UNIDAD 1

Conceptos basicos de la Programación Orientada a Objetos.

1.- **INTRODUCCION**.

1.a) El Problema...

Si nos preguntamos: ¿Para qué nos sirve el software?. podriamos enumerar algunas de las actividades en las que participa.

a) Controla las reservaciones en una cafetería.
b) Controla las reservaciones en lineas aéreas.
c) Controla la generación de energía en una Planta eléctrica.
d) Controla la operación de los elevadores en cientos de edificios.
e) Controla la navegación de barcos a través de rutas marítimas.

Para poder realizar correctamente todas estas actividades, se requieren enormes sistemas de elevada complejidad, los cuales para su diseño deberán ser entendidos y modelados a detalle, de tal forma que la información puede fluir en ellos de una manera precisa y hacia otros sistemas también complejos.

Los Programas de Aplicación son complejos, ya que tienen la misión de modelar el mundo real, En nuestros dias la mayoria de las aplicaciones son extensas y complejas para ser diseñadas por un solo individuo, ésto nos lleva a un tiempo de desarrollo grande, el cual puede ser mayor que su vida útil, gran problema.

En un proyecto grande el problema crece ya que al agregarle un nuevo elemento, el cual, al interactuar con elementos ya existentes puede ocasionar **bugs**, los cuales se multiplican exponencialmente al agregar nuevos elementos, adicional a ésto, un **bug**, puede generar **bugs** adicionales.

Bajo este panorama, el mantenimiento de grandes sistemas se transforma en un verdadero dolor de cabeza; ya que al implementar pequeñas modificaciones éstas, son dificiles de aislar, y cuando se produce un **bug**, no sabemos a ciencia cierta donde esta ocurriendo, sume a ésto, que la persona que hizo la modificación, NO conoce a detalle todo el sistema, se convierte en un verdadero laberinto.

Analizando el software bajo estas circunstancias, su desarrollo y actualización se convierte en una cuesta arriba, ya que para agregarle flexibilidad y mejoras, pequeñas o substanciales, involucra un gran costo y esfuerzo. Es por ello que los programadores proceden con cautela, al hacer modificaciones, realizan un gran número de pruebas, lo que involucra aumento en el tiempo de desarrollo.

1.b) Como nos puede ayudar la abstracción.

El mundo real es un lugar complejo. Entre más detalle de él observamos, más complejidad encontramos, aquí salta una pregunta, ¿Cómo entender el mundo real?, para modelarlo, en nuestro caso, traducirlo a un programa.

1.b.1) La abstracción, como un proceso mental Natural.

La gente tipicamente entiende el mundo real, construyendo modelos mentales de partes de él, lo que le permite interactuar con él. Un Modelo Mental es una vista simplificada del comportamiento de las cosas, de tal manera que podamos interactuar con ellas. En esencia, este proceso de construir modelos, es el mismo para el diseño de software, con la diferencia de que el desarrollo de software produce un modelo que puede ser manipulado por una computadora.

Un Modelo Mental, deberá ser más simple que el sistema que representa, sino no se podra usar. Por ejemplo, considere un mapa como el modelo de un territorio, para que sea útil, deberá ser más simple que el territorio. Si incluyera cada detalle y fuera del tamaño del territorio, nunca cumpliría su propósito. Un mapa es útil y nos ayuda porque es abstracto y solamente contiene aquellos detalles del territorio que nos interesan; así por ejemplo, un mapa de carreteras nos dice la ruta de una localidad a otra, y su distancia, un mapa topográfico modela el contorno de un territorio, cada uno es útil para un propósito específico, porque omite los demás detalles.

De la misma forma que un mapa deberá ser más pequeño que el territorio que representa e incluir solo cierta información seleccionada, así un modelo Mental abstracto de un sistema deberá contener ciertas características e ignorar aquellas que son irrelevantes para su comprensión.
Este poceso de comprensión es psicologicamente necesario y natural; la abstracción es crucial para entender el mundo.

