una aplicación. No es estrictamente necesario usar el enfoque en todas las etapas del ciclo de vida de una aplicación. Si se desea, se puede elaborar un buen análisis y diseño OO, aún cuando la implementación no necesariamente siga el mismo esquema. Sin embargo, es una excelente alternativa usar OO en todo el ciclo de vida, buscando aprovechar al máximo todas las bondades de este nuevo paradigma.

El concepto OO como paradigma, según Greiff[3]: "es una forma de pensar, una filosofía, de la cual surge una cultura nueva que incorpora técnicas y metodologías diferentes. Tomando una postura un tanto ontológica, es decir, que aspira a explicar el origen de las ideas, de la realidad; por lo que se considera que el universo computacional está poblado por objetos, cada uno responsabilizándose por sí mismo, y comunicándose con los demás por medio de mensajes".

Bajo este concepto entendemos que el modelo mejora el diseño, desarrollo y mantenimiento del software ofreciendo una solución a largo plazo a los problemas y preocupaciones que han existido desde el comienzo del desarrollo del software.

Existen conceptos ligados en torno a la tecnología orientada a objetos: el paradigma, los principios, el análisis y el diseño, mismos que a continuación se comentan.

[3] Greiff W. R. Paradigma vs Metodología: El Caso de la POO (Parte II). Soluciones Avanzadas. Ene-Feb 1994. pp. 31-39.

3.3 Fundamentos de Orientación a Objetos.

3.3.1 El paradigma Orientado a Objetos se basa en el concepto "Objeto".

Un objeto es aquello que cuenta con propiedades más valores (una avioneta tiene marca, color, alas, número de motores, capacidad de pasajeros, etc.), comportamiento (acciones y reacciones a mensajes) e identidad (propiedad que lo distingue de los demás objetos). La estructura y comportamiento de objetos similares están definidos en su clase común (transportes aéreos). Una clase es un conjunto de objetos que comparten una estructura y comportamiento común.

La diferencia entre un objeto y una clase es que un objeto es una entidad completa y concreta que existe en tiempo y espacio, mientras que una clase puede representar solo una parte del objeto, de manera que una clase representa la abstracción de la "esencia" de un objeto. De aquí que un objeto no es una clase.

Los principios del modelo OO son: abstracción, encapsulación, modularidad y herencia principalmente, y en menor grado tipificación (typing), concurrencia, persistencia y polimorfismo. Booch dice que si un modelo que se dice OO no contiene alguno de los primeros cuatro elementos, entonces no es OO, a continuación explicaremos los conceptos antes mencionados:

- **Abstracción.** Es una descripción simplificada o especificación de un sistema que enfatiza algunos de los detalles o propiedades del sistema, mientras suprime otros.

- **Encapsulación.** Se consigue al ocultar información. Es el proceso de ocultar todos los detalles de un objeto que no contribuyen a sus características esenciales.

- **Modularidad.** Divide un sistema, de acuerdo a su complejidad, en varios subsistemas hasta cubrir toda la complejidad del sistema inicial. Cada subsistema toma una actividad muy bien definida y delimitada.

- **Herencia.** Es una jerarquía entre clases que permite crear nuevas clases con base en una o más clases ya existentes. La nueva clase toma todas las características y el comportamiento de las clases de las cuales se hereda y puede agregar otras. Esto permite reutilizar código y generar nuevos programas por extensión.

- **Tipificación.** Se refiere al uso de datos abstractos Es la definición precisa de un objeto de tal forma que objetos de diferentes tipos no puedan ser intercambiados o, cuando mucho, puedan intercambiarse de manera muy restringida.

- **Concurrencia.** Cuando dos procesos ocurren en paralelo. El objeto uno puede comportarse de una manera, mientras que a su vez el objeto dos se puede estar comportando de otra manera en la misma instancia de tiempo.

- **Persistencia.** Es la propiedad de un objeto a través de la cual su existencia trasciende el tiempo (es decir, el objeto continua existiendo después de que su creador ha dejado de existir) y/o el espacio (es decir, la localización del objeto se mueve del espacio de dirección en que fue creado).

- **Polimorfismo.** Es cuando un método funciona de varias maneras según el objeto que invoque a dicho método.

Según Booch[4] los beneficios del enfoque Orientado a Objetos son:

- Primero, el uso del modelo OO nos ayuda a explotar el poder expresivo de todos los lenguajes de programación basados en objetos y los orientados a objetos, como Smalltalk, Pascal, C++, CLOS, Ada y recientemente Java.

- Segundo, el uso del modelo OO alienta el reuso no solo del software, sino de diseños completos.

- Tercero, produce sistemas que están construidos en formas intermedias estables y por ello son más resistentes al cambio en especificaciones y tecnología.

Greiff Comenta: "Que el Análisis Orientado a Objetos (OOA por sus siglas en inglés de Object Oriented Analysis) "es un método de análisis que examina los requerimientos desde la perspectiva de las clases y objetos encontrados en el vocabulario del dominio del problema". Según Booch, el Diseño Orientado a Objetos "es un método de diseño abarcando el proceso

[4] Booch 1986 Booch G. 1998. Software Architecture and the UML. Presentación disponible en: http://www.rational.com/uml como arch.zip.

de descomposición orientado a objetos y una notación para representar ambos modelos lógico y físico".

3.3.2 Ventajas de usar el enfoque Orientado a Objetos.

Las ventajas de usar el enfoque OO se traducen en mejoramientos de calidad a lo largo del ciclo de vida de una aplicación, facilitando además el mantenimiento y la creación de nuevas versiones que extiendan el programa.

Al disminuir las barreras entre las etapas de análisis, diseño y desarrollo, se garantiza que se está hablando de las mismas cosas y en los mismos términos desde el comienzo del análisis hasta el final de la etapa de implementación. Esto evita inconsistencias y permite verificar que las cosas están claramente definidas y cumplen con todos los requerimientos, incluso antes de escribir una línea de código del programa. Las características anteriormente mencionadas (encapsulamiento, herencia, reutilización) permiten crear un software mucho más robusto.

Se pueden enunciar varios beneficios de la aproximación orientada por objetos:

- Reutilización de software. Permite describir clases y objetos que podrán ser usados en otras aplicaciones.

- Estabilidad. El diseñador piensa en términos de comportamiento de objetos, no en detalles de bajo nivel; diseño rápido y de alta calidad, puesto que se concentra en satisfacer los requerimientos y no en detalles técnicos; integridad; facilidad de programación al usar efectivamente toda la información de la fase de diseño, poniéndola en términos de un lenguaje específico.

- Facilidad de mantenimiento. Dado que al tener el modelo del mundo, es fácil realizar mantenimiento en términos de objetos, atributos y métodos de los mismos; independencia en el diseño, el diseño de un software se puede hacer independientemente de plataformas, software y hardware.

3.4 Metodologías de Análisis y Diseño Orientado a Objetos (OOAD).

En cuanto a las metodologías OO (por sus siglas en ingles OOAD "Object Oriented analysis and Design"), diremos que hay un gran número de métodos orientado a objetos actualmente. Muchos de los métodos pueden ser clasificados como orientados a objetos porque soportan de manera central los conceptos de la orientación a objetos. Algunos de las metodología más conocidas y estudiadas hasta antes del UML según Jacobson[5] son:

- Object Oriented Analysis and Design (OOAD), Booch.
- Object Modeling Technique (OMT), Rumbaugh.
- Object Oriented Analysis (OOA), Coad/Yourdon.
- OOSE Method, Jacobson.
- Object Oriented Systems Analysis (OOSA), Shaler y Mellor.

Actualmente las metodologías más importantes de análisis y diseño de sistemas han fusionado en lo que es el UML, bajo el respaldo las metodología de Booch, Rumbaugh y Jacobson.

Las metodologías OOAD caen en dos tipos básicos, los métodos ternarios, los cuales muestran una evolución natural de los métodos estructurados, sin embargo manejan notaciones separadas para datos, la modelación dinámica y los procesos. El tipo unitario confirma la causa del porque combinar procesos (métodos) y datos, sólo se necesita una anotación. El tipo unitario es considerado la metodología más cercana a los objetos y la más fácil de aprender desde el principio, sin embargo la desventaja de este es que los productos que surgen desde el análisis, son un tanto complicados para ser revisados por el usuario.

En las siguientes secciones, se describen las metodologías más representativas del análisis y diseño orientado a Objetos, conducido por Booch, Coad y Yourdon, Fusión, Jacobson, Rumbaugh, y Shlaer y Mellor.

[5] Jacobson, I. 1992. Object-Oriented Software Engineering; A Use Case Driven Aproach. ACM Press. Adison-Wesley Publishing. Co. U.S.A. 524 p. Ilus. pp. 465-493.

3.4.1 (OOAD) Análisis y Diseño Orientado a Objetos por Booch.

La metodología de Grady Booch orientada al OOAD es una del más populares, incluso soportadas en varias herramientas como es Visio o Rational Rose, producto lanzado por Rational Software Corporation, en donde él ha estado con como principal científico desde su fundación en los 80´s. (Hoy James Rumbaugh e Ivar Jacobson se han unido a la compañía Racional Software lanzado uno de los mejores productos en el mundo de OOAD).

La metodología de notación lógica de Booch[6] ,proporciona un desarrollo orientado al objeto en las fases del análisis y del diseño. La fase de análisis consta de ciertos pasos:

- El primer paso es establecer los requerimientos del cliente.
- El segundo paso es un análisis del dominio, esto es, realizar el análisis con cierta orientación a objetos, incluyendo los diagramas de estado.

Es recomendable utilizar los diagramas de estado con un número pequeño de fases, sin embargo, no es útil para sistemas con un gran número de transiciones. Una vez que un diagrama de transición de estados tiene más de 8 a 10 estados, se vuelve difícil de manejar.

La fase de análisis se termina al determinar las validaciones basadas en los requisitos del cliente.

Una vez que se termine la fase de análisis, la metodología de la notación lógica de Booch desarrolla la arquitectura en la fase del diseño. La fase del diseño es iterativa entre el diseño lógico, el diseño físico, los prototipos y la prueba.

La metodología de la notación lógica de Booch es secuencial en el sentido que la fase de análisis requiere terminar para continuar con la fase del diseño. Sin embargo, cada fase está compuesta de pasos cíclicos más pequeños. La metodología de Booch se concentra en la fase del análisis del diseño y no considera la puesta en práctica o la fase de prueba en mucho detalle.

[5] Booch G. 1998. Software Architecture and the UML. Presentación disponible en: http://www.rational.com/uml

3.4.2 (OOA/OOD) Análisis Orientado a Objetos / Diseño Orientado a Objetos por Coad y Yourdon.

Coad y Yourdon publican sus primeros libros prácticos en el OOAD (1990 y 1991). En los cuales los focos de la metodología en el análisis, y usos de negociación, dan una notación más amistosa que las de Booch, Shlaer y Mellor.

Según Coad y Yourdon[7] el principal resultado del análisis y diseño orientado a objetos (OOA/OOD) es la reducción de la complejidad de un problema y las responsabilidades del sistema dentro de él. El método de OOA/OOD se basa en una representación profunda y uniforme mediante clases y objetos que según Coad y Yourdon traen como consecuencia:

- La casi nula diferencia entre el análisis y las notaciones del diseño.
- Una transición transparente del análisis al diseño.
- Es también importante que los pasos descritos no tienen que ser seguidos en la orden secuencial dada, por lo que no sigue el modelo en cascada, espiral e incremental.
- Los analistas y diseñadores requieren conocer estrategias así como habilidades.
- Existe una representación uniforme entre el Análisis Orientado a Objetos (OOA) el Diseño Orientado a Objetos OOD y la Programación Orientada a Objetos (OOP).

Para Coad y Yourdon el acercamiento orientado a objetos encierra los conceptos de clases, objetos, la herencia.

Aunque Coad y la metodología de Yourdon son tal vez unas de las metodologías de OO más fáciles para aprender y familiarizarse, la queja más común es que es demasiado simple y no adecuado para proyectos grandes, por lo que se considera una metodología limitativa.

[7] Yourdon, E. y Coad, P., Análisis Orientado al objeto, Acantilados De Englewood, New Jersey: Prentice-Prentice-Hall, 1991.

3.4.3 Metodología Fusión por Derek Coleman.

Derek Coleman es principal tecnólogo y director de la ingeniería y de desarrollo del producto en RosettaNet, donde él se encuentra comisionado en la Unidad de Negocios Global del Software de Hewlett-Packard. Derek representa a Hewlett-Packard en la mesa de consultoría técnica de OASIS de sus siglas en Ingles (Organización de Consultoría en los Estándares de Información Estructurada).

En 1990, Derek Coleman de Hewlett-Packard dirigía un equipo de investigación para desarrollar un conjunto de necesidades para OOAD, El principal requerimiento era la obtención de una metodología simple mediante una notación efectiva.

El resultado fue la Fusión, que Coleman y otros desarrollaban, fue una adaptación de ideas de otras metodologías. Las cuales fueron incorporadas de ciertas ideas de Booch, Jacobson, Rumbaugh entre otros, así como el rechazo de muchas otras ideas de estas mismas metodologías.

La Fusión se basa en un conjunto pequeño pero comprensivo de las técnicas de diagramación "diagramming" bien definidas para describir pasos del análisis y del diseño.
Fusión consiste en tres fases: análisis, diseño y puesta en práctica. Fusión también proporciona las reglas para comprobar la consistencia de los modelos que se utilizan. Además, puede dar pautas para obtener una versión simplificada de la misma, ya que no todas las técnicas podrían ser útiles.

Coleman no usa algunos de los componentes de Rumbaugh, Shlaer y Mellor en la Fusión, porque los componentes no eran útiles en la práctica.

El acercamiento pragmático de Fusión parece tener potencial considerable para las aplicaciones de Cliente/servidor, pero esta metodología no está siendo vendiendo tan agresivamente como la mayor parte de otras metodologías.

3.4.4 (OOSE) Ingeniería de Software Orientado a Objetos por Jacobson.

Jacobson es considerado uno de los más experimentados en OO para aplicar éste a problemas de negocio tales como aplicaciones Cliente/Servidor.

La característica distintiva en Jacobson[8] es el "caso de uso" una definición de caso de uso consiste de un diagrama y una descripción de una interacción sencilla entre un actor y un sistema; en donde el actor puede ser un usuario final.

El gran peligro con OOSE es la suposición que se puede expresar todo en "casos de uso" lo que puede ser un poco complejo para algunos sistemas. Sin embargo, las descripciones de caso de uso y sus diagramas de interacción correspondientes pueden proveer una notación simple, e incluso para las aplicaciones de Cliente/Servidor, en los cuales los análisis de caso de uso son fáciles de interpretar por los usuarios finales quienes finalmente son los que brindan la autorización de la fase de análisis Orientada a Objetos.

3.4.5 (OMT) Técnica de Modelación orientada a Objetos por Rumbaugh.

La esencia del desarrollo en la Técnica de Modelación orientada a Objetos (OMT) es la identificación y la organización de los conceptos y dirigirlos a Objetos.

La metodología de Jaime Rumbaugh (1991), ofrece unas de las descripciones más completas. Aunque es carente del diseño Orientado a Objetos, contiene un gran número de ideas que integran una metodología orientada a Objetos.

De las principales metodologías, Rumbaugh soporta muchos diagramas tradicionales de metodologías estructuradas, y contiene una gran variedad en la notación en el momento de realizar una modelación.

La principal diferencia es el hecho de que el acercamiento orientado objeto no se basa en la descomposición funcional sino en describir los objetos verdaderos que desempeñan un papel en el mundo verdadero.

3.4.6 Análisis de Sistemas Orientado a Objetos por Shlaer y Mellor.

Shlaer y Mellor desarrollaron el método de Análisis de Sistemas Orientado a Objetos (OOSA). La técnica principal Shlaer y el uso de Mellor se llama Modelación de Información, el cual se utiliza para ayudar en la formalización del conocimiento.

[8] Jacobson, I. 1998. "Applying UML in The Unified Process" Presentación. Rational Software. Presentación disponible en http://www.rational.com/uml

El primer análisis de Shlaer y Mellor Orientado a Objetos salía en 1988, siendo éste unos de los ejemplos más tempranos de la metodología y ha evolucionado positivamente desde aquellos tiempos.

Desde OOSA, las pautas se han mejorado para repartir acciones en procesos con las características deseables. También, los diagramas de alto nivel se han introducido dando la ventaja de escalar ediciones referentes a proyectos grandes.

Shlaer y Mellor inicia con una modelación de información. Note que esto es más como un modelo de datos que un modelo de objetos. Después, un modelo de estado documenta los estados de objetos y las transiciones entre ellos. Finalmente, un diagrama de flujo de datos muestra el modelo de proceso.

A continuación mostraremos un cuadro comparativo referente a las metodologías Análisis y diseño orientado a objetos (OOAD) figura 3.1.

Metodología	Propietario	Tipo	Alcance	Etapa de Fortaleza	Aplicaciones
Booch	No	Ternario	Complejo, Amplio, Pragmático	Diseño	Todas
Coad y Yourdon	No	unitario	Simple, Amplio, Pragmático	Análisis	Cliente/Servidor
Fusion	No	Ternario	Complejo, Amplio, Pragmático	Ciclo de Vida Completo	Todas
Jacobson	Si	Ternario	Complejo, Amplio	Ciclo de Vida Completo	Todas
Rumbaugh OMT	No	Ternario	Complejo, Amplio, Relativamente pragmático	Análisis y Diseño	Todas, aunque compleja la implantar.
Shlaer y Mellor (OL)	No	Ternario	Complejo, Amplio	Diseño	Todas

Figura 3.1.- Comparación de Metodologías OOAD.

3.5 Modelamiento Orientado a Objetos.

Nos referimos a un modelo como la abstracción de algo, con el propósito de comprenderlo antes de construirlo.