La abstracción es esencial para un ser humano y es una potente herramienta cuando se maneja complejidad. Considere por ejemplo, el hecho de memorizar números, supongamos que solo es capaz de memorizar 7 digitos, sin embargo si los agrupa y los relaciona con un número de teléfono, ha relegado los digitos individuales a un estado de detalle de bajo nivel y creado un nivel alto de abstracción en donde los 7 números se han convertido en una entidad, usando este mecanismo ahora puede memorizar 7 números de teléfono, con lo cual ha aumentado su habilidad para lidear con la complejidad. La agrupación de varias entidades, en una, es un mecanismo poderoso al servicio de la abstracción.

1.b.2) La historia de abstracción en software.

La abstracción es la llave que nos permite diseñar buenos programas. Las metas de software de aplicación son más ambiciosas que antes ya que ahora contamos con la ayuda de la abstracción.

En los primeros dias de la computadora, los programadores introducian instrucciones binarias a las computadoras, directamente, manipulando interruptores sobre su panel de control. Los mnemonicos del Lenguaje Ensamblador fueron una forma de abstracción para evitar a los programadores la necesidad de tener que recordar secuencias de bits, de los cuales se componen las instrucciones.

El siguiente paso en el nivel de abstracción fué la creación de instrucciones de programación. Una instrucción de programación se compone por un conjunto de aquellas instrucciones primitivas, ahora agrupadas en macro-instrucciones con un nombre definido. Una macro-instrucción realiza varios procesos en una computadora.

Los Lenguajes de Programación de Alto Nivel, les permiten a los programadores no estar en contacto con la arquitectura de una máquina dada, para lograr este propósito cierto conjunto de instrucciones se reconocieron como universalmente útiles, escribiendo los programas solo en términos de ellas. Cada instrucción invoca una serie de instrucciones de máquina, dependiendo de la máquina donde se haya compilado el programa. Este tipo de abstracción les permite a los programadores escribir software de propósito general, sin preocuparse en que tipo de máquina va a correr.

Secuencias de instrucciones de alto nivel se pueden agrupar en procedimientos e invocarlos mediante una sentencia, aumentando con ello, el nivel de abstracción.

En la programación estructurada se encuentra el uso de abstracciones de control tales como lazos o sentencias if-then, incorporadas en lenguajes de alto nivel.

Más recientemente, la abstracción de tipos de datos fué permitida a los programadores, lo que les facilitó escribir código, sin tener en cuenta la manera en que los datos son representados, al hacer ésto el programador no tiene que meterse con la representación de los datos, programando a nivel de estructura. Por ejemplo un conjunto puede definirse como una colección de elementos no duplicados. Usando esta definición, podemos especificar las operaciones que pueden realizarse sobre el conjunto, sin especificar si sus elementos son almacenados en un arreglo, en una lista enlazada o en otro tipo de estructura de datos.

El diseño Orientado a Objetos, descompone el sistema en **objetos**, siendo éste el componente básico del diseño. El uso de objetos nos permite manejar mecanismos poderosos de abstracción. Antes de explicar como el objeto nos permite manejar la abstracción, contestaremos algunas preguntas basicas.

1.c) ¿Qué son los Objetos?.

La teoría de programación Orientada a Objetos, intenta manejar los problemas de complejidad del mundo real, abstrayendo sus caracteristicas y encapsulandolas dentro de un objeto. La definición de estos objetos se convierte en un problema de estructurar características y actividades.

Una manera de programar es empezar dividiendo la información en 2 entidades; funciones y datos. Es una forma interesante de ver el mundo, y podemos hacer grandes cosas usando esta idea. Esta manera de programar, la cual puede ser llamada procedural, primero define todas las actividades a realizar y entonces descompone cada trabajo en trabajos más pequeños hasta llegar a nivel de comando del lenguaje. Este tipo de diseño le permite a la programación procedural casi inmediatamente poder empezar con la implementación de los programas: definiendo las etapas que componen cada función y las particularidades de los datos a ser operados sobre ellas.

Otra forma es utilizando programación Orientada a Objetos, la cual se inicia con un enfoque un poco más abstracto.

La primera cuestión es la finalidad del programa, el qué, no el cómo, El objetivo es determinar los objetos y sus conexiones, para hacer ésto, es necesario descubrir las operaciones a realizarse y la información que resulta de dichas operaciones. Entonces se divide responsabilidad entre operaciones e información para Objetos. Cada Objeto debe conocer como realizar sus propias operaciones y recuerda su propia información. El diseño Orientado a Objetos descompone un sistema en entidades que conocen sus reglas de operación dentro del sistema, Saben como son ellas mismas, ésto involucra funciones y datos. Los Objetos conocen ciertos datos acerca de ellos mismos, (una persona conoce el color de su pelo y ojos) los Objetos saben como hacer ciertas funciones, (una persona sabe como realizar una compra en un supermercado, o como realizar su trabajo). Un Objeto sin embargo, no conoce las propiedades de todo el sistema. (una persona no sabe el trabajo de todas las demás o su color de pelo y ojos)

1.c.1) Encapsulamiento.

Un Objeto se conoce a si mismo, pero no conoce a los demás, algunos datos del sistema residen dentro de un objeto y otros datos residen dentro de otros Objetos, así mismo, algunos objetos pueden realizar ciertas funciones, y otros, otras funciones.

El hecho de agrupar en un Objeto tanto los datos como las operaciones que afectan sus datos, se le llama; **encapsulamiento**. El encapsulamiento maneja la complejidad inherente a los problemas del mundo real, distribuyendo dicha complejidad en objetos individuales.

El conocimiento encapsulado dentro de un objeto, se puede ocultar del mundo exterior. Como una consecuencia, un objeto se ve diferente desde dentro, que desde fuera del mismo, (de la misma manera que una persona tiene una personalidad interna

y otra que muestra al mundo) el Objeto tiene una cara pública, la cual muestra a otras entidades dentro del sistema, esta cara pública consiste de qué es, puede hacer y decir, por lo tanto las otras entidades conocen del Objeto, como pueden interactuar con él.

Por otra parte, los objetos tienen su parte privada, la parte privada de un Objeto es; como es y como realiza su trabajo. Como realiza sus operaciones o manejo de información no interesa a otras partes del sistema. Este principio se conoce como ocultamiento de información. Usando este principio, los Objetos son libres de modificar su lado privado, sin afectar al resto del sistema.

1.c.2) Antropomorfismo (senejanza a lo humano)

El antropomorfismo dentro del diseño Orientado a Objetos puede ser una ayuda para conceptualización. El diseño Orientado a Objetos ve el mundo como un sistema de agentes de cooperación y colaboración. (Como los sistemas dentro del ser humano) El trabajo dentro de un Sistema Orientado a Objetos se realiza por un Objeto, él cual envia un requerimiento a otro Objeto para que realice una de sus operaciones, revele alguna de su información o ambas cosas. Este primer requerimiento puede iniciar una cadena de requerimientos a otros Objetos.

Considerando este espíritu antropomórfico de los sistemas, podemos decir que el software nace, vive y muere. Examinemos el ciclo de vida del software.

1.d) ¿Qué es el ciclo de vida del Software?.

La vida del software se inicia con la especificación de sus requerimientos, su vida sigue creciendo con la implementación y prueba, si pasa las pruebas, se gradúa al estatus de producto. Una vez que llega al usuario y este encuentra **bugs**, debera entrar a mantenimiento, el creciente aumento de necesidades del usuario hace que se le agreguen nuevas características creando nuevas versiones, las cuales deberán pasar por; diseño, implementación y prueba.