Un modelo sirve para:

- Verificar una entidad física antes de su construcción.
- Comunicarnos con otros usuarios.
- Visualizar.
- Disminuir la complejidad.

Entendemos la Abstracción como un examen selectivo de ciertos aspectos del problema, eliminando los aspectos que no son importantes, de tal forma que este concepto, cuenta con ciertas características.

- La abstracción debe servir para un propósito que nos determina lo que no es importante, delimitando nuestro universo.
- Es posible obtener múltiples abstracciones de la misma cosa.
- Todas las abstracciones son incompletas e inadecuadas.
- Un buen modelo captura los aspectos cruciales del problema y omite el resto.

Las mayores deficiencias (y desafíos) en el proceso de desarrollo de software se encuentran en sus primeras fases. La distancia entre el espacio del problema (Universo de Discurso) y el espacio de la solución (producto software) hace necesario que la especificación de requisitos del sistema, resultante del proceso de modelado conceptual, tenga dos importantes propiedades: debe ser abstracta y declarativa.

- **Propiedad abstracta.** El modelo conceptual debe ser abstracto para facilitar la captación y modelado de aspectos del problema de una forma lo más cercana posible a los conceptos del dominio del problema.

- **propiedad declarativa.** Por otra parte, el modelo conceptual debe ser declarativo de manera que permita postergar decisiones de implementación. Esta característica nos separara de lo que denominamos enfoques orientados a objetos clásicos, en los que la especificación es principalmente imperativa y se limita en general a la estructura de los objetos y los perfiles de las operaciones.

La tarea de verificación del modelo exige disponer de algún formalismo preciso y a la vez con toda la capacidad expresiva necesaria para capturar todos los aspectos de interés del problema.

Las notaciones gráficas son atractivas pues facilitan el modelado y la legibilidad de una especificación de requisitos, pero pierden estas ventajas cuando se sobrecargan con detalles. Por otra parte, las representaciones puramente textuales, y más aún siendo formales, tienen el grado de detalle que se desee pero exigen más esfuerzo en su utilización. Esto sugiere que metodológicamente la mejor solución es una combinación de ambas técnicas: gráficas y textuales.

Como el objetivo es obtener un modelo completo y preciso del sistema, una representación textual final es la más adecuada, pues tanto las técnicas textuales como las gráficas pueden ser finalmente trasladadas a texto para describir el modelo en su totalidad.

3.5.1 Modelo Conceptual.

En la fase de Modelización Conceptual se construyen tres modelos del sistema:

- **Modelo de Objetos.**
- **Modelo Dinámico.**
- **Modelo Funcional.**

Estos tres modelos describen la sociedad de objetos desde tres puntos de vista complementarios.

3.5.1.1 Modelo de Objetos.

El Modelo de Objetos utiliza un Diagrama de Configuración de Clases (DCC) para definir y mostrar la estructura y comportamiento de todas las clases identificadas en el dominio del problema, así como sus relaciones. El DCC es un modelo semántico extendido, en el cual interactúan diversos elementos que a continuación menciono:

- **Clases.** Una clase se representa gráficamente como una caja dividida en tres áreas:
  - **Cabecera.** contiene la declaración del nombre de la clase.
  - **Parte Estática.** contiene la definición de los atributos que representan el estado de los objetos de la clase. Los atributos podrán ser constantes, variables y derivados. Aquellos atributos utilizados para identificar objetos se subrayan.
  - **Parte Dinámica.** contiene la declaración de los servicios de la clase. Cada servicio se declara especificando su nombre y argumentos (con sus tipos respectivos). Se distinguirá (gráficamente) entre los eventos de creación, borrado y los eventos compartidos con otras clases.

A continuación se representa una clase (figura 3.2).

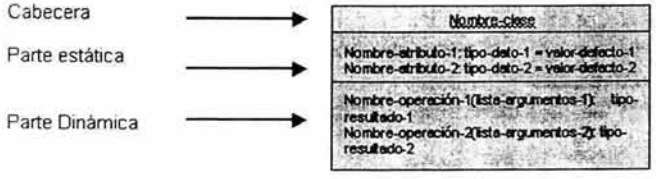


Figura 3.2.- Representación de una clase.

- **Acciones o Eventos.** Son servicios que un objeto puede activar (actuando como agente) para consultar o modificar el estado de otro objeto (figura 3.3).

Figura 3.3.- Acciones de un Objeto.

- **Relaciones.** Es el enlace o interacciones existentes entre objetos, las relaciones estructurales que podemos modelar son la agregación (parte-de) y la herencia (es-un), mismas que a continuación se enuncian:

  - **Relación de Agregación** entre clases, incluyendo información sobre cardinalidades (mínimas y máximas) que determinan cuántos objetos componentes forman parte de un objeto compuesto e inversamente, cuántos objetos compuestos pueden estar compuestos de un objeto en particular.

En la figura 3.4 se observa la relación de agregación a través del diagrama de configuración de clases.

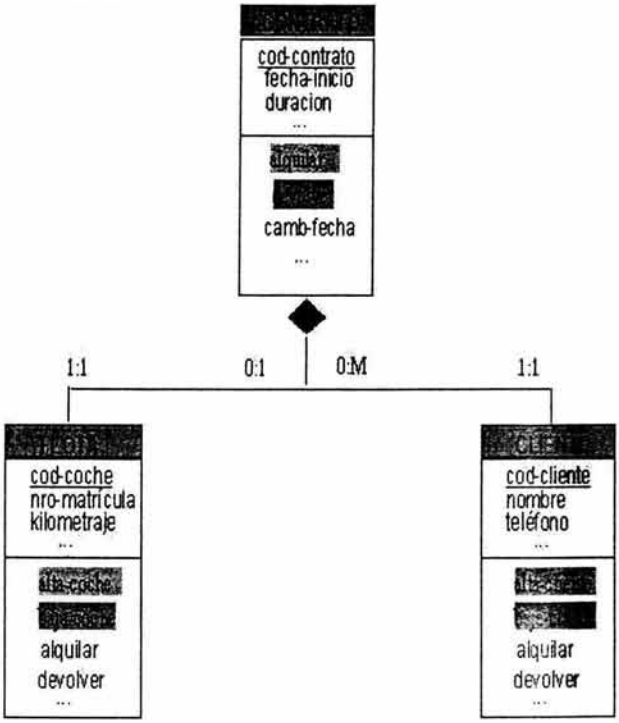


Figura 3.4.- Diagrama de configuración de clases (DCC), mostrando la relación "Agregación".

- **La Herencia.** se define gráficamente trazando una flecha desde la subclase hacia la superclase correspondiente. Esta flecha de especialización puede etiquetarse con una condición de especialización o con los eventos correspondientes de activación/cancelación de la especialización temporal (rol).

En la figura 3.5 se observa la relación de herencia a través del diagrama de configuración de clases.

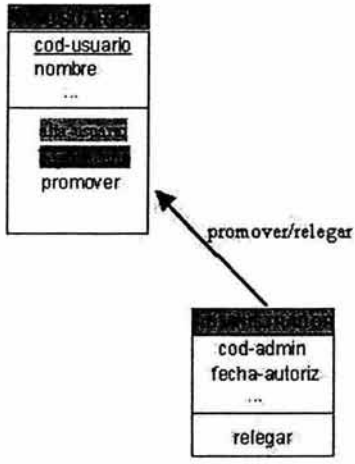


Figura 3.5.- Diagrama de configuración de clases (DCC), mostrando la relación "Herencia".

En el modelo de objetos, una clase podrá estar definida por un conjunto de clases que cubren el resto de sus propiedades figura 3.6.

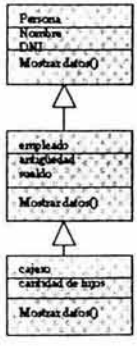


Figura 3.6.- definición de una clase, mediante otras.

Las principales características del modelo de objetos son:

- Describe la estructura de los objetos en un sistema (identidad, relaciones con otros objetos, atributos y operaciones).

- Marco en el que podemos ubicar los Modelos Dinámico y Funcional.

- Los objetos son las unidades en las que dividiremos el mundo y que serán las moléculas de los distintos modelos.

- Se representa de forma gráfica mediante diagramas de objetos que contendrán clases de objetos.

- Las clases se disponen en jerarquías que comparten estructura y comportamientos comunes, y que están asociadas con otras clases.

Finalmente mostraremos un Diagrama de configuración de clases (figura 3.7).

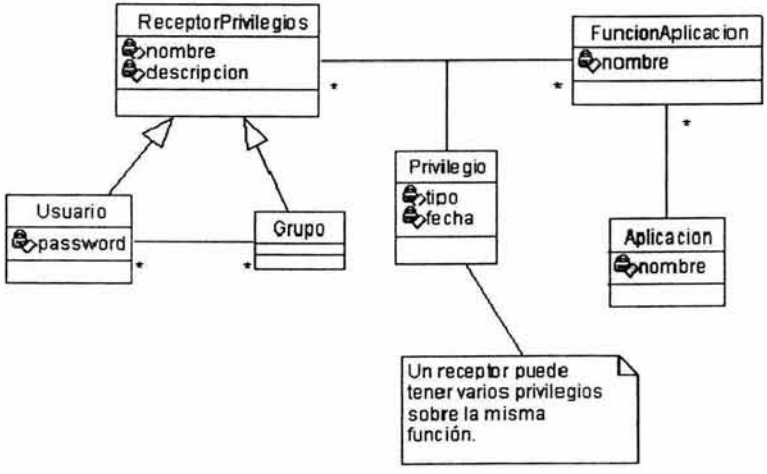


Figura 3.7.- Diagrama de configuración de clases.

3.5.1.2 Modelo Dinámico.

En el Modelo Dinámico se representan aspectos relacionados con las secuencias posibles de eventos (vidas posibles) y la interacción entre objetos.

Las principales características del modelo dinámico son:

- Describe los aspectos del sistema que tienen que ver con el tiempo y la secuencia de las operaciones: Sucesos (eventos), Estados y Acciones.
- Captura el control del sistema.
- Se representa gráficamente mediante diagrama de estados.

Para representar estos aspectos, tenemos dos tipos de diagramas:

- Diagrama de Transición de Estados (DTE).
- Diagrama de Interacción de Objetos (DIO).

3.5.1.2.1 Diagrama de Transición de Estados (DTE).

Los DTE se utilizan para describir el comportamiento de los objetos estableciendo las sus vidas posibles. Una vida válida de un objeto, es una secuencia de eventos que caracteriza un comportamiento correcto para todos los objetos de la clase.

En este contexto, los estados del DTE denotan situaciones en las que pueden encontrarse los objetos durante su existencia como consecuencia de la ocurrencia de eventos relevantes. Las transiciones representan cambios de estado permitidos que pueden restringirse introduciendo condiciones.

La especificación de una transición posee la siguiente sintaxis:

evento | acción | transacción [if precondición] [when condición-de-control]

Una precondición es una condición definida sobre atributos del objeto que debe satisfacerse en el estado de partida para que el servicio pueda ocurrir.

Una condición-de-control es una condición que sirve para evitar el posible no-determinismo entre dos o más transiciones que partiendo del mismo estado van a estados diferentes, estando etiquetadas con la misma acción.

En la figura 3.8 y 3.9 mostraremos dos diagramas de transición de estados.

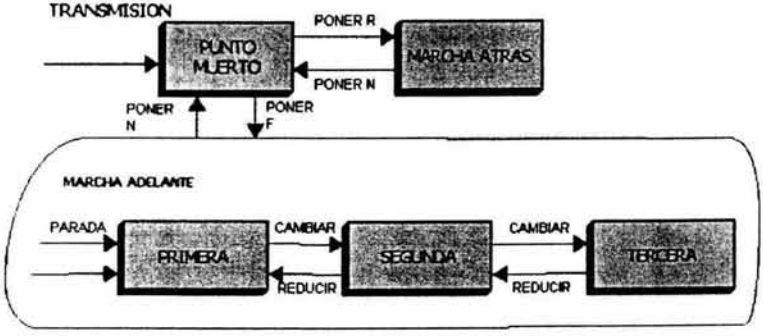


Figura 3.8.- Diagrama de transición de estados.

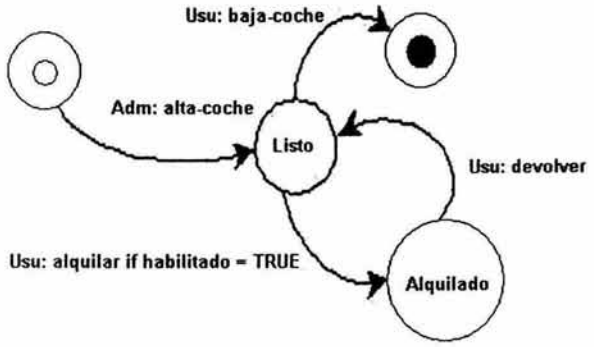


Figura 3.9.- Diagrama de transición de estados.

3.5.1.2.2 Diagrama de Interacción de Objetos (DIO).

La interacción entre objetos se modela gráficamente mediante un Diagrama de Interacción entre Objetos (DIO). En el DIO podemos especificar dos interacciones básicas:

- **Disparos.** Son servicios de una clase que se activan de forma automática cuando se satisface una condición en un objeto dicha clase.

Los disparos se definirán dibujando una flecha que vaya de la cabecera de una clase a la acción disparada y etiquetada con la condición de disparo (figura 3.10).

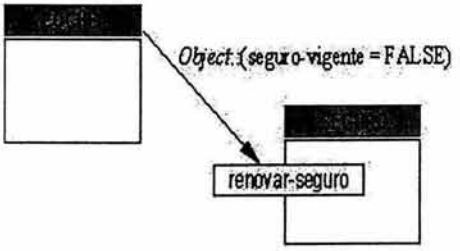


Figura 3.10.- representación de un "Disparo" en un diagrama de interacción.

- **Interacciones Globales.** son transacciones compuestas por servicios de clases diferentes.

Las interacciones globales se introducen conectando los servicios que componen la interacción y nominándola mediante un identificador de interacción global común.

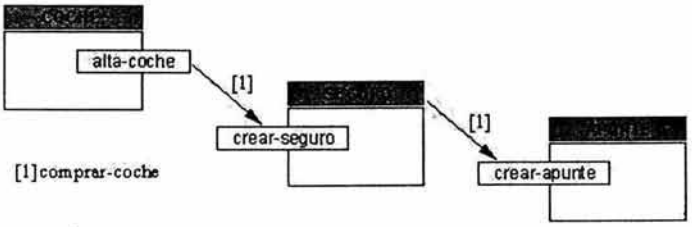


Figura 3.11.- representación de "Interacciones Globales" en un diagrama de interacción.

3.5.1.3 Modelo Funcional.

El propósito del Modelo Funcional es capturar la semántica asociada a los cambios de estado de forma fácil e intuitiva.

En este modelo especificaremos mediante un diálogo interactivo el efecto de un evento sobre los atributos. El valor de cada atributo se modificará dependiendo de la acción ocurrida, de los argumentos del evento y del estado actual del objeto.

Las principales características del modelo funcional son:

- Describe los aspectos del sistema que tienen que ver con el tiempo y la secuencia de Describe los aspectos que tienen que ver con la transformación de los valores: funciones, correspondencias, Restricciones y dependencias funcionales.

- Captura lo que hace el sistema, sin preocuparse del como ni del cuando.

- Se representa mediante diagramas de flujo de datos.

El Método Orientado a Objetos proporciona un modelo mediante el cual el analista sólo tiene que categorizar cada atributo de entre un conjunto predefinido de tres categorías. Estas categorías determinan qué información se necesita para determinar cómo cambia el valor del atributo ante la ocurrencia de determinados eventos.

Las tres categorías de atributos son:

- **Cardinales.** Sus eventos relevantes incrementan o decrementan su valor en una determinada cantidad.

- **Independientes del estado.** tienen un valor que sólo depende de la última acción ocurrida. Una vez ha ocurrido una acción relevante el nuevo valor del atributo es independiente del valor que tenía antes.

- **De situación.** Mediante la activación de una acción portadora se le asigna al atributo un valor de un dominio discreto.

La información que se obtiene al finalizar la construcción de los tres modelos constituye la especificación del sistema.

3.5.2 Relación entre Modelos.

Cada uno de los modelos describe un aspecto del sistema manteniendo referencias al resto de los modelos, sin embargo es necesario establecer los límites de operación de cada uno.

El Modelo de Objetos define las estructuras de datos sobre las que actuarán el Modelo Dinámico y el Modelo Funcional. Los sucesos serán las operaciones de los objetos definidas en el Modelo de Objetos.

Puede existir una eventual ambigüedad acerca de qué modelo debe contener ciertos elementos de información.

Puede ocurrir que ciertas propiedades del sistema no queden bien expresadas por ningún modelo, en cuyo caso el lenguaje natural será la mejor alternativa.

3.5.3 Modelo de Ejecución.

Una vez especificado el sistema, el modelo de ejecución establece la creación de un diccionario de datos.

Diccionario de datos. Contiene las características lógicas de los sitios donde se almacenan los datos del sistema, incluyendo nombre, descripción, alias, contenido y organización. Identifica los procesos donde se emplean los datos y los sitios donde se necesita el acceso inmediato a la información, se desarrolla durante el análisis de flujo de datos y auxilia a los analistas que participan en la determinación de los requerimientos del sistema, su contenido también se emplea durante el diseño.

Razones para su utilización:

- Para manejar los detalles en sistemas muy grandes, ya que tienen enormes cantidades de datos, aun en los sistemas más chicos hay gran cantidad de datos.

- Los sistemas al sufrir cambios continuos, es muy difícil manejar todos los detalles. Por eso se registra la información, ya sea sobre hoja de papel o usando procesadores de texto.

Los analistas mas organizados usan el diccionario de datos automatizados diseñados específicamente para el análisis y diseño de software.

En la figura 3.12 mostramos un ejemplo del diccionario de datos.

Nombre	Descripción	Estructura
Código del Producto	Identifica de forma única el producto por presentación.	Código del Producto = Base + Adicción + Ojo
Ensamblajes	Reporte de los ensamblajes que se realizaron.	Ensamblajes = Nombre del Producto + Código del Producto + Fecha producción + Cantidad Ensamblada
Error en Figuras	Se recibió información cuantitativa del producto que se entiende no está correcta.	Error en Productos = Nombre del Reporte + Código del Producto + Explicación del Error + Fecha
Error en Pedidos	Explicación de errores encontrados en pedidos de productos.	Error en Pedidos = #Orden + Explicación del Error + Fecha
Error en Productos	Se recibió información descriptiva del producto que se entiende no está correcta.	Error en Productos = Nombre del Reporte + Código del Producto + Explicación del Error +

Figura 3.12.- Diccionario de datos.

La orientación a objetos es un nuevo paradigma para el diseño de aplicaciones informáticas que permite reducir la complejidad de forma segura y robusta. Este nuevo paradigma de diseño puede aplicarse de manera natural al diseño de sistemas en general.

Los diagramas gráficos del modelado orientado a objetos ayudan en la toma de decisiones durante el diseño porque representan de manera sencilla y compacta los elementos esenciales de la navegación.

El modelo dinámico describe los aspectos de comportamiento de un sistema que cambia con el tiempo y es de utilidad para implementar los aspectos de control, es decir aquella

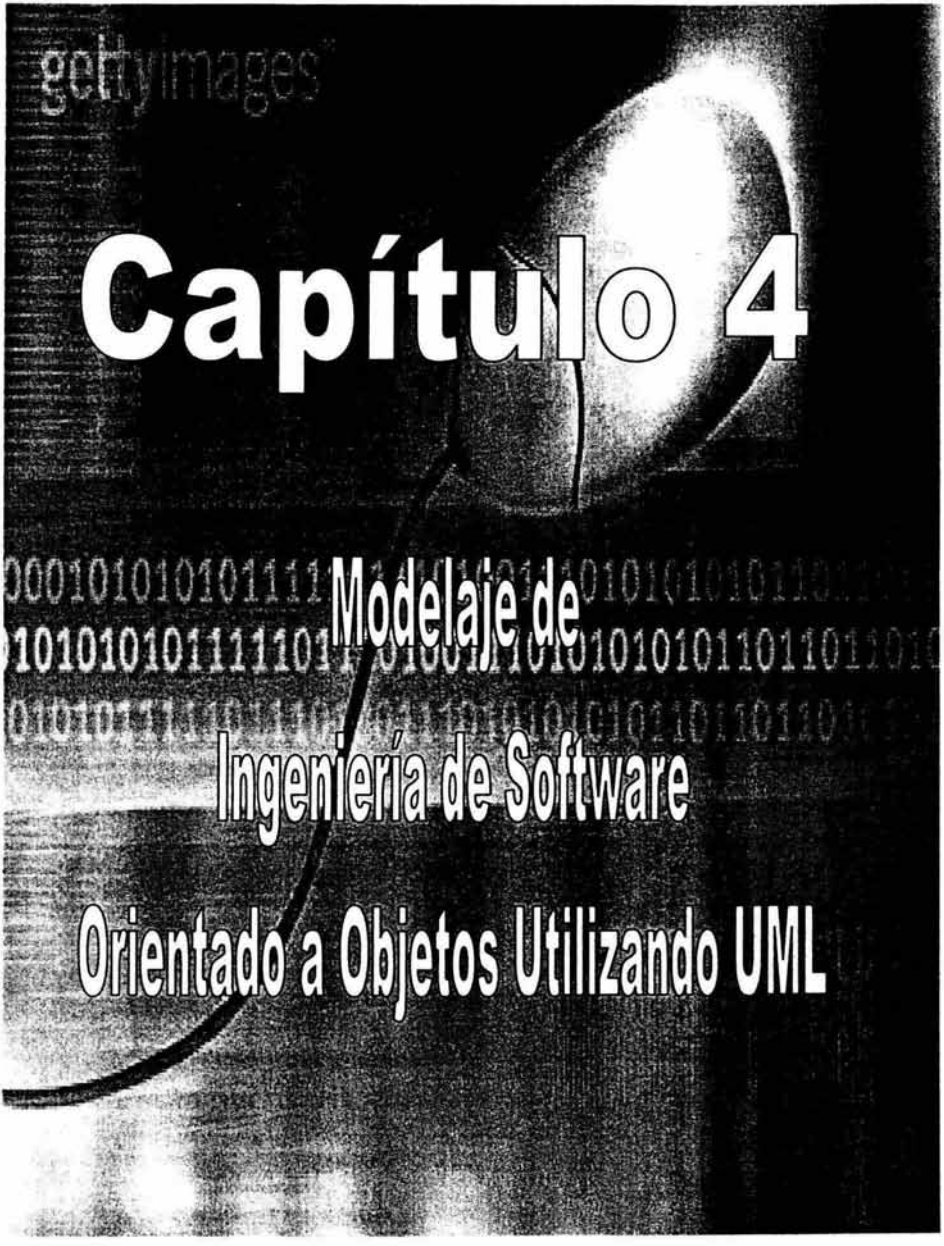
parte del sistema que describe las secuencias de operaciones que se producen en respuesta a estímulos externos.

La importancia del modelo dinámico dependerá de la naturaleza de la aplicación, así por ejemplo, dicho modelo no será preponderante para un depósito de datos puramente estático cómo una base datos, adquiriendo mayor importancia para sistemas interactivos.

El modelo dinámico constituye un soporte muy útil para el paso a la fase de diseño, ya que permite describir los encadenamientos de los métodos.
Los sucesos que representamos en los diagramas de estado correspondiente a un objeto, suelen implementarse como operaciones para ése objeto en el modelo de objetos. Según algunos autores, como James Rumbaugh, los sucesos son más expresivos que las operaciones, ya que el efecto de un suceso depende tanto de la clase del objeto cómo de su estado.

El proceso orientado a objetos según Booch sigue vigente y sus principios son base de algunos de los nuevos procesos ya que podemos exponer la hipótesis de que más que un ciclo de vida orientado a objetos, existe ciclos de vidas que son más o menos aptos para la aplicación de metodologías orientadas a objetos.

En el siguiente capítulo abordaremos el tema de UML en donde se observa que éste lenguaje integra lo mejor de los modelos de IS con una tendencia orientada a objetos.


gettyimages
Capítulo 4
Modelaje de
Ingeniería de Software
Orientado a Objetos Utilizando UML

4.1 Introducción.

Desde mediados de los años ochenta se ha venido imponiendo gradualmente la tecnología de objetos en el ámbito de la producción de software, aplicándose primero el enfoque de la orientación a objetos en la construcción de programas, e incorporándose después en el resto de actividades del ciclo de vida del software, como la planificación, el análisis o el diseño.

Esta tecnología permite aumentar la productividad y mejorar la calidad del software, constituyendo las metodologías orientadas a objetos, un marco idóneo para abordar la complejidad creciente en el desarrollo de software, gracias a sus modelos más realistas y consistentes, dando lugar a productos más flexibles y fáciles de mantener y reutilizar.

Actualmente existe una multitud de métodos de desarrollo orientados a objetos, con la característica común de basarse en los mismos conceptos fundamentales de esta tecnología, pero con diferencias en la notación utilizada en los modelos de análisis y diseño propuestos y en la forma de introducir la orientación a objetos en el ciclo de vida del software.

Las razones del cambio desde el enfoque estructurado al enfoque orientado a objetos (OO) se refieren principalmente a la mejora de la productividad, de la calidad y del mantenimiento del software que ofrece esta nueva tecnología, y su mejor adaptación a las nuevas características de las aplicaciones informáticas que en día se requiere, tanto para el procesamiento distribuido basado en la filosofía cliente/servidor, como en las interfaces gráficas de usuario (ambientes web).

Otro aspecto del enfoque OO que lo hace adecuado para garantizar la calidad de los productos si se aplica desde las primeras fases de los proyectos es su capacidad para modelar los requisitos de los usuarios de una forma muy intuitiva. En este sentido, la gran ventaja que está suponiendo la OO como paradigma de ingeniería del software es precisamente que este enfoque se aplica, tanto en las etapas de análisis y diseño, como en la construcción o programación del software, convirtiéndose la etapa del desarrollo en un proceso de depuración de los objetos que fueron conceptualizados durante el análisis.

En cuanto a las metodologías OO existentes en la actualidad, aunque el número es muy elevado, sólo unas pocas aportan alguna novedad significativa respecto a las otras, el resto se derivan de éstas. En este sentido, hay que resaltar el gran esfuerzo de unificación

de las notaciones utilizadas en los métodos con mayor implantación del mercado, que ha dado origen al denominado Lenguaje de Modelado Unificado (UML)[9] en 1997.

El lenguaje UML tiene una notación gráfica muy expresiva que permite representar en mayor o menor medida todas las fases de un proyecto informático: desde el análisis con los casos de uso, el diseño con los diagramas de clases, objetos, etc., hasta la implementación y configuración con los diagramas de despliegue.

4.2 Historia de UML.

El lenguaje UML[10] comenzó a gestarse en octubre de 1994, cuando Rumbaugh se unió a la compañía "Rational Software Corporation" fundada por Booch (dos reputados investigadores en el área de metodología del software). El objetivo de ambos era unificar dos métodos que habían desarrollado: el método Booch y el OMT (Object Modelling Tool). En la figura 4.1 observaremos el logo de UML.

Figura 4.1.- Logo de UML.

El primer borrador apareció en octubre de 1995. En esa misma época otro reputado investigador, Jacobson, se unió a Rational y se incluyeron ideas suyas. Estas tres personas son conocidas como los "tres amigos". Además, este lenguaje se abrió a la colaboración de otras empresas para que aportaran sus ideas. Todas estas colaboraciones condujeron a la definición de la primera versión de UML.

[9] Unified Modeling Language Documentation. UML Resource Center (1999). http://www.rational.com/uml/resources/documentation.
[10] G. Booch, J. Rumbaugh y I. Jacobson, "El Lenguaje Unificado de Modelado", Addison Wesley, Madrid 1999.

En la figura 4.2 se muestra de manera grafica la evolución de UML.

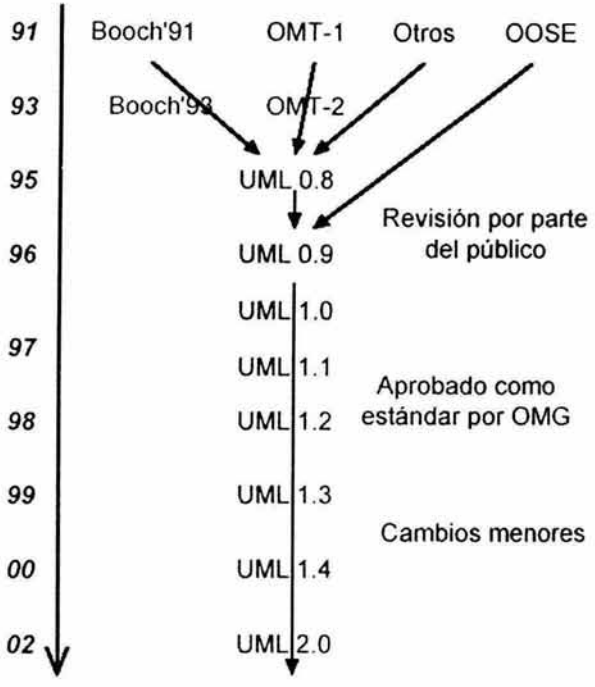


Figura 4.2.- Evolución de UML.

Esta primera versión se ofreció a un grupo de trabajo para convertirlo en 1997 en un estándar del OMG (Object Management Group http://www.omg.org).

Este grupo que gestiona estándares relacionados con la tecnología orientada a objetos (metodologías, bases de datos orientadas a objetos, CORBA, etc.), propuso una serie de modificaciones y una nueva versión de UML (la versión 1.1), que fue adoptada por el OMG como estándar en noviembre de 1997. Desde aquella versión han habido varias revisiones que gestiona la OMG bajo su "Revision Task Force" (RTF) quien en realidad es un conjunto de personas cuya responsabilidad es la de generar revisiones menores de la

especificación UML. La última versión aprobada es la 1.5., posteriormente se desarrolló una nueva versión en la que se incluyen cambios importantes (principalmente añadir nuevos diagramas) que conducen a la versión 2.0 planificada originalmente para fines del 2002, sin embargo, la últimas noticias del 18 de noviembre del 2003 nos informan que "IBM Software Group" adquiere "Rational Software Corporation", razón por la cual IBM planeo liberar la versión 2.0 de UML en Abril del 2004, haciéndolo en forma oficial hasta el 7 de Mayo de ese mismo año.

Rational Suite ha sido actualizado para permitir a los clientes crear, ejecutar, garantizar la calidad e implantar software en plataformas adicionales y en múltiples lenguajes y tecnologías, como WebSphere Studio, Eclipse, Linux, DB2®, Microsoft® Visual C++® y .NET, EMC® VMWare®, Sybase® PowerBuilder®, Oracle® Forms, y Borland® Delphi(tm).

Al observar la evolución que ha tenido UML, podemos deducir que la fuerza de la cual se le ha dotado es enorme, ya que se ha convertido en un pilar y líder en el dictamen de la modelación de sistemas complejos.

4.3 Modelado Visual.

Tal como indica su nombre, UML es un lenguaje de modelado unificado, partiendo del hecho de que un modelo es una simplificación de la realidad. El objetivo del modelado de un sistema es capturar las partes esenciales del mismo, de tal forma que para facilitar este modelado, se realiza una abstracción y se plasma bajo una notación gráfica. A esto se conoce como modelado visual.

El modelado visual permite manejar la complejidad de los sistemas a analizar o diseñar. De la misma forma que para construir una choza no hace falta un modelo, cuando se intenta construir un sistema complejo como un rascacielos, es necesario abstraer la complejidad en modelos que el ser humano pueda entender.

UML sirve para el modelado completo de sistemas complejos, tanto en el diseño de los sistemas software como para la arquitectura de hardware donde se ejecuten.

Otro objetivo de este modelado visual es que sea independiente del lenguaje de implementación (programación), de tal forma que los diseños realizados usando UML se pueda implementar en cualquier lenguaje que soporte las posibilidades de UML (principalmente lenguajes orientados a objetos, más no necesariamente).

UML es además un método formal de modelado. Esto aporta las siguientes ventajas:

- Mayor rigor en la especificación.
- Permite realizar una verificación y validación del modelo realizado.
- Se pueden automatizar determinados procesos y permite generar código a partir de los modelos y a la inversa (a partir del código fuente generar los modelos). Esto permite que el modelo y el código estén actualizados, con lo que siempre se puede mantener la visión en el diseño, de más alto nivel, de la estructura de un proyecto.

4.3.1 Objetivos de UML.

Los objetivos de UML son muchos, pero se pueden sintetizar en:

- **Visualizar.** UML permite expresar de una forma gráfica un sistema de forma que otro lo puede entender.
- **Especificar.** UML permite especificar cuáles son las características de un sistema antes de su construcción.
- **Construir.** A partir de los modelos especificados se pueden construir los sistemas diseñados.
- **Documentar.** Los propios elementos gráficos sirven como documentación del sistema desarrollado que pueden servir para su futura revisión.

Aunque UML está pensado para modelar sistemas complejos, el lenguaje es los suficientemente expresivo como para modelar sistemas que no son informáticos, como flujos de trabajo (*workflow*) en una empresa, diseño de la estructura de una organización y por supuesto, en el diseño de hardware.

4.4 Metodología "Proceso unificado de desarrollo".

Aunque UML es bastante independiente del proceso de desarrollo que se siga, los mismos creadores de UML han propuesto su propia metodología de desarrollo, denominada el Proceso Unificado de Desarrollo[11].

El Proceso Unificado, es un conjunto de etapas de desarrollo de software flexible, el cual se adapta a proyectos variados en tamaños y complejidad. Se ha basado en muchos años de experiencia del uso de la tecnología orientada a objetos en el desarrollo de software de misión crítica, fundada por "los tres amigos" o los tres grandes de la metodología orientada a objetos: Grady Booch, James Rumbaugh e Ivar Jacobson.

El Proceso Unificado guía a los equipos de proyecto en cómo administrar el desarrollo iterativo de un modo controlado mientras se balancean los requerimientos del negocio, el tiempo al mercado y los riesgos del proyecto.

El proceso describe los diversos pasos involucrados en la captura de los requerimientos y en el establecimiento de una guía arquitectónica lo más pronto para diseñar y probar el sistema hecho de acuerdo a los requerimientos y a la arquitectura.

El proceso describe qué entregables producir, cómo desarrollarlos y también provee patrones. Este proceso es soportado por herramientas que automatizan entre otras cosas, el modelado visual, la administración de cambios y las pruebas.

La metodología del Proceso Unificado de Desarrollo ha adoptado un enfoque que se caracteriza por:

- Interacción con el usuario continúa desde un inicio.
- Disminución de riesgos antes de que ocurran.
- Liberaciones frecuentes, de diversos documentos, dependiendo de la fase en la que se opere.
- Aseguramiento de la calidad.
- Involucramiento del equipo en todas las decisiones del proyecto.

[11] I. Jacobson, G. Booch, J. Rumbaugh , "El Proceso Unificado de Desarrollo", Addision Wesley, 2000

- Anticiparse al cambio de requerimientos.

Así pues esta metodología utiliza el UML para expresar gráficamente todos los esquemas de un sistema de software, sin embargo, las principales características que definen a un proceso unificado nos indica que éste debe ser:

- **Dirigido por casos de uso.** Basándose en los casos de uso, los desarrolladores crean una serie de modelos de diseño e implementación que los llevan a cabo. Además, estos modelos se validan para que sean conformes a los casos de uso. Finalmente, los casos de uso también sirven para realizar las pruebas sobre los componentes desarrollados.