La gente inevitablemente crece, se hace vieja y muere, sin embargo el software puede ser rejuvenecido, podemos pensar en la vida del software literalmente como un ciclo. Podemos imaginar su vida como una espiral, en la cual, el ciclo de creación puede ser repetido infinidad de veces.

1.d.1) Ciclo del software.
Tradicional y Orientada a Objetos.

Considerando que un ciclo de vida de un software llega hasta su graduación, podemos observar el espiral para la programación procedural. En este caso la mayor cantidad de tiempo se la lleva la implementación del diseño y en el caso del diseño Orientado a Objetos, la mayor cantidad de tiempo es consumido en el Diseño, ésto se debe a que en el Diseño Orientado a Objetos, se debe poner especial atención en este renglón para conseguir un diseño facil de reusar, mantener y modificar. Al hacer un

cuidadoso diseño nos permite entender con mayor claridad el problema y por consiguiente la implementación se realizará más rapidamente.

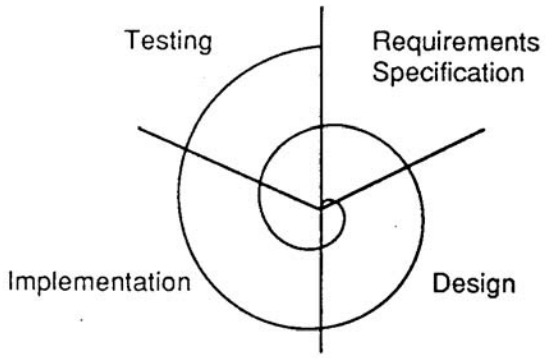

Figura # 1.
Diseño Tradicional

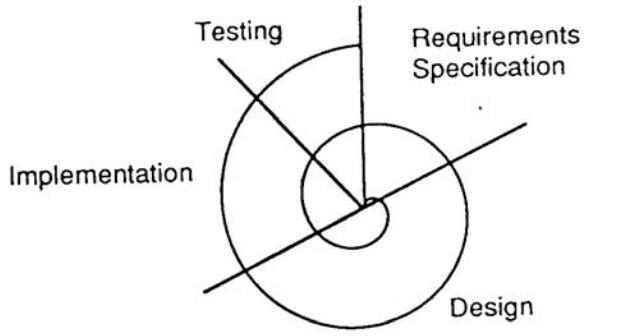

Figura # 2
Diseño Orientado a Objetos

El Diseño Orientado a Objetos, tiende a crear software robusto, él cual puede ser facilmente reutilizado, refinado, probado, mantenido y extendido.

El software es **reusado**, cuando es empleado como una parte de otro programa para el cual no fué originalmente diseñado.
El software es **refinado**, cuando es usado como base para el desarrollo de otro programa.
El software es **probado**, cuando se determinan sus características operativas, las cuales deben concordar con sus especificaciones.
El software es **mantenido**, cuando se encuentran errores y éstos son corregidos.
El software es **extendido**, cuando a un programa ya existente se le adicionan nuevas características para mejorar su funcionalidad.

Es necesario hacer notar que las herramientas del diseño Orientado a Objetos no garantiza las características arriba mencionadas, sin embargo, si ayudan a realizar un diseño más cuidadoso, y éste por consecuencia traerá dichas características.

1.d.2) Las 4 etapas del ciclo de vida.

l) **Especificación de requerimientos.**

Una buena especificación de requerimientos, nos describe:

a) Que es capaz y que no es capaz de hacer el software.

b) Proporciona la importancia relativa de cada característica en relación a las demás.
c) Limitaciones con respecto al mundo real.

II) **Diseño.**

El diseño final deberá consistir de:

a) Un sistema de objetos que cubran los requerimientos.
b) Una descripción de los aspectos públicos de los Objetos.
c) Los patrones de comunicación entre los Objetos.