- **Iterativo e incremental.** Todo sistema informático complejo supone un gran esfuerzo que puede durar desde varios meses hasta años. Por lo tanto, lo más práctico es dividir un proyecto en varias fases. Actualmente se suele hablar de ciclos de vida en los que se realizan varios recorridos por todas las fases. Cada recorrido por las fases se denomina iteraciones. Además, cada iteración parte de iteración anterior incrementado o revisando la funcionalidad implementada.

En la figura 4.3 se observa el proceso iterativo e incremental.

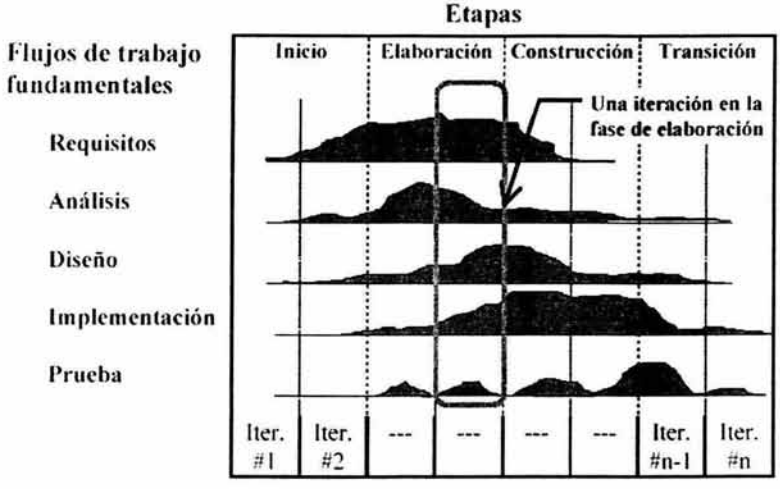


Figura 4.3.- Proceso iterativo e incremental.

- **Centrado en la arquitectura**[12]. El Proceso Unificado se enfoca en la arquitectura como el centro del desarrollo para asegurar que el desarrollo basado en componentes sea punto clave para un alto nivel de reutilización de código. Una vista arquitectónica es "una descripción simplificada (una abstracción) de un sistema desde un punto de vista, que cubre particularidades y omite entidades que no son relevantes a esta perspectiva".

[12] E. Hernández, J. Hernández, C. Lizandra, "C++ Estandar", ITP Paraninfo 2001.

4.5 Diagramas de UML.

Una de las formas más sencillas de modelar un sistema es sin duda la utilización de medios gráficos, como son los diagramas. Un diagrama ofrece una vista del sistema a modelar. Para poder representar correctamente un sistema. UML ofrece una amplia variedad de diagramas para visualizar el sistema desde varias perspectivas.

UML incluye los siguientes diagramas:

- Diagrama de casos de uso.
- Diagrama de clases.
- Diagrama de secuencia.
- Diagrama de colaboración.
- Diagrama de estados.

4.5.1 Elementos Comunes a Todos los Diagramas.

A fin de entender mejor los diagramas, explicare los elementos comunes a todos los diagramas.

- **Notas.** Una nota sirve para añadir cualquier tipo de comentario a un diagrama o a un elemento de un diagrama. Es un modo de indicar información en un formato libre, cuando la notación del diagrama en cuestión no nos permite expresar dicha información de manera adecuada. Una nota se representa como un rectángulo con una esquina doblada con texto en su interior. Puede aparecer en un diagrama tanto sola, como unida a un elemento por medio de una línea discontinua. Puede contener restricciones, comentarios, el cuerpo de un procedimiento, etc.

A Continuación observamos un diagrama en donde aparece una nota dirigida hacia un usuario (ver figura 4.4).

Figura 4.4.- Diagrama con una Nota.

- **Navegabilidad.** Se trata de un concepto de diseño que esta implícito en las relaciones existentes entre los diferentes elementos de un diagrama (objetos o clases) el cual se representa mediante una flecha. Significa que es posible "navegar" desde el objeto o clase origen hasta otro objeto o clase destino.

- **Objetos.** Un objeto se representa de la misma forma que una clase. En el área superior aparecen el nombre del objeto junto con el nombre de la clase subrayados, según la siguiente sintaxis: nombre_del_objeto: nombre_de_la_clase Puede representarse un objeto sin un nombre específico, entonces sólo aparece el nombre de la clase.

En la figura 4.5 observamos la representación de diversos objetos.

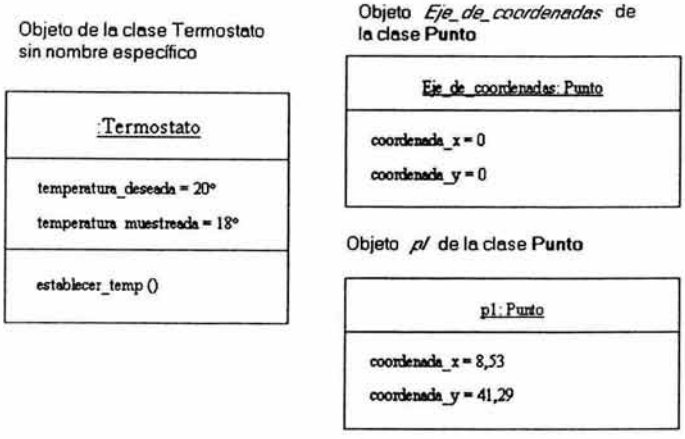


Figura 4.5.- Representación de un objeto.

- **Asociaciones.** Las asociaciones entre dos clases se representan mediante una línea que las une. La línea puede tener una serie de elementos gráficos que expresan características particulares de la asociación. A continuación se verán los más importantes de entre dichos elementos gráficos.

  El nombre de la asociación es opcional y se muestra como un texto que está próximo a la línea. Se puede añadir un pequeño triángulo negro sólido que indique la dirección en la cual leer el nombre de la asociación. En el ejemplo de la figura 4.6 se puede leer la asociación como "Director manda sobre Empleado".

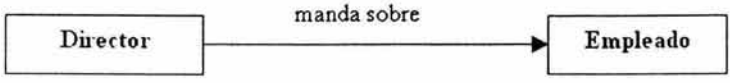


Figura 4.6.- Asociación entre elementos.

Los nombres de las asociaciones normalmente se incluyen en los modelos para aumentar la legibilidad. Sin embargo, en ocasiones pueden hacer demasiado abundante la información que se presenta, con el consiguiente riesgo de saturación. En ese caso se puede suprimir el nombre de las asociaciones consideradas como suficientemente conocidas. En las asociaciones de tipo agregación y de herencia no se suele poner el nombre.

- **Roles.** Para indicar el papel que juega una clase en una asociación se puede especificar un nombre de rol. Se representa en el extremo de la asociación junto a la clase que desempeña dicho rol.

  En la figura 4.7, en la cual se observa que una empresa tiene el rol de "contratante" y el trabajador de "Empleado". Lo mismo ocurre en el caso de una persona, ya que ésta puede tener el rol de "padre" o "hijo"

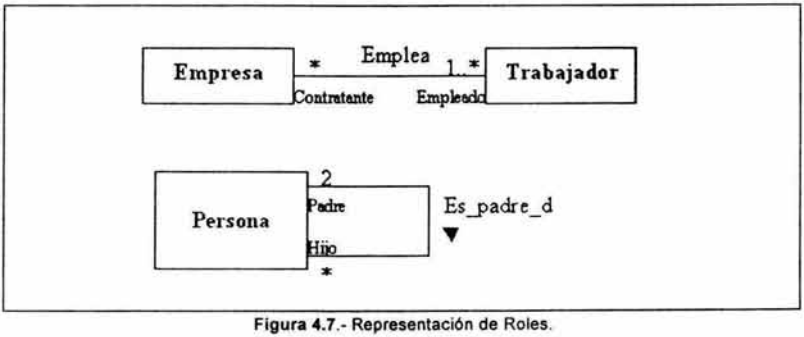


Figura 4.7.- Representación de Roles.

4.5.2 Diagrama Casos de uso.

El diagrama de casos de uso representa la forma en como un Cliente (Actor) opera con el sistema en desarrollo, además de la forma, tipo y orden en como los elementos interactúan (operaciones o casos de uso).

Un diagrama de casos de uso consta de los siguientes elementos: Actor, Casos de Uso y relaciones de Uso, mismos que a continuación se explican:

4.5.2.1 Actor.

Una definición previa, es que un Actor es un rol que un usuario juega con respecto al sistema. Es importante destacar el uso de la palabra rol, pues con esto se especifica que un Actor no necesariamente representa a una persona en particular, sino más bien la labor que realiza frente al sistema (figura 4.8).

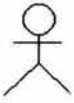


Figura 4.8.-Representación de un Actor.

Como ejemplo a la definición anterior, tenemos el caso de un sistema de ventas en que el rol de Vendedor con respecto al sistema puede ser realizado por un Vendedor o bien por el Jefe de Local.

4.5.2.2 Caso de Uso.

Es una operación/tarea específica que se realiza tras una orden de algún agente externo, sea desde una petición de un actor o bien desde la invocación desde otro caso de uso (figura 4.9).

Figura 4.9.- Representación de un Caso de uso.

4.5.2.3 Relaciones de Uso.

Un caso de uso, en principio, describe una tarea que tiene un sentido completo para el usuario, sin embargo, dado que (en su mayoría) un caso de uso interactúa con otros casos de uso, es útil describir la manera en la cual se relacionan éstos. las relaciones existentes con de tres tipos:

- **Asociación** ——→. Es el tipo de relación más básica que indica la invocación desde un actor o caso de uso a otra operación (caso de uso). Dicha relación se denota con una flecha simple.

- **Dependencia o Instanciación** ----------> . Es una forma muy particular de relación entre clases, en la cual una clase depende de otra, es decir, se instancia (se crea). Dicha relación se denota con una flecha punteada.

- **Generalización** ——▷ . Este tipo de relación es uno de los más utilizados, cumple una doble función dependiendo de su estereotipo, que puede ser de Uso (<<uses>>) o de Herencia (<<extends>>).

  Este tipo de relación esta orientado exclusivamente para casos de uso (y no para actores).

  - **Uso (<<uses>>).** Se recomienda utilizar cuando se tiene un conjunto de características que son similares en más de un caso de uso y no se desea mantener copiada la descripción de la característica.

    En la figura 4.10 se muestra cómo el caso de uso "Realizar Reintegro" utiliza el caso de uso "Proceso de autorización".

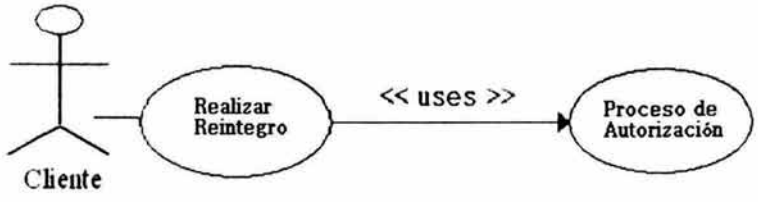


Figura 4.10.- Caso de uso "Realizar Reintegro", en éste utiliza a otro caso de uso.

  - **Herencia (<<Extends>>).** Se recomienda utilizar cuando un caso de uso es similar a otro (características).

    En la figura 4.11 se muestra como el caso de uso Comprar Producto permite explícitamente extensiones en el siguiente punto de extensión: "info regalo".

La interacción correspondiente a establecer los detalles sobre un producto que se envía como regalo están descritos en el caso de uso "Detalles Regalo".

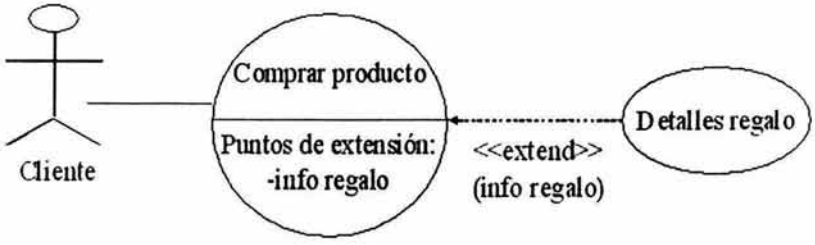


Figura 4.11.- Caso de uso Comprar con extensión a otro caso de uso.

De lo anterior cabe mencionar que tiene el mismo paradigma en diseño y modelamiento de clases, en donde esta la duda clásica de usar o heredar.

A continuación veremos un ejemplo en donde se utilice la totalidad de los elementos involucrados en los diagramas de casos de uso, para ello utilizaremos el caso de una Máquina Recicladora.

La intención es modelar un sistema que controla una máquina de reciclamiento de botellas, tarros y jabas. El sistema debe controlar y/o aceptar:

- Registrar el número de artículos "item (s)" ingresados.
- Imprimir un recibo cuando el usuario lo solicita:
  - Describe lo depositado.
  - El valor de cada artículo.
  - Total.
- El usuario/cliente presiona el botón de comienzo.

- Existe un operador que desea saber lo siguiente:
  - Cuantos artículos han sido retornados en el día.
  - Al final de cada día el operador solicita un resumen de todo lo depositado en el día.
- El operador debe además poder cambiar:
  - Información asociada a artículos.
  - Dar una alarma en el caso de que:
    - Artículo se atora.
    - No hay más papel.

Como una primera aproximación identificamos a los actores que interactúan con el sistema (figura 4.12):

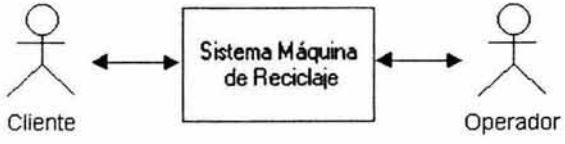


Figura 4.12.- Primera aproximación para el Caso de uso de una máquina recicladora.

Luego, tenemos que un Cliente puede Depositar Artículos y un Operador puede cambiar la información de un artículo o bien puede Imprimir un informe (figura 4.13):

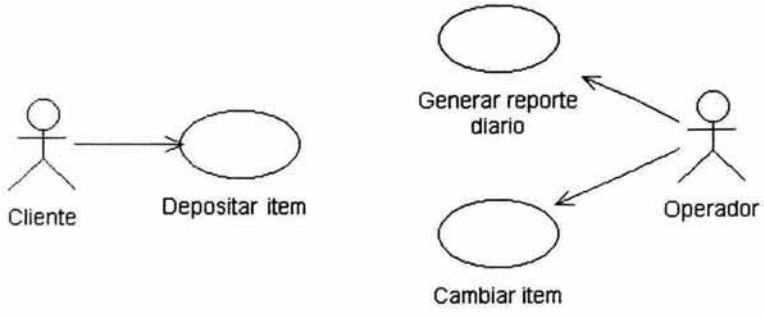


Figura 4.13.- Caso de uso para describir las acciones que podrá hacer un cliente y un operador.

Además podemos notar que un artículo puede ser una Botella, un Tarro o una Jaba (figura 4.14).

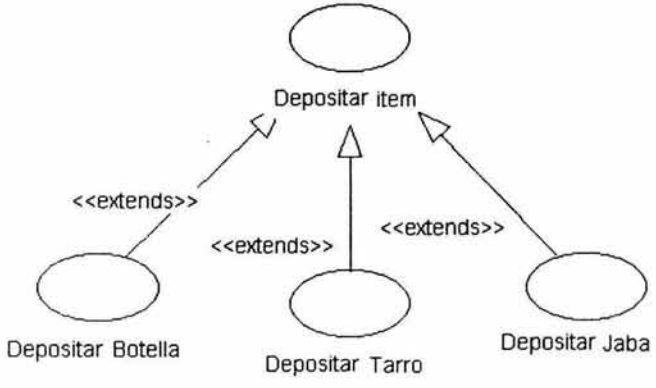


Figura 4.14.- Caso de uso en donde se describe la herencia (<<extends>>) de la acción "depositar artículo", para una Botella, Tarro y Jaba.

Otro aspecto es la impresión de comprobantes, que puede ser realizada después de depositar algún artículo por un cliente o bien puede ser realizada a petición de un operador (figura 4.15).

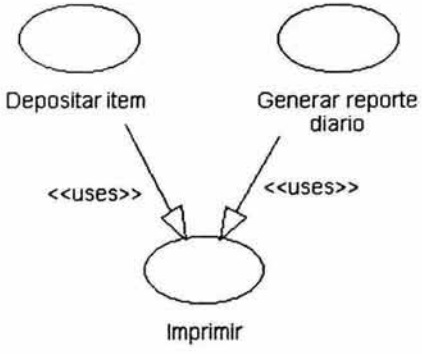


Figura 4.15.- Caso de uso en donde se describe que la acción "Imprimir" utiliza los casos de uso: Depositar artículo y Generar reporte diario

Finalmente el diseño completo del diagrama Casos de Uso quedaría (figura 4.16):

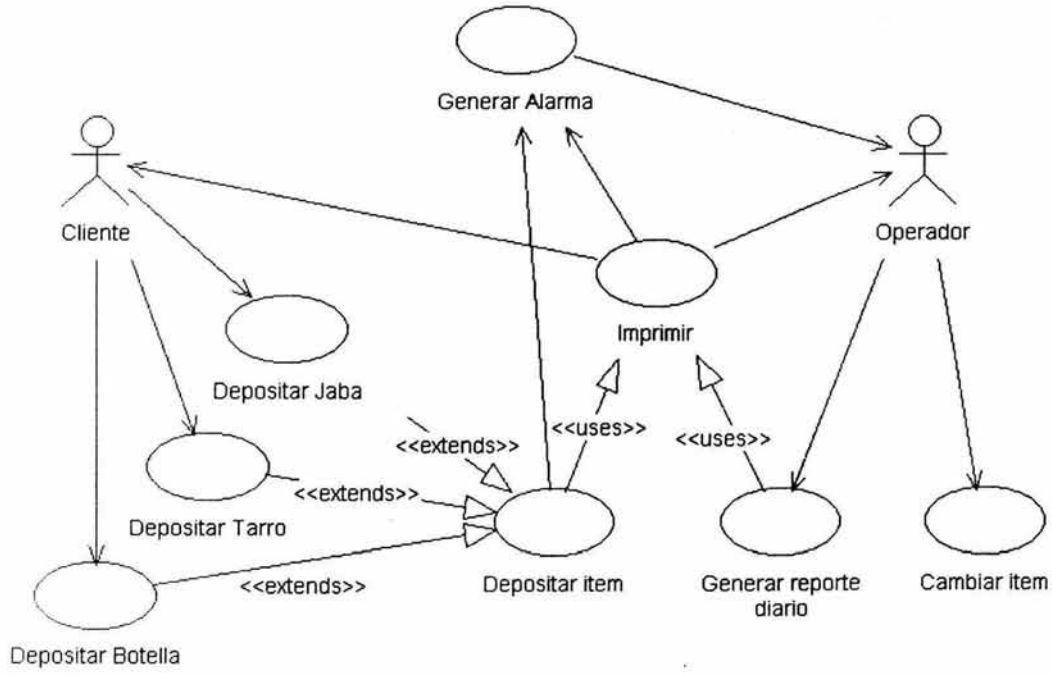


Figura 4.16.- diagrama Casos de Uso (completo) de una máquina recicladora..

4.5.3 Diagramas de Clases.

Un diagrama de clases sirve para visualizar las relaciones entre las clases que involucran el sistema, las cuales pueden ser asociativas, de herencia, de uso.

Un diagrama de clases esta compuesto por los siguientes elementos: Clase y relaciones entre estos

4.5.3.1 Clase.

Es la unidad básica que encapsula toda la información de un Objeto (un objeto es una instancia de una clase). A través de ella podemos modelar el entorno en estudio (una Casa, un Auto, una Cuenta Corriente, etc.).

En UML, una clase es representada por un rectángulo que posee tres divisiones: (ver figura 4.17)

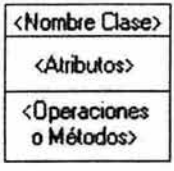


Figura 4.17.- Representación de una clase.

En donde se puede ver tres divisiones:

- **Superior.** Contiene el nombre de la Clase.
- **Intermedio.** Contiene los atributos (o variables de instancia) que caracterizan a la Clase (pueden ser private, protected o public).
- **Inferior.** Contiene los métodos u operaciones, los cuales son la forma como interactúa el objeto con su entorno (dependiendo de la visibilidad: private, protected o public).

A continuación daremos un ejemplo a fin de representar la clase "Cuenta (Bancaria)" que posee como característica:

- Balance
- Y puede realizar las operaciones de:
  - Depositar
  - Girar

- Balance

El diseño asociado sería (figura 4.18):

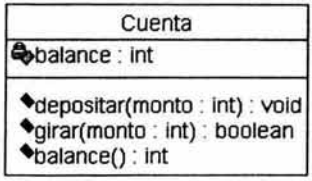


Figura 4.18.- Clase "Cuenta".

En la figura anterior observamos que una clase está compuesta de atributos y métodos

4.5.3.1.1 Atributos.

Los atributos o características de una Clase pueden ser de tres tipos, los que definen el grado de comunicación y visibilidad de ellos con el entorno, estos son:

- **public (+,**  **).** Indica que el atributo será visible tanto dentro como fuera de la clase, es decir, es accesible desde todos lados.

- **private (-,**  **).** Indica que el atributo sólo será accesible desde dentro de la clase (sólo sus métodos lo pueden acceder).

- **protected (#,**  **).** Indica que el atributo no será accesible desde fuera de la clase, pero si podrá ser acezado por métodos de la clase además de las subclases que se deriven (ver herencia).

4.5.3.1.2 Métodos.

Los métodos u operaciones de una clase son la forma en como ésta interactúa con su entorno, éstos pueden tener las características:

- **public (+,**  **).** Indica que el método será visible tanto dentro como fuera de la clase, es decir, es accesible desde todos lados.

- **private (-, ).** Indica que el método sólo será accesible desde dentro de la clase (sólo otros métodos de la clase lo pueden acceder).

- **protected (#, ).** Indica que el método no será accesible desde fuera de la clase, pero si podrá ser acezado por métodos de la clase además de métodos de las subclases que se deriven (ver herencia).

4.5.3.2 Relaciones entre Clases.

Ahora ya definido el concepto de Clase, es necesario explicar como se pueden interrelacionar dos o más clases (cada uno con características y objetivos diferentes).

Pero, antes es necesario explicar el concepto de cardinalidad de relaciones:

Cardinalidad. La cardinalidad es una restricción que se pone a una asociación, que limita el número de instancias de una clase que pueden tener con una instancia de la otra clase. Puede expresarse de las siguientes formas:

- **Cero o uno a muchos:** 0..* (0..n) o 1..* (1..n). Bajo éste rango en el cual uno de los extremos es un asterisco, significa que es un intervalo abierto. Por ejemplo, 1..* significa 1 o más.
- **número fijo:** m (en donde m denota el número, ejemplo: 1).
- **Con un asterisco:** * . No se tienen límites, el * representa el infinito.
- **A través de Comas:** 1, 3..5, 7, 15..*. representa una combinación de los tipos de relaciones antes mencionadas.

En la figura 4.19 se observa que una cuenta bancaria puede tener ligada "n" operaciones, sin embargo, se observa que un abuelo sólo puede vivir con una cantidad de cero a cuatro personas.

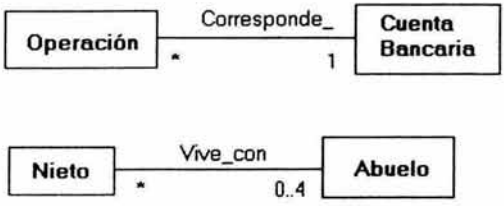


Figura 4.19.- Diagrama donde se visualiza la cardinalidad.

Ahora si conozcamos los tipos de relaciones entre clases:

- **Herencia (Especialización/Generalización)** ——▷. Indica que una subclase hereda los métodos y atributos especificados por una Super Clase, por ende la Subclase además de poseer sus propios métodos y atributos, poseerá las características y atributos visibles de la Super Clase (public y protected).

En la figura 4.20 vemos un ejemplo de herencia.

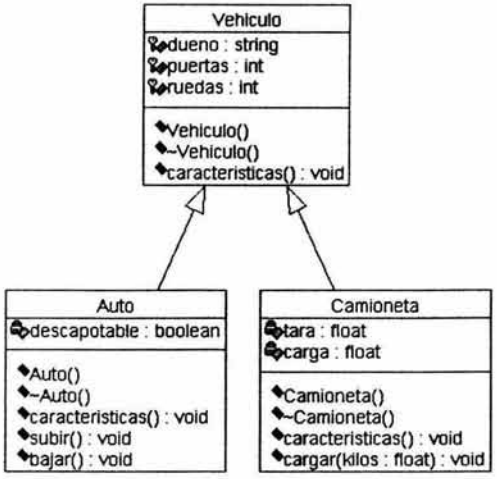


Figura 4.20.- Representación de Herencia.

En la figura se especifica que Auto y Camión heredan de Vehículo, es decir, Auto posee las Características de Vehículo además posee algo en particular que es Descapotable, en cambio Camión también hereda las características de Vehículo pero posee como particularidad propia Acoplado, Tara y Carga.

Cabe destacar que fuera de este entorno, lo único "visible" es el método Características aplicable a instancias de Vehículo, Auto y Camión, pues tiene definición pública, en cambio atributos como Descapotable no son visibles por ser privados.

- **Agregación** ◇——→. Para modelar objetos complejos, n bastan los tipos de datos básicos que proveen los lenguajes: enteros, reales y secuencias de caracteres. Cuando se requiere componer objetos que son instancias de clases definidas por el desarrollador de la aplicación, tenemos dos posibilidades:

    - Por Valor: Es un tipo de relación estática, en donde el tiempo de vida del objeto incluido esta condicionado por el tiempo de vida del que lo incluye. Este tipo de

relación es comúnmente llamada Composición (el Objeto base se construye a partir del objeto incluido, es decir, es "parte/todo").

- Por Referencia: Es un tipo de relación dinámica, en donde el tiempo de vida del objeto incluido es independiente del que lo incluye. Este tipo de relación es comúnmente llamada Agregación (el objeto base utiliza al incluido para su funcionamiento).

En la figura 4.21 observamos un ejemplo de agregación.

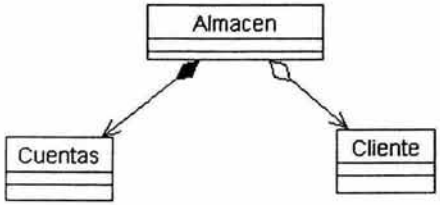


Figura 4.21.- Representación de agregación.

En donde se destaca que:

- Un Almacén posee Clientes y Cuentas (los rombos van en el objeto que posee las referencias).
- Cuando se destruye el Objeto "Almacén" también son destruidos los objetos "Cuentas", en cambio no son afectados los objetos "Cliente".
- La composición (por Valor) se destaca por un rombo relleno.
- La agregación (por Referencia) se destaca por un rombo transparente.

- **Asociación** $\longrightarrow$. La relación entre clases conocida como Asociación, permite asociar objetos que colaboran entre si. Cabe destacar que no es una relación fuerte, es decir, el tiempo de vida de un objeto no depende del otro.

En la figura 4.22 un cliente puede tener asociadas muchas Ordenes de Compra, en cambio una orden de compra solo puede tener asociado un cliente.

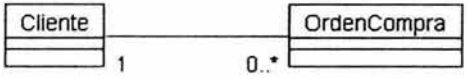


Figura 4.22.- Asociación.

4.5.3.3 Tipos de Clase.

A continuación veremos los dos tipos de clase existentes:

- **Clase Abstracta.** Una clase abstracta se denota con el nombre de la clase y de los métodos con letra "itálica". Esto indica que la clase definida no puede ser instanciada pues posee métodos abstractos (aún no han sido definidos, es decir, sin implementación). La única forma de utilizarla es definiendo subclases, que implementan los métodos abstractos definidos (figura 4.23).

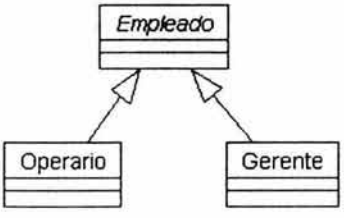


Figura 4.23.- Representación de una clase abstracta.

Clase parametrizada. Una clase parametrizada se denota con un subcuadro en el extremo superior de la clase, en donde se especifican los parámetros que deben ser pasados a la clase para que esta pueda ser instanciada. El ejemplo más típico es el caso de un Diccionario en donde una llave o palabra tiene asociado un significado, pero en este caso las llaves y elementos pueden ser genéricos.

La generalidad puede venir dada de una plantilla (como en el caso de C++) o bien de alguna estructura predefinida (especialización a través de clases) (ver figura 4.24).

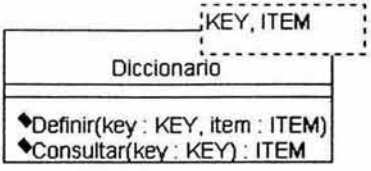


Figura 4.24.- Representación de una clase parametrizada.

En el ejemplo no se especificaron los atributos del Diccionario, pues ellos dependerán exclusivamente de la implementación que se le quiera dar.

A fin de dejar más claro la utilización del diagrama de clase utilizaremos el ejemplo de una máquina despachadora de café y supongamos que se requiere desarrollar el control de una máquina de entrega de café automática:

La máquina debe permitir a una persona entregar una cantidad de dinero en monedas de 100, 200 o 500, escoger uno de los productos de acuerdo a su precio (café negro, café claro, café con leche), escoger (si es pertinente) un nivel de azúcar y entregar el producto y dar cambio.

El dinero que los usuarios introducen se guarda en un recipiente aparte al disponible para dar cambio, el cual se encuentra ordenado por denominación.

Supongamos el modelamiento de una máquina de café. El diagrama de clases podría ser como el que se muestra en la figura 4.25:

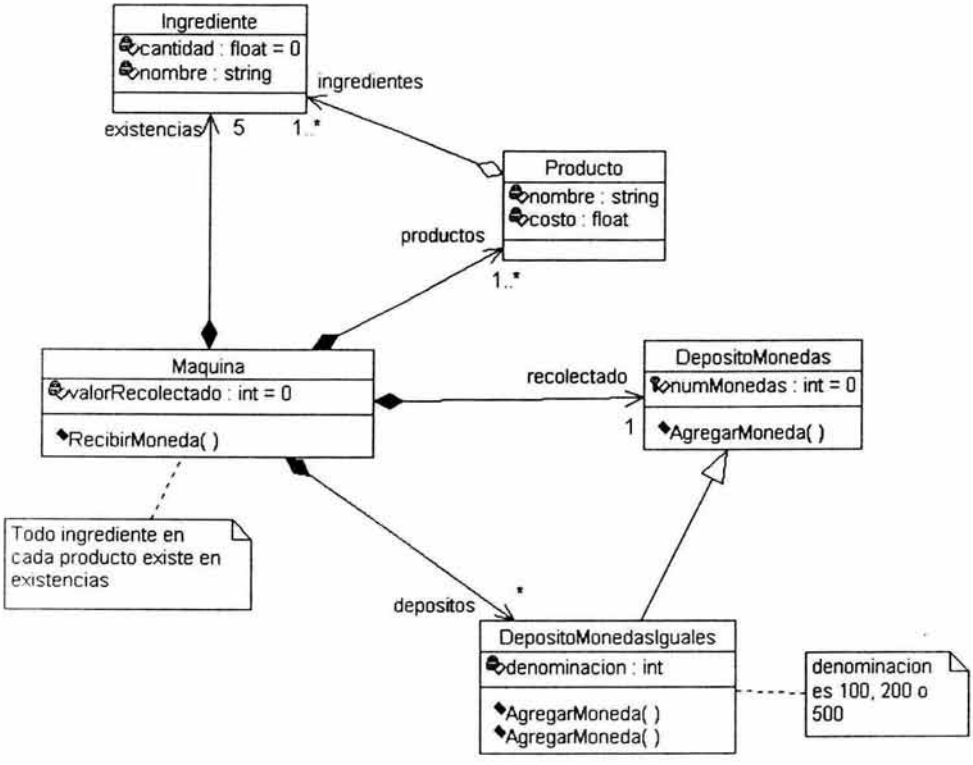


Figura 4.25.- Diagrama de clases para la representación de una máquina despachadora de café.

4.5.4 Diagramas de interacción.

El diagrama de interacción, representa la forma en como un Cliente (Actor) u Objetos (Clases) se comunican entre si en petición a un evento. Esto implica recorrer toda la secuencia de llamadas, de donde se obtienen las responsabilidades claramente.

Dicho diagrama puede ser obtenido de dos partes, desde el Diagrama de Clases o el de Casos de Uso (son diferentes).

Los componentes de un diagrama de interacción son:

- Un Objeto o Actor.
- Mensaje de un objeto a otro objeto.
- Mensaje de un objeto a si mismo.

- **Objeto/Actor.** El rectángulo representa una instancia de un Objeto en particular, y la línea punteada representa las llamadas a métodos del objeto (figura 4.26).

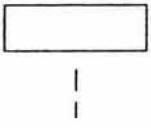


Figura 4.26.- Representación de un Objeto y llamadas a métodos.

- **Mensaje a Otro Objeto.** Se representa por una flecha entre un objeto y otro, representa la llamada de un método (operación) de un objeto en particular (figura 4.27).

Figura 4.27.- Mensaje entre objetos.

- **Mensaje al Mismo Objeto.** No solo llamadas a métodos de objetos externos pueden realizarse, también es posible visualizar llamadas a métodos desde el mismo objeto en estudio (figura 4.28).

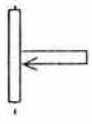


Figura 4.28.- Mensaje así mismo.

En los diagramas de interacción se muestra un patrón de interacción entre objetos. Hay dos tipos de diagrama de interacción, ambos basados en la misma información, pero cada uno enfatizando un aspecto particular: Diagramas de Secuencia y Diagramas de Colaboración.

4.5.4.1 Diagrama de Secuencia.

Un diagrama de Secuencia muestra una interacción ordenada según la secuencia temporal de eventos. En particular, muestra los objetos participantes en la interacción y los mensajes que intercambian ordenados según su secuencia en el tiempo. El eje vertical representa el tiempo, y en el eje horizontal se colocan los objetos y actores participantes en la interacción, sin un orden prefijado. Cada objeto o actor tiene una línea vertical, y los mensajes se representan mediante flechas entre los distintos objetos.

El tiempo fluye de arriba abajo. Se pueden colocar etiquetas (como restricciones de tiempo, descripciones de acciones, etc.) bien en el margen izquierdo o bien junto a las transiciones o activaciones a las que se refieren.

En el siguiente ejemplo, tenemos el diagrama de secuencia, que da detalle al para caso de una la máquina despachadora de café, la cual proveniente del siguiente modelo de clases, observado con anterioridad (figura 4.29):

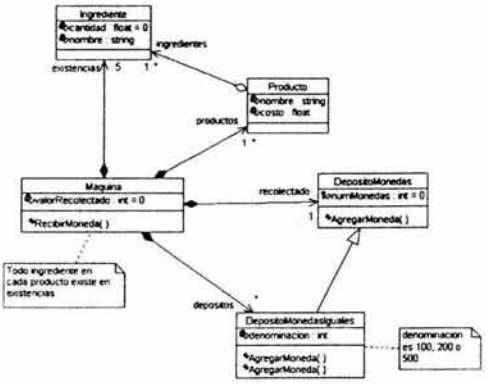


Figura 4.29.- Diagrama de clases ejemplo "Maquina despachadora".