III) **Implementación.**

Usando el diseño Orientado a Objetos, el diseñador adquiere un mejor entendimiento del problema antes de iniciar su codificación.

Los Lenguajes Orientados a Objetos proporcionan un excelente soporte para el diseño Orientado a Objetos, sin embargo, no son estrictamente necesarios, implementaciones realizadas con cualquier otro lenguaje de programación pueden sacar beneficio de este proceso de diseño. Idealmente, el lenguaje usado deberá proporcionar facilidades para hacer cosas que deberán hacerse y dificultad para aquellas cosas que no deberán ser hechas.

IV) **Prueba.**

Al realizar pruebas del sistema, es necesario considerar que en el diseño Orientado a Objetos, el sistema esta formado por entidades las cuales pueden ser aisladas y probadas de manera individual, un error puede ser más facilmente encontrado y tratado.

Similarmente las cuidadosas especificaciones de las interfases entre entidades nos permiten realizar pruebas más faciles para encontrar discrepancias entre la salida de un componente y la entrada requerida por otro. Estas cuidadosas especificaciones de las interfases requieren una comprensión total de las responsabilidades de cada componente.

* **Mantenimiento.**

Un programa se puede mantener si se entiende pero si no se comprende, cambios hechos en él, van a producir **bugs** que van a empeorar su operación.

Como un diseño Orientado a Objetos usa encapsulamiento y ocultamiento de información y los patrones de comunicación son más rigidos, pueden se más facilmente entendibles, lo que dá como consecuencia:

* Una mejor detección del lugar donde radica el problema.
* La mejor determinación de ramificaciones del problema.

* **Refinamiento y extensión.**

Si es más fácil entender el software, será más fácil construir sobre él. Si el software es bien diseñado con una rigurosa consistencia, pueden extenderse sus interfaces y agregar nuevas entidades.

1.e) ¿Qué partes de un sistema pueden ser reusadas?.

El encapsulamiento nos permite construir entidades que por sus características pueden ser reusables, podemos hablar basicamente de 3 tipos:

1) Componentes.

2) Frameworks

3) Aplicaciones.

1) **Componentes.**

Entre estos podemos enumerar; listas, arreglos, cadenas. Más recientemente, Radio buttons y check boxes.

El enfoque que se debe dar al diseñar componentes es; hacer su desarrollo de forma general, de tal manera que puedan se componenetes de tantos sistemas como sea posible. Para las aplicaciones que hacen uso de ellos, los componentes son considerados como cajas negras, y no es necesario para el desarrollador de la aplicación que conozca la implementación de dichos componentes.

2) **Frameworks.**

Este tipo de código representa esqueletos estructurados de programas, los cuales deberán ser rellenados para tener un programa de aplicación completo. Como ejemplos podemos mencionar:

a) Un sistema de ventanas.
b) Sistema de simulación.

El objetivo al diseñar Frameworks, es el hacerlos refinables. La interface(s) al resto de la aplicación debera se preciso, tanto como sea posible. Los Frameworks son vistos por el programador como cajas blancas, por lo que cuando se hace uso de ellas, se deberá entender perfectamente su estructura.

3) **Aplicaciones.**

Las aplicaciones son programas completos; por ejemplo, procesadores de palabras, hojas de calculo, una calculadora, un sistema de nóminas, etc.

El objetivo al desarrollarlos es hacerlos claros, de tal forma que puedan ser mantenidos.

Idealmente, una aplicación se desarrolla en base a frameworks y componentes, al hacer ésto, de la aplicación creada pueden resultar, extensiones del frameworks o el desarrollo de nuevos componentes.

El desarrollo de aplicaciones deberán hacer un uso ingenioso de componentes y frameworks, para tener las características y ser compatibles con los sistemas actuales. Para que una aplicación sea fácil de extender y mantener en el futuro, la selección inicial en el diseño tendrá un fuerte impacto a la hora de hacer cambios y extensiones.