Por lo que el diagrama de secuencia para dicho diagrama es (figura 4.30):

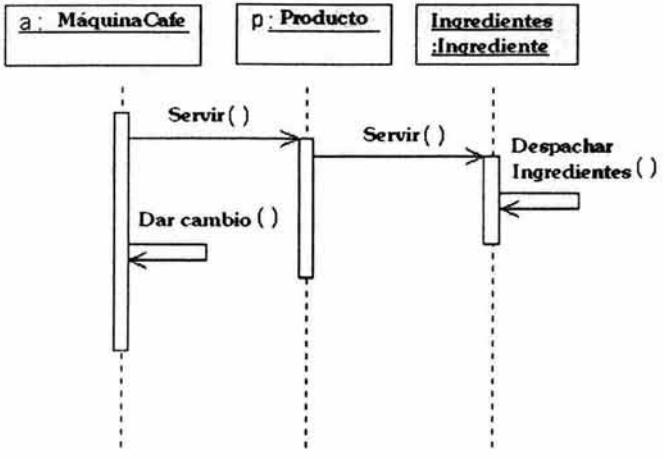


Figura 4.30.- Diagrama de secuencia.

En donde se hacen notar las sucesivas llamadas a Servir() (entre objetos) y la llamada a Despachar Ingredientes() y dar cambio().

4.5.4.2 Diagrama de Colaboración.

Los diagramas de colaboración proporcionan la representación principal de un escenario, ya que las colaboraciones se organizan entorno a las relaciones existentes entre unos objetos con otros, así como la secuencia de los mensajes que surgen en estás relaciones, la cuales se indican a través de un número.

Este tipo de diagramas se utilizan más frecuentemente en la fase de diseño, es decir, cuando estamos diseñando la implementación de las relaciones. Se toma como ejemplo el caso de la máquina despachadora de café, pero sólo para la sección de "Pedir el Producto" (figura 4.31)

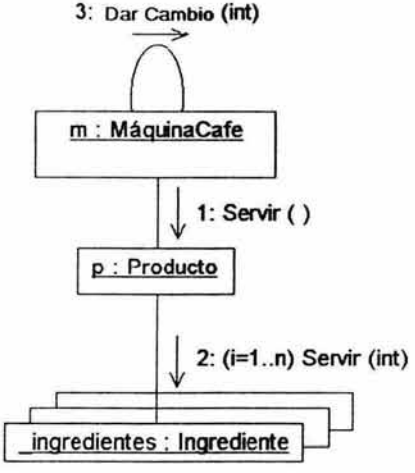


Figura 4.31.- Diagrama de Colaboración.

4.5.4.3 Diagrama de Estado.

Un estado es una condición durante la vida de un objeto, de forma que cuando dicha condición se satisface se lleva a cabo alguna acción o se espera por un evento. El estado de un objeto se puede caracterizar por el valor de uno o varios de los atributos de su clase, además, el estado de un objeto también se puede caracterizar por la existencia de un enlace con otro objeto.

El diagrama de estados y transiciones engloba todos los mensajes que un objeto puede enviar o recibir. En un diagrama de estados, un escenario representa un camino dentro del diagrama. Dado que generalmente el intervalo entre dos envíos de mensajes representa un estado, se pueden utilizar los diagramas de secuencia (figura 4.30) para buscar los diferentes estados de un objeto.

En todo diagrama de estados existen por lo menos dos estados especiales inicial y final: start y stop. Cada diagrama debe tener uno y sólo un estado start para que el objeto se encuentre en estado consistente. Por contra, un diagrama puede tener varios estados stop.

Una transición entre estados representa un cambio de un estado origen a un estado sucesor destino que podría ser el mismo que el estado origen, dicho cambio de estado puede ir acompa nado de alguna acción. Las acciones se asocian a las transiciones y se considera que ocurren de forma rápida y no interrumpible.

Existen dos formas de transicionar en un diagrama de estados: automáticamente y no automáticamente. Se produce una transición automática cuando se acaba la actividad del estado origen (no hay un evento asociado con la transición). Se produce una transición no automática cuando existe un evento que puede pertenecer a otro objeto o incluso estar fuera del sistema.

Los diagramas de estados muestran el comportamiento de los objetos, es decir, el conjunto de estados por los cuales pasa un objeto durante su vida, junto con los cambios que permiten pasar de un estado a otro.

Un ejemplo para el caso de la máquina de café son los estados posibles (figura 4.32).

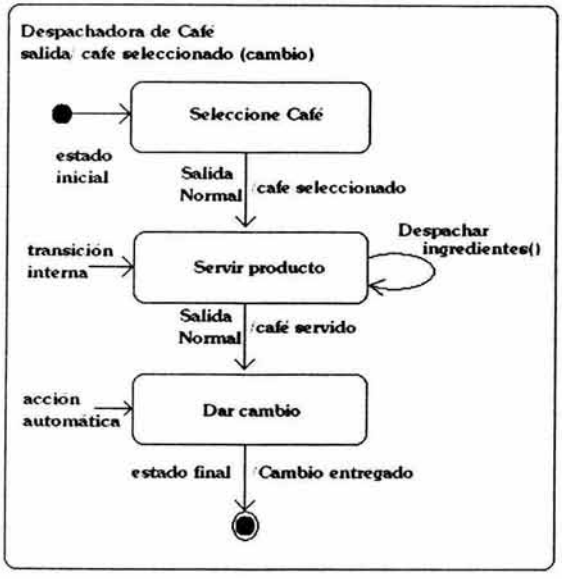


Figura 4.32.- Diagrama de Estados.

4.6 Ejemplo "Venta de boletos de cine mediante Internet".

A pesar de haber visualizado los elementos comunes en los diagramas y en general la interpretación de estos en la gran mayoría de los mismos, en general se puede decir que dependerá de las características del sistema a modelar para la utilización de unos u otros diagramas, sin embargo, de los diagramas más representativos (y los más usados) son: diagrama de Casos de Uso, diagrama de Secuencia y diagrama de Clases, mismos que observaremos mediante un ejemplo que pretende modelar un sistema "Venta de boletos de cine mediante Internet".

- **Diagrama de Casos de Uso.** representa gráficamente los distintos escenarios en los que se utilizará un sistema, mostrando en c/u de ellos lo que el sistema tiene que hacer y como lo tiene que hacer; Una definición estricta de un caso de uso esta dad como

cada interacción supuesta con el sistema a desarrollar, donde se representan los requisitos funcionales.

En la figura 4.33 se utiliza el diagrama de casos de uso, en donde se muestran tres actores (los clientes, los taquilleros y los jefes de taquilla) y las operaciones que pueden realizar (sus roles).

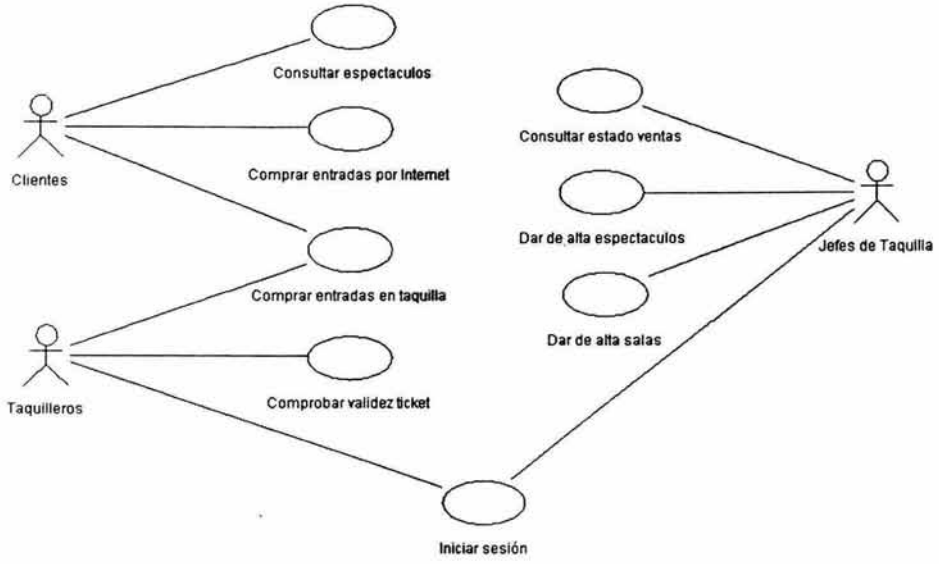


Figura 4.33.- Diagrama de Casos de uso del ejemplo "Venta de boletos por Internet".

En la figura 4.33 observamos que:

Los Clientes interactúan en los siguientes casos de uso:

- Consultar espectáculos.
- Comprar entradas por Internet.

- Comprar entradas en taquilla.

Los Taquilleros interactúan en los siguientes casos de uso:

- Comprar entradas en taquilla.
- Comprobar validez de ticket.
- Iniciar sesión.

Los Jefes de Taquilla interactúan en los siguientes casos de uso:

- Iniciar sesión.
- Consultar estados ventas.
- Dar de alta espectáculos.
- Dar de alta salas.

- **Diagrama de secuencia.** En éste se muestra la interacción de los objetos que componen un sistema de forma temporal. Siguiendo el ejemplo de venta de entradas, la figura 4.34 muestra la interacción de crear una nueva sala para un espectáculo.

  En el diagrama de secuencia, se observa que el "Usuario" ejecutará la acción de crear una nueva Sala, al sistema y este le responderá solicitándole datos, mismos que serán capturados y validados, para así el sistema procederá a crear una nueva Sala, para que así finalmente el sistema, obtenga respuesta de la sala y este a su entregar respuesta al usuario.

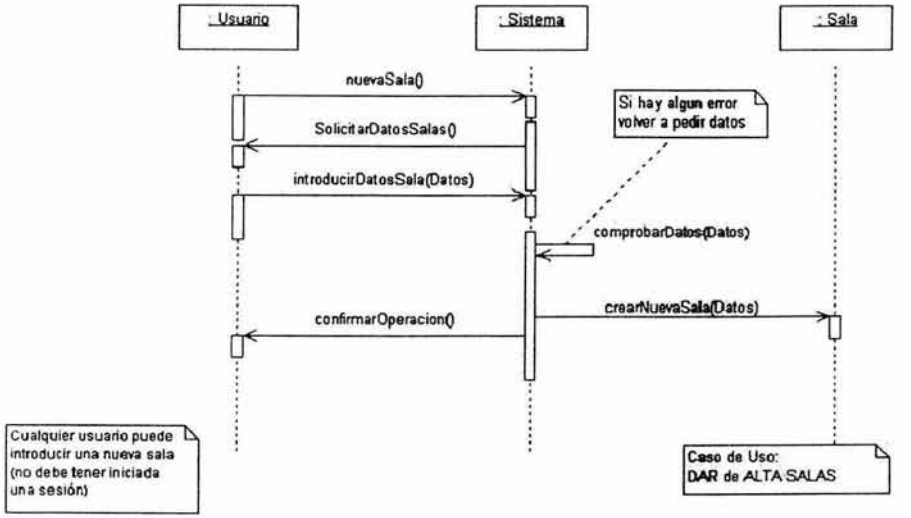


Figura 4.34.- Diagrama de Secuencia del ejemplo "Venta de boletos por Internet".

- **Diagrama de clases.** muestra un conjunto de clases, interfaces y sus relaciones. Éste es el diagrama más común a la hora de describir el diseño de los sistemas orientados a objetos.

A continuación se muestra el diagrama de clases en base a nuestro ejercicio (figura 4.35).

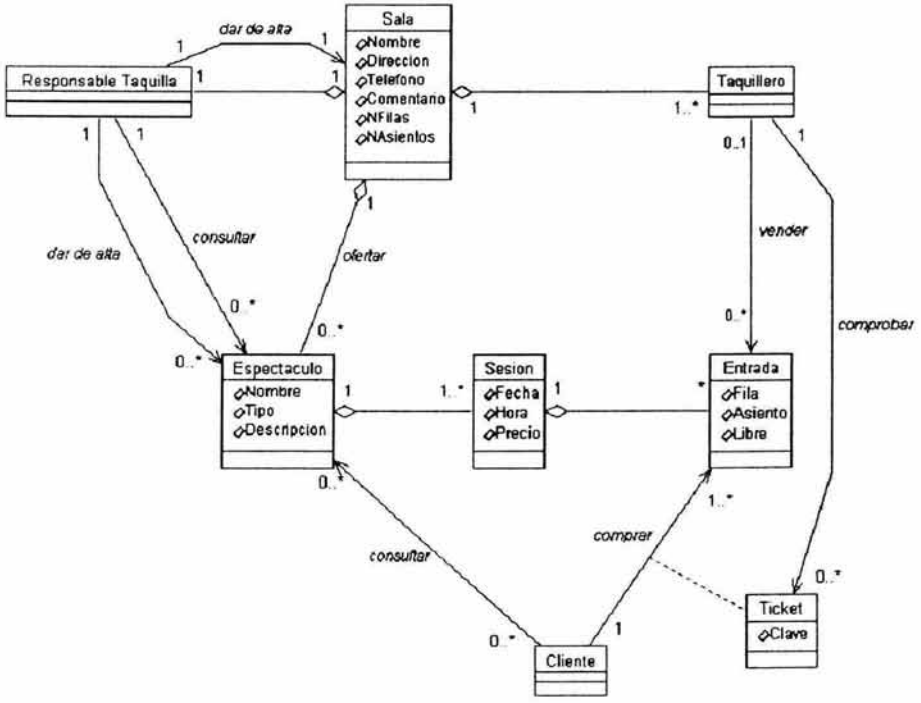


Figura 4.35.- Diagrama de Clases del ejemplo "Venta de boletos por Internet".

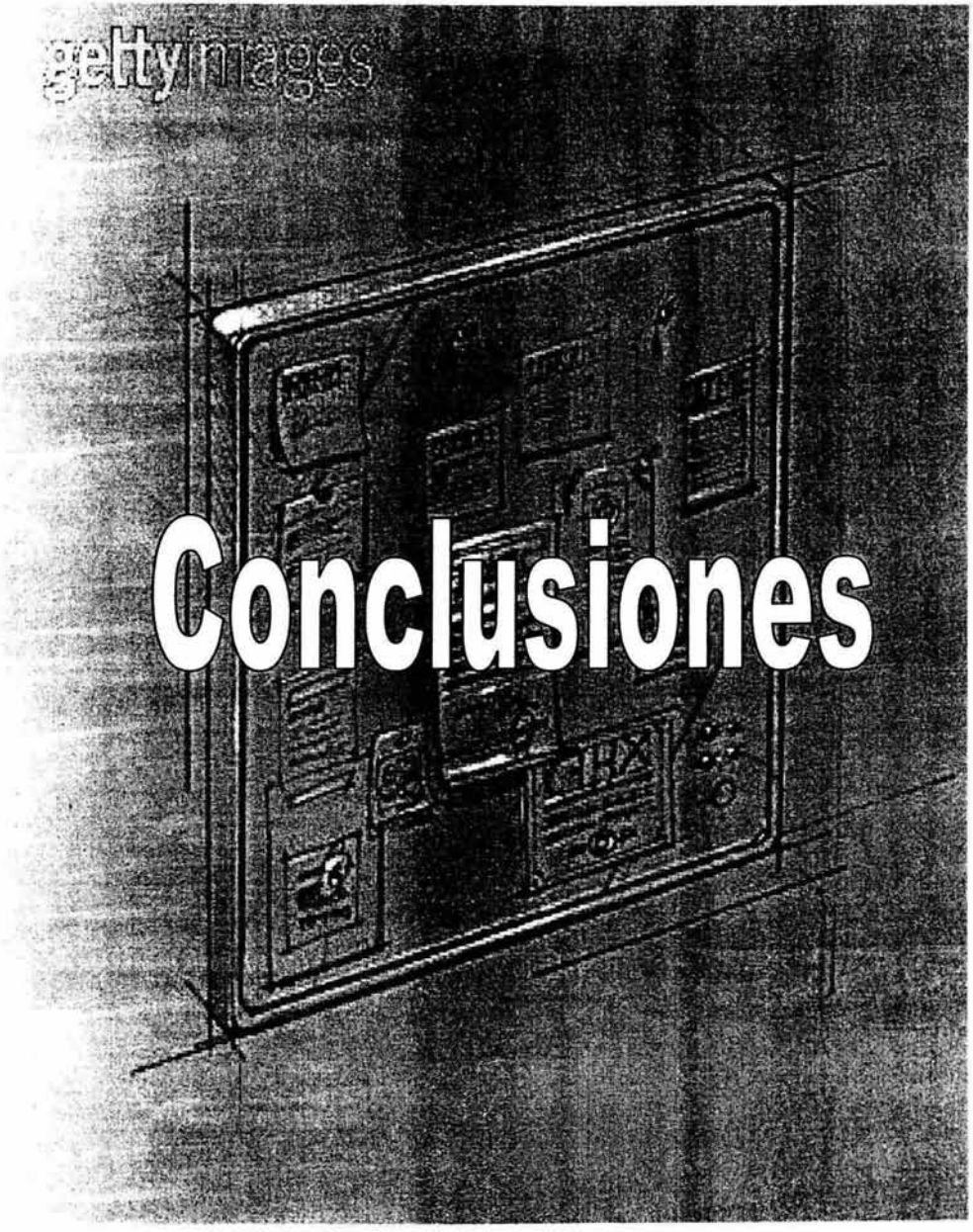
- En la figura 4.25 se muestran las clases globales, sus atributos y las relaciones de una posible solución al problema de la venta de entradas, en donde se visualizan las siguientes clases:

- Responsable Taquilla.
- Sala.
- Taquillero.
- Espectáculo.
- Sesión.
- Entrada.
- Cliente.
- Ticket.

El resto de diagramas muestran distintos aspectos del sistema a modelar. Para modelar el comportamiento dinámico del sistema están los de colaboración y estados.

El esquema de interacción entre la interfaz y el modelo del mundo permite trabajar en paralelo cada uno de ellos y permite realizar cambios sin afectar el proceso de desarrollo. Por otra parte, al trabajar OO se facilita la reutilización de código así como la portabilidad del mismo en el caso de usar lenguajes de programación OO como C, C++, C#, JAVA, entre otros.

La inclusión del modelo OO articulado al ciclo de la Ingeniería de Software (IS) permite aprovechar todo el potencial de las metodologías de la IS. Esto es importante a la hora de desarrollar software de calidad. Esta integración de enfoques facilita el mantenimiento de los sistemas informáticos, así como la expansión de éste a medida que se requiera, garantizándose así integridad con cada cambio que se realice en el modelo del mundo.


gettyimages
Conclusiones

Hemos pretendido ofrecer una visión global de lo que es el software, sin embargo, no debemos olvidar que el origen de todo nuestro esfuerzo en mejorar las técnicas de desarrollo se debe a la satisfacción de los requerimientos de nuestro usuario final y es por esto, que la Ingeniería de Software es una actividad que involucra a clientes, usuarios, equipo de desarrollo, administradores de proyectos, etc.; por lo tanto, el proceso de IS no depende solamente de la forma en cómo se percibe el problema, sino también, del nivel de experiencia que tengan los involucrados.

Tomando en cuenta la magnitud de comunicación y el trabajo en equipo que debe existir en la IS, es necesario enfatizamos más en cerrar las brechas que todavía existen, incluyendo los siguientes elementos:

- *Factores sociales. involucrar al grupo para compartir sus experiencias.*
- *Factores de problemas específicos. el dominio de la estructura y estándares disponibles.*
- *Factores organizacionales. tiempo y costo presupuestados.*
- *Factores de diseño. por ejemplo, interfases de usuario*

Es importante tomarse el tiempo necesario para conocer a nuestros clientes y usuarios, así como su ambiente de trabajo. Esto, también ayuda a establecer una buena relación de trabajo y comunicación entre el equipo de desarrollo y los clientes. Es realmente necesario que los clientes y usuarios participen en la definición de sus requerimientos, pues ellos son los que deciden nuestro destino en el proyecto, deciden si les el proyecto le gusta o no y además lo financian, de lo contrario llegaríamos a un resultado final totalmente distinto a lo que originalmente nos han solicitado.

Dada la complejidad del software, es necesario apoyarse en modelos que ayuden a representar los diversos problemas a los que hoy en día se enfrenta nuestro usuario final, sin embargo, no existe un modelo ideal para la Ingeniería de Software (IS); encontrar el método o la técnica perfecta es una ilusión, pues cada método y técnica ofrece diferentes soluciones ante un problema. Los modelos mencionados nos dan un ámbito global de las ventajas y desventajas de cada uno de éstos para que nosotros aprovechemos lo mejor de cada uno de estos e inclusive los integremos en una solución.

Si pudiéramos hablar de la Ingeniería de software como teoría y práctica, la teoría son los modelos, los cuales nos dan las bases para interpretar cualquier problema y la práctica es el modelaje orientado a objetos.

La orientación a objetos es un nuevo paradigma para el diseño de aplicaciones informáticas que permite reducir la complejidad de forma segura y robusta. Este nuevo paradigma de diseño puede aplicarse de manera natural al diseño de sistemas en general.

Los diagramas gráficos del modelado orientado a objetos ayudan en la toma de decisiones durante el diseño porque representan de manera sencilla y compacta los elementos esenciales de la navegación.

Finalmente encontramos en UML la fusión entre teoría y práctica, resolviendo de forma bastante satisfactoria un viejo problema del desarrollo de software como es su modelado gráfico. Además, se ha llegado a una solución unificada basada en lo mejor que había hasta el momento, lo cual lo hace todavía más excepcional.

El desarrollo de modelos interactivos es una necesidad actual que debe ser atacada por desarrolladores de software y es por ello que los diagramas de UML resultan adecuados para describir el comportamiento de un sistema. Por ejemplo, los diagramas de interacción son idóneos para la descripción del comportamiento de varios objetos en un único caso de uso, pero los diagramas de estado describen el comportamiento de los objetos a través de diferentes casos de uso.

Es fácil predecir que UML será el lenguaje de modelado de software de uso universal y las principales razones para esto es:

- *En el desarrollo de este lenguaje han participado investigadores de reconocido prestigio.*
- *El lenguaje ha sido apoyado prácticamente por todas las empresas importantes de informática.*
- *La OMG ha aceptado a UML como un estándar y .*
- *Prácticamente todas las herramientas CASE y de desarrollo han adaptado a UML como lenguaje de modelado.*

El éxito de UML será medido por su apropiado uso en proyectos exitosos. UML no garantiza el éxito, sino que permite a los Ingenieros enfocarse en la distribución de valor, usando un consistente, estandarizado y soportable por herramientas, lenguaje para el modelamiento.

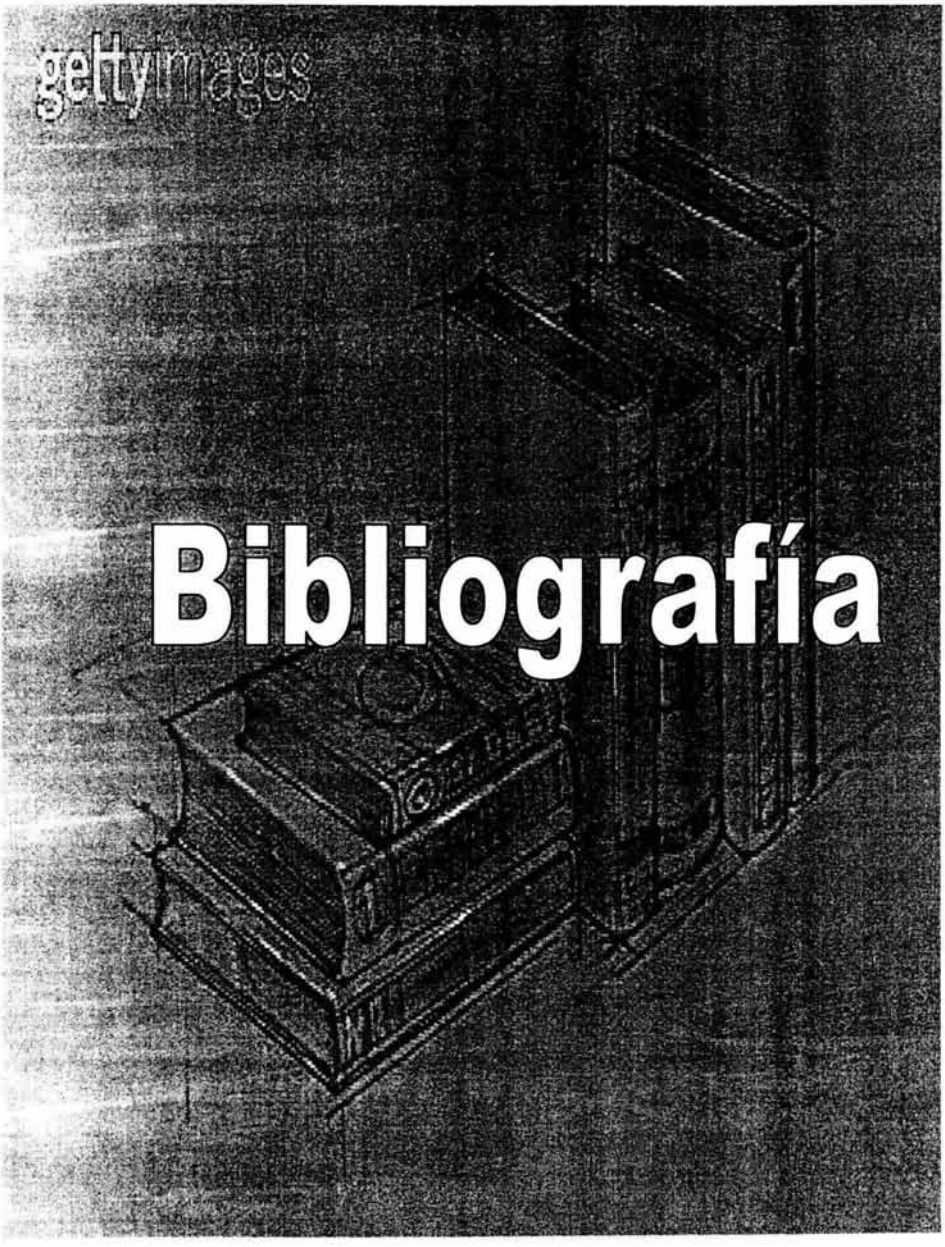
La sensación de que el software está propiciando uno de esos extraños momentos en el cambio de la industria entera se debe a la ingeniería del software, sin embargo, no es sino

hasta los 90´s en donde se enfatizo el término "Administración de Software" en donde se dio una nueva visión en busca de una descripción más apropiada a las actividades involucradas en el proceso así como a enfatizar la importancia de mantener una buena relación entre los clientes y el equipo del proyecto.

Dado que el origen de UML es relativamente reciente, existe un mundo de oportunidades para crecer y ser parte de una solución integral. Espero que la experiencia expuesta en ésta tesis, así como el acopio de información que aquí se brinda, sea de ayuda para las generaciones que nos siguen.

¡No le quiten a la universidad lo que ella les da: Prestigio!!!
(Profesor Juan Méndez, en su memoria)

Daniel Benjamín Alcántara Zavala
Noviembre del 2004.



Bibliografía

- **Adam Smith**, *"Investigación sobre la naturaleza y la causa de la riqueza de las naciones"*, 1ª edición, Addison-Wesley 1776.
- **Álvarez Cárdenas S.**, "*Metodología para el desarrollo de aplicaciones para medios ambientes visuales estructurados*", 1993.
- **Frederick P. Brooks**, *"The Mythical Man-Mont"*, Addison-Wesley, 1995.
- **Grady Booch, James Rumbaugh, Ivar Jacobson**, *"El Lenguaje Unificado de Modelado"*, Addison Wesley. 1999.
- **Gamma, E. & Helm, R. & Johnson, R. & Vlissides, J.**, *"Design Patterns. Elements of Reusable Object-Oriented Software"*, Prentice – Hall, 1990.
- **Goldberg, A. & Rubin**, *"Succeding with Objects. Decisión Frameworks for Project Management"*, Addison Wesley, 2001.
- **Lauren Ruth Wiener**, *"Digital Woes" (Dolor digital)*, Addison-Wesley, 1993.
- **López Nathalie, Migueis Jorge, Pichon Emmanuel**, *"Integrar UML en los proyectos"*, Eyrolles Gestión 2000, 1998.
- **Martin Fowler con Kendall Scott**, *"UML Gota a Gota"* Eyrolles Gestión 2000, 1997.
- **Muller Pierre Alain**, *"Modelado de objetos con UML"*, Eyrolles Gestión 2000, 1997.
- **Nancy G. Leveson**, *"Safeware: System Safety and Computers"*, Addison-Wesley, 1995.
- **Peter G. Neumann**, *"Computer-Related Risks"*, Addison-Wesley, 1995.
- **Piattini Mario G , Calvo - Manzano José A , Cervera Joaquín , Fernández Sanz Luis**, *"Análisis y diseño detallado de Aplicaciones Informáticas de Gestión"*, Rama. 1996.
- **Pressman Roger S.**, *"Ingenieria del Software. Un enfoque práctico"* McGraw-Hill, 1998.

- **Rumbaugh, James et al.**, *"Modelado y diseño orientados a objetos"*, Madrid: Prentice Hall, 1996.

- **Rumbaugh James , Blaha Michael, Premerlani William, Eddy Frederick, Lorensen William**, *"Modelado y diseño orientados a objetos"*, Prentice - Hall . 1991.

- http://www.sei.cmu.edu/legacy/case/case_whatis.html, **Brackett, Jhon W.**, *"Software Requirements Software Engineering Institute"*, Education Program Carnegie Mellon University, 1990.

- http://firstmonday.org/issues/issue6_12/kelty/, **Christopher M. Kelty**, *"Free Software / Free Science"*.

- http://www.sigmod.org/sigmod/dblp/db/indices/a-tree/s/Schwabe:Daniel.html, **Chwabe, Daniel and Simone D.J. Barbosa**, *"Navigation Modeling in Hypermedia applications"*. Technical Report MCC 42/94, Departamento de Informática, PUC-Rio, 1994.

- http://www.dwheeler.com/oss_fs_why.html, **David A. Wheeler**, *"Why Open Source Software/Free Software (OSS/FS)? Look at the Numbers!"*.

- http://firstmonday.org/issues/issue6_12/lancashire, **David Lancashire**, *"Code, Culture and Cash: The Fading Altruism of Open Source Development"*.

- http://www.acm.org/crossroads/espanol/xrds7-4/onpatrol74.html, **Feamster N**, *"La Seguridad en la Ingeniería de Software 2001"*.

- http://congreso.hispalinux.es/congreso2001/actividades/ponencias/robles/pdf/desarrolladores.pdf, **Gregorio Robles**, *"Los desarrolladores de Software Libre"*.

- http://www.ifi.unizh.ch/groups/req/, **Hofmann, Hubert**, *"Requirements Engineering Institute for Informatics"*, University of Zurich. 1993.

- http://sinetgy.org/~jgb/articulos/soft-libre-monopolios/, **Jesús M. González**, *"Software libre, monopolios y otras yerbas"*.

- http://citeseer.ist.psu.edu/context/282812/0, **Lange, Danny B.**, *"An Object-Oriented Design Method for Hypermedia Information Systems"*. Conferencia annual de Hawaii referente a los sistemas enfocados a las ciencias, 1994.

- http://www.software-engineer.org/article_index.php?name=Requirement+Management&category_id=9, **Malan, Ruth.**, *"Functional Requirements and Use Cases"*, Hewlette-Packard Company. 1999.

- www.huihoo.com/development/rup/apprmuc.htm, **Oberg, Roger, Probasco Leslee, Maria Ericsson.**, *"Applying Requirements Management with Use Cases"*. Rational Software Corporation. 1998.

- http://www.omg.org/technology/documents/formal/uml.htm, **Object Management Group**, *"OMG Unified Modeling Language Specification"*. 1999.

- http://www.magma.com.ni/~jorge/upoli_uml/refs/Que_es_UML.doc, **OReilly & Associates**, *"UML en Resumen: Una Rápida Referencia de Escritorio"*,1998.

- http://floss1.infonomics.nl/finalreport/, **Rishab A. Ghosh, Rüdiger Glott, Bernhard Krieger & Gregorio Robles**, *"Free/Libre Open Source Software: Survey and Study - Final report"*.

- http://www.itba.edu.ar/capis/webcapis/PVBITBA/articulosp.htm, **Rossi Bibiana, Britos Paola, García Martínez**, *"Modelado de Objetos"*, Revista del Instituto Tecnológico de Buenos Aires Nº 21. 1998.

- http://physics.nebrwesleyan.edu/UNL.html, **Saiedian, H., Dale, R.** , *"Requirements Engineering: Making the connection between the software developer and customer"*. Department of Computer Science, University of Nebraska. 1999.

- http://www.sigmod.org/sigmod/dblp/db/indices/a-tree/s/Samos:Jos=eacute=.html, **Samos, J.**, *"Definition of External Schemas in Object Oriented Databases"*, Conferencia sistemas de información orientado a objetos. 1995.

- http://citeseer.ist.psu.edu/scholl91updatable.html, **Scholl, M.H., Laasch, C. Tresch**, *"Updatable Views in Object-Oriented"*. 1991.

- http://home.comcast.net/~salhir/, **Sinan Si Alhir**, *"¿Qué es el Lenguaje para Modelamiento Unificado (UML)?* (What is the Unified Modeling Language (UML)?), Julio 1998.

- http://widi.berlios.de, **The Widi Survey**, *"Encuesta realizada a más de 5500 desarrolladores (2001)"*.

Glosario

Acrónimo. Palabra formada por las iniciales u otras letras de otras palabras.

ADA. Lenguaje de programación de propósito general que puede ser usado para cualquier problema. Tiene una estructura de bloque y mecanismos de tipo de datos igual que Pascal, aunque con extensiones para aplicaciones de tiempo real y distribuidas. Provee una forma más segura de encapsulación que Pascal y las últimas versiones estándares incrementan el desarrollo de objetos y hay herencia de métodos

ALGOL 58. (Algorithm Language) (Lenguaje de Algoritmos). Lenguaje de programación que apareció en 1958, fue diseñado para ser independiente de cualquier computadora de su tiempo, con una gramática bien definida.

ALGOL 60. Lenguaje de programación que se convirtió en un estándar para la descripción de algoritmos: utilizado principalmente sobre papel (mundo académico y de investigación). Se utilizó bastante en Europa. Surge en París en enero de 1960.

Analógicos. Señales visuales o acústicas que se convierten en una tensión eléctrica variable, que se puede reproducir directamente a través de altavoces o almacenar en una cinta o disco. Este tipo de señales son mucho más vulnerables a los ruidos y las interferencias que las señales digitales.

Abstracta. Se dice de lo que no representa cosas concretas, sino que atienden exclusivamente a elementos de forma, color, proporción.

Ambigua. Posibilidad de que algo pueda entenderse de varios modos o de que admita distintas interpretaciones.

Atributo. Son cada una de las características de un objeto: identificador, descripción.

B. Lenguaje de programación que surge en los Laboratorios Bell, con este lenguaje se crearon las primeras versiones del sistema operativo UNIX.