Si una aplicación específica resulta potencialmente útil, se debe pensar seriamente en convertirla en un componente, con lo que el código se hace reusable.

Como un **Resumen** a esta introducción tenemos:

Un sistema complejo puede verse con un enfoque de cosas y procesos; hay suficientes razones para aplicar la Descomposición Orientada a Objetos, con lo cual vemos al mundo como un conjunto de objetos que colaboran a la realización de actividades de alto nivel.

El Diseño Orientado a Objetos es el método que nos lleva a una descomposición Orientada a Objetos; El Diseño Orientado a Objetos define una notación y proceso para construir sistemas complejos de software, y ofrecer un rico conjunto de modelos lógicos y físicos mediante los cuales podamos analizar diferentes aspectos del sistema considerado.

2.- **EVOLUCION DEL MODELO DE OBJETO.**

2.a) Tendencia en la ingeniería de software.

Si observamos hacia atras la tendencia en el desarrollo de software, vemos 2 tendencias:

* El giro que dió la programación, desde pequeños programas a grandes sistemas.
* La evolución de los lenguajes de programación.

Wagner realizó una clasificación de los lenguajes de alto nivel más populares en generaciones de acuerdo a sus características.

* Primera generación. (1954 - 1958)

- FORTRAN I Expresiones matematicas.
- Algol 58 Expresiones matematicas.
- Flowmatic Expresiones matemáticas.
- IPL V Expresiones matematicas.

* Segunda Generación. (1959 - 1961)

- FORTRAN II Subrutinas, compilación separada.
- ALGOL 60 Estructura en bloques, tipos de Datos.
- COBOL Descripción de Datos, manejo de archivos

* Tercera Generación. (1962 - 1970)

- PL/1 FORTRAN + ALGOL + COBOL
- ALGOL 68 Sucesor riguroso de ALGOL 60
- Pascal Sucesor simple de ALGOL 60
- Simula Clases y abstracción de datos.

* La Generación de Transición. (1970 - 1980)

Se desarrollaron muchos lenguajes pero pocos perduraron
- Ada Sucesor de ALGOL 68, Pascal y Simula.
- CLOS* Mejoras de Lisp, LOOPS y Flavors.
- C++ Derivado de C y Simula.

* CLOS: Common Lisp Object System

El C++ es el Lenguaje de nuestro mayor interes, es la clase de lenguaje al cual llamamos; Basado en Objetos, Orientado a base de Objetos o Programación Orientada

a Objetos, y es el que mejor soporta la Descomposición Orientada a Objetos del software.

Vamos a analizar algunas topologías de desarrollo de software. Estas topologías estan basadas en las caracteristicas de los lenguajes, asi tenemos:

1) Primera Topología, de la primera generación y de los primeros lenguajes de la segunda generación.

En la figura # 2, vemos la Primera topología, la cual nos muestra la construcción física del software y la interrelación entre sus partes. En la figura podemos observar que para lenguajes como el FORTRAN y COBOL la construcción del bloque de todas las aplicaciones es el subprograma (párrafo en COBOL). Las aplicaciones escritas en estos lenguajes tienen una estructura plana, consistente solo de datos globales y subprogramas.
NOTA: Las flechas en la figura indican dependencias de los subprogramas sobre varios datos.

Durante el diseño, uno puede separar logicamente diferentes conjuntos de datos unos de otros, aunque no hay principios de diseño para ello. Un error en una parte del programa puede tener un efecto en cadena desbastador a través del resto del sistema, ya que los datos globales estan disponibles para todos los subprogramas. Cuando se hacen modificaciones a un sistema es dificil mantener la integridad del diseño original.

Un programa escrito en estos lenguajes es muy difícil de seguir y mantener.