BASIC. Acrónimo de (Beginners All-purpose Symbolic Instruction Code) (Código de Instrucciones Simbólicas de Uso General para Principiantes). Lenguaje de programación de alto nivel para el desarrollo de programas de tipo general creado en 1964, hoy en desuso.

Bencina. Líquido incoloro, volátil e inflamable, obtenido del petróleo, que se emplea como disolvente.

C. Lenguaje de programación estructurado que surge en 1972. El lenguaje "C", es una evolución del lenguaje B, Al contrario de sus antecesores, C es un lenguaje con tipos, es decir, que cada elemento de información ocupaba un 'palabra' en memoria y la tarea de tratar cada elemento de datos como número entero, real, o arreglos, no recaía en el programador.

C++. Lenguaje de programación estructurado, basado en el lenguaje "C", razón por la cual se conoce como un superconjunto del lenguaje "C", al que recubre con una capa de soporte a la POO (Programación Orientada a Objetos). Por lo tanto permite la definición, creación y manipulación de objetos.

CAD. Acrónimo de (Computer Aided Design) (Diseño Asistido por Computadora). Técnicas que permiten a los diseñadores, arquitectos, aparejadores, etc., utilizar en su trabajo herramientas informáticas para acortar los tiempos necesarios en el diseño de productos.. En arquitectura, el CAD facilita la labor de diseño y de cálculo estructural, aportando también información sobre los materiales a utilizar, etc.

CASE. Acrónimo de (Computer Aided Software Engineering) (Ingeniería de Software Asistida por computadora). Conjunto de herramientas, lenguajes y técnicas de programación que permiten la generación de aplicaciones de manera semiautomática. Las herramientas CASE liberan al programador de parte de su trabajo y aumentan la calidad del programa a la vez que disminuyen sus posibles errores.

Casos de uso. Diagrama en donde se representan los distintos escenarios en los que participa un usuario. La identificación de estos casos de uso se hace con base en los requerimientos de la aplicación a desarrollar.

Clase. Definición de atributos y métodos para un conjunto de objetos.

CLIPPER. Lenguaje de programación enfocado a la administración de la información que manejaba DBASE, formado por un conjunto de comandos y funciones similares a las usadas con DBASE. Su primera versión se creó en 1985 en los laboratorios de Natuncket. CLIPPER está escrito en lenguaje C y Ensamblador.

CLOS. Acrónimo de (Common Lisp Object System) (Sistema de Objetos comunes Lisp). Lenguaje de programación basado en los conceptos de funcionalidad múltiple, inherente, y combinación de métodos. Todos los objetos en el sistema son instancias de clases que son una extensión al lenguaje común LISP. Surge en el verano de 1986 con el apoyo de Xerox.

COBOL. Acrónimo de (Common Business Oriented Language) (Lenguaje Común Orientado hacia Aplicaciones de Empresa), Lenguaje de programación de tercera generación muy empleado para aplicaciones comerciales. Utiliza el idioma inglés como base para las instrucciones incluidas en el programa. Este lenguaje de programación fue desarrollado entre 1959 y 1961 que utiliza como base el idioma inglés y que se caracteriza por su gran facilidad de lectura.

COM. Acrónimo de (Component Object Model) (Modelo de Componentes de Objetos) Estándar establecido por Microsoft que constituye una redefinición de la extensión del estándar OLE (Object Linking and Embedding), que servía para que objetos documentales generados por una aplicación interactúen con otros objetos documentales generados por otras aplicaciones, para abarcar todo tipo de objetos. Permite que las aplicaciones puedan utilizar los servicios de otras aplicaciones.

Computadora Personal. Se denomina computadora al conjunto formado por la CPU (unidad de procesamiento central), el monitor y el teclado. Estas máquinas trabajan con software variado, sin embargo, deberán de contar con un sistema operativo.

CORBA. Acrónimo de (Common Object Request Broker Architecture) (Arquitectura de Requerimiento de Intermediación de Objetos Comunes) Estándar definido por el OMG (Object Management Group) para manejar objetos distribuidos, particularmente en el Internet. Las alternativas a CORBA en el mercado son #dcom">DCOM (Distributed Component Object Model) de Microsoft, y RMI (Remote Method Invocation) que es parte de Java.

DBASE. Programa de base de datos para DOS y Windows de Borland. dBASE fue el primer DBMS relacional completo para computadores personales y originalmente se desarrolló para máquinas CP/M. En un principio fue comercializado por Ashton-Tate, adquirida más tarde por Borland. El formato de archivos dBASE DBF se ha convertido en un estándar de facto utilizado por muchas aplicaciones.

DBF. Extensión que indica que el fichero en cuestión ha sido generado con el programa dBase, y tiene una estructura de base de datos.

DBMS. Acrónimo de (Data Base Management System) (Sistemas de Administración de Bases de Datos). Sistema de administración de bases de datos. Software que controla la organización, almacenamiento, recuperación, seguridad e integridad de los datos en una base de datos. Acepta solicitudes de la aplicación y ordena al sistema operativo transferir los datos apropiados.

DCOM. Acrónimo de (Distributed COM) (COM Distribuido). Metodología capaz de manejar objetos provenientes de otros equipos. Para el efecto, DCOM hace uso de servicios como DNS (Domain Name System). El estándar DCOM apareció en el mercado con Windows NT 4.0. Las alternativas a DCOM son CORBA (Common Object Request Broker Architecture) del Object Management Group, y RMI (Remote Method Invocation) que es parte de Java.

Depuración. Hace referencia a métodos para refinar el código del programa que se está desarrollando, identificando y eliminando todos los posibles errores que éste tenga.

Desarrollo. Por extensión, se utiliza la palabra «desarrollo» para indicar el trabajo de elaboración de un programa o aplicación.

Diagrama de interacción. Indica la secuencia de acciones que deben seguirse para realizar una tarea en el modelo computacional. Este tipo de diagrama puede indicarse de dos maneras: diagrama de secuencia, en el cual se muestra la secuencia lineal de acciones en determinado momento; diagrama de colaboración, muestra la secuencia de acciones de modo no lineal, resaltando las relaciones y/o dependencias entre diferentes clases del modelo.

Disciplina. 1 Doctrina; regla de enseñanza impuesta por un maestro a sus discípulos.
2 Asignatura.
3 Conjunto de reglas para mantener el orden y la subordinación entre los miembros de un cuerpo.

DNS. Acrónimo de (Domain Name Service) (Servicio de Nombres de Dominio). Servicio de búsqueda de datos de uso general, distribuido y multiplicado. Su utilidad principal es la búsqueda de direcciones IP de sistemas

anfitriones (hosts) basándose en los nombres de éstos. El estilo de los nombres de host utilizado actualmente en Internet es llamado "nombre de dominio".

DOS. Acrónimo de (Disk Operating System) (Disco de Sistema Operativo). Sistema operativo utilizado en la mayoría de las computadoras personales (PC) existentes. El origen del nombre de Sistema Operativo de Disco se debe a que el DOS permite la administración del disco duro y los disquetes.

Escenario de interacción. Cada uno de los momentos de interacción que tiene el usuario con la aplicación (registrarse, escoger reto, etc.).

Exponencial. Crecimiento que tiene un ritmo que aumenta cada vez más rápidamente.

Extrapolado. Deducción del valor de una variable en una magnitud a partir de otros valores no incluidos en dicha magnitud.

FORTRAN. Acrónimo de (FORmula TRANslator) Traductor de fórmulas. Primer lenguaje de programación de alto nivel y compilador, desarrollado en 1954 por IBM. Originalmente fue diseñado para expresar fórmulas matemáticas, y aunque en ocasiones se emplea para aplicaciones comerciales, es aún el lenguaje que más se usa para problemas científicos, de ingeniería y matemáticos.

Hardware. Parte física del ordenador, es decir, aquellas partes que podemos tocar, los componentes propiamente dichos: un disco duro, un lector de cd, una placa base, etc.

Hemisferio boreal. Mitad de la superficie de la esfera terrestre, dividida por el Ecuador o un meridiano: el Ecuador delimita los hemisferios austral y boreal.

Interconexión. Enlazar o conectar diversos dispositivos entre sí, como lo ocurrido en una computadora personal y sus periféricos.

Interfaces. Conexión e interacción entre hardware, software y el usuario. El diseño y construcción de interfaces constituye una parte principal del trabajo de los ingenieros, programadores y consultores. Los usuarios "conversan" con el software. El software "conversa" con el hardware y otro software. El hardware "conversa" con otro hardware. Todo este "diálogo" no es más que el uso de interfaces.

Invariante de clase. Es todo aquello que debe cumplir siempre cada clase. Por ejemplo, si se definiera la "clase Persona" se tendría el siguiente invariante: "una Persona siempre tiene nombre, apellido y documento de identidad. No existen dos personas con el mismo documento de identidad".

IP. Acrónimo de (Internet Protocol) (Protocolo Internet). IP es un protocolo de internetworking que provee servicios, sin conexión, a través de múltiples redes de conmutación de paquetes.

Iterativa. Repetición de una secuencia de instrucciones o eventos. Por ejemplo, en un lazo de programa, una iteración se produce una vez a través de las instrucciones del lazo.

JAVA. Lenguaje de programación de uso generico con orientación hacia paginas web, Java fue diseñado en 1990 por James Gosling, de Sun Microsystems, como software para dispositivos electrónicos de consumo. Curiosamente, todo este lenguaje fue diseñado antes de que diese comienzo la era World Wide Web, puesto que fue diseñado para dispositivos electrónicos como calculadoras, microondas y la televisión interactiva.

LISP. Acrónimo de (List Processing) (Procesamiento en Lista). Lenguaje de programación para ordenadores o computadoras, orientado a la generación de listas, desarrollado en 1959-1960 por John McCarthy y usado principalmente para manipular listas de datos.

Mainframe. Palabra para referirse a las grandes computadoras. Un ejemplo típico sería la arquitectura 390 de IBM. Es decir, máquinas capaces de administrar muchas terminales y unidades periféricas (Ver: Periférico). Originalmente, mainframe no era sino el armario metálico que contenía la unidad central de los grandes ordenadores.

Método. Corresponde a cada una de las funciones que puede llevar a cabo un objeto, p.ej. crearse, destruirse, dar su identificación, etc.

Modelo estático. En la notación UML, el modelo estático corresponde al diagrama donde se muestran todas las clases definidas para la aplicación, indicando para cada clase sus atributos y métodos, así como las relaciones que tiene con las demás clases.

Modelo dinámico. Corresponde al con junto de casos de uso de la aplicación.

OCR. Reconocimiento Optico de Caracteres. Es una tecnología que permite leer un documento impreso y transformarlo en un texto digital para trabajar en un computador.

Objeto. Elemento que cuenta con características propias y comportamiento particulares. Bajo un enfoque orientado a objetos, un objeto es cualquier cosa que puede ser identificada plenamente en el mundo.

OLE. Acrónimo de (Object Linking and Embedding) (Concatenación y Embebimiento de Objetos). Estándar desarrollado por Microsoft para permitir incrustar objetos pertenecientes a otras aplicaciones. El concepto de OLE (Object Linking and Embedding) evolucionó hacia los estándares COM (Component Object Model).

OMG. Acrónimo de (Object Management Group) (Grupo de Administración de Objetos). Grupo Técnico responsable de la definición de estándares para CORBA (Common Object Request Broker Architecture). Las alternativas a CORBA en el mercado son DCOM (Distributed Component Object Model) de Microsoft, y RMI (Remote Method Invocation) que es parte de Java.

OMT. Acrónimo de (Object Modeling Technique). Técnica de modelación de Objetos propuesta por Rumbaugh.

OOA. Acrónimo de (Object Oriented Analysis). Método "Análisis Orientado a Objetos" propuesta por Coad y Yourdon.

OOAD. Acrónimo de (Object Oriented Analysis and Design). Método "Análisis y diseño Orientado a Objetos" propuesta por Booch.

OOSA. Acrónimo de (Object Oriented Systems Análisis). Método "Análisis de Sistemas Orientado a Objetos" propuesto por Shaler y Mellor.

OOSE. Acrónimo de (Object Oriented Software Engineering). Método "Ingeniería de Software Orientado a Objetos" propuesto por Jacobson.

Omnipresente. Que está presente en todas partes a la vez; ubicuo.

Ontológico, -ca. Parte de la metafísica que trata del ser en general y de sus propiedades trascendentales.

Ontologismo. m. fil. Doctrina del filósofo italiano Gioberti (1801-1852), que aspira a explicar el origen de las ideas mediante la adecuada intuición del Ser absoluto.

Ordenador. Ver: Computadora Personal.

Panacea. Remedio o solución capaz de solventarlo o arreglarlo todo.

Paradigma. (gr. parádeigma, modelo)
m. Ejemplo que sirve de norma, esp. de una conjugación o declinación.
Conjunto virtual de elementos de una misma clase gramatical, que pueden aparecer en un mismo contexto.

PASCAL. Lenguaje de programación desarrollado por Niclaus Wirth alrededor de 1970. Su uso es frecuente en la formación de programadores. Pascal ha sido ampliamente adaptado para programación de sistemas y aplicaciones.

PC. Ver Computadora Personal.

Periféricos. Conjunto de tecnologías Hardware que permiten conocer con más detalle el funcionamiento interno y el comportamiento de algunos componentes del PC. Por una parte tenemos los dispositivos externos o periféricos que se intercomunican con nuestro ordenador.

PERL. Lenguaje de programación de propósito general, que surgió con la idea de simplificar las tareas de administración de UNIX, pero que por su potencia se emplea en la actualidad en multitud de casos. la versión de PERL que llegó a ser más conocida fue la versión 4.

Polimorfismo. Es una característica presente en la programación.

Portabilidad. Capacidad que tiene una aplicación de ejecutarse en diferentes plataformas de hardware y software.

Pragmático, -ca. adj. Relativo a la acción y no a la especulación.
adj.-s. Relativo al pragmatismo.

Pragmatismo. (gr. pragma, acción, asunto)
m. Doctrina filosófica que considera al hombre, no como un ser pensante, sino como un ser práctico.

PROLOG. Acrónimo de (Programming In Logic) (Programación Logica). Lenguaje de programación que tiene su aplicación práctica en el desarrollo de software destinado a Sistemas Expertos.

Prueba piloto. Prueba de la aplicación realizada con un grupo representativo de la población objetivo.

RAD. Acrónimo de (Rapid Application Development Tools) (Herramientas para Desarrollo Rápido de Aplicaciones) Herramientas utilizadas generalmente para el desarrollo rápido de Front Ends. Entre las herramientas RAD de mayor uso se encuentran: Visual Basic, Visual C++, Power Builder, Visual Cafe, Delphi, etc.

RTF. Acrónimo de (Revision Task Force) (Destacamento de fuerzas de Revisión). Conjunto de personas cuya responsabilidad es la de generar revisiones menores de la especificación UML.

Realimentación. Hecho de volver a alimentar, establecer un contacto bidireccional entre dos o más entidades.

RMI. Acrónimo de (Remote Method Invocation) (Invocación con Método Remoto) Estándar para manejo de objetos distribuidos que es parte de Java. Las alternativas en el mercado para RMI son DCOM (Distributed Component Object Model) de Microsoft y CORBA (Common Object Request Broker Architecture) del Object Management Group.

ROM. Acrónimo de (Read Only Memory) (Memoria de Solo Lectura). Se le denomina memoria de sólo lectura, debido a que los datos almacenados en ella no pueden ser modificados, y generalmente está ubicada en la tarjeta del sistema.

SO. Ver Sistema Operativo.

Sistema Operativo. Un sistema operativo es un programa (o conjunto de programas) de control que tiene por objeto facilitar el uso de la computadora y conseguir que ésta se utilice eficientemente.

Semántico. Es el conjunto de reglas que proporcionan el significado de una sentencia o instrucción de cualquier lenguaje de programación.

Sintaxis. Conjunto de normas que gobiernan la asociación de variables de programación para formar las instrucciones.

SIMULA I. Lenguaje de programación, diseñado en Europa (en el NCC, Norwegian Computer Center) para describir sistemas y programar simulaciones discretas. La primera referencia escrita a SIMULA data del 5 de enero de 1962, en una carta de Kristen Nygaard al investigador francés Charles Salzmann, en el que se le llamaba "compilador de Monte Carlo".

SIMULA 67. Lenguaje de programación de propósito general que surge de los modelos de simulación en 1967. Parte del éxito de este lenguaje se debe a que se realizaron implementaciones para ordenadores IBM, DEC, Control Data y UNIVAC.

SMALLTALK. Lenguaje de Programación con raíces del lenguaje SIMULA desarrollado dentro del Grupo de Investigación de Aprendizaje en el Centro de Investigación de Xerox en Palo Alto a comienzos de los 70'. Las raíces del Sistema Operativo Apple y Windows de Microsoft son de Smalltalk.

Software. Conjunto de instrucciones que las computadoras emplean para manipular datos. El Software es un conjunto de programas, documentos, procedimientos, y rutinas asociados con la operación de un sistema de computo. Distinguiéndose de los componentes físicos llamados hardware.

SW. Ver Software.

UML. Acrónimo de (Unified Modeling Language) (Lenguaje de Modelaje Unificado). Lenguaje de modelación, que permite de manera estándar modelar los datos de determinada aplicación, con una notación para expresar los datos (atributos, métodos), las relaciones entre los mismos y el con junto de requerimientos que pueden ser satisfechos en la aplicación.

UNIX. Sistema operativo para ordenadores tipo mainframe. Soporta gran número de usuarios y posibilita la ejecución de distintas tareas de forma simultánea (multiusuario y multitarea). Su facilidad de adaptación a distintas plataformas y la portabilidad de las aplicaciones (está escrito en lenguaje C) que ofrece hacen que se extienda rápidamente.

Usuario. Nombre otorgado a las personas que utilizan las aplicaciones o sistemas informáticos.