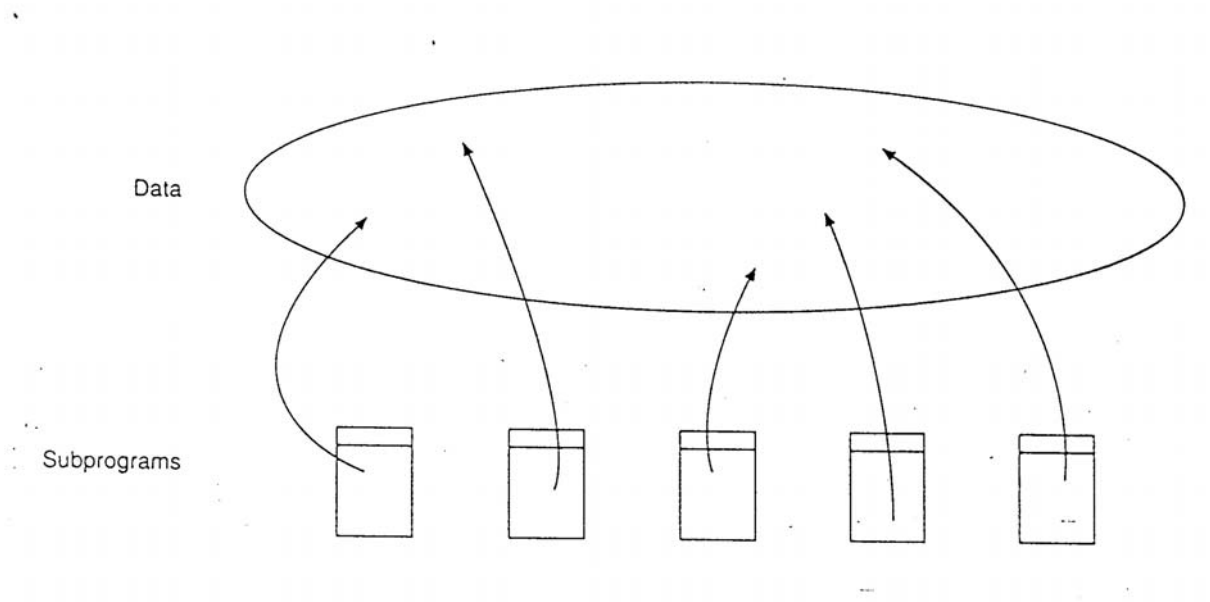

Figura # 2, Primera topología.

2) Segunda topología, de los últimos lenguajes de la segunda generación y de los primeros lenguajes de la tercera generación.

Data de mediados de los 60's se desarrolla la primera abstracción del software, llamada "La Abstracción Procedural". Los subprogramas fueron inventados por 1950 pero en ese entonces no se utilizaron como abstracción, sino solo como una forma de ahorrar esfuerzo.

El punto de vista de utilizar los subprogramas como un mecanismo de abstracción, trae consigo tres consecuencias importantes:

* Los lenguajes fueron inventados con un mecanismo de pase de parámetros.
* Se consideró la fundamentación de la programación estructurada y el alcance y visibilidad de las declaraciones.
* Aparecieron métodos para el Diseño Estructurado, ofreciendo lineamientos para diseñar y construir grandes sistemas usando subprogramas como la base de construcción de bloques.

Esta topología se muestra en la Figura # 3.

3) Tercera Topología, de los últimos lenguajes de la Tercera Generación.

Empezando con el FORTRAN II, y siguiendo con otros lenguajes de la Tercera Generación, nos proporcionan nuevas herramientas para desarrollar grandes proyectos. Entre estas herramientas esta el poder dividir un programa en partes y manejarlas independientemente. Hablamos de la compilación por separado, en donde cada módulo contiene datos y subprogramas. Ahora se consideró a cada módulo como un importante mecanismo de abstracción. En algunos de estos lenguajes se carecía de verificación de tipos, por lo que errores de pase de parámetros podian se detectados solo hasta la corrida del programa.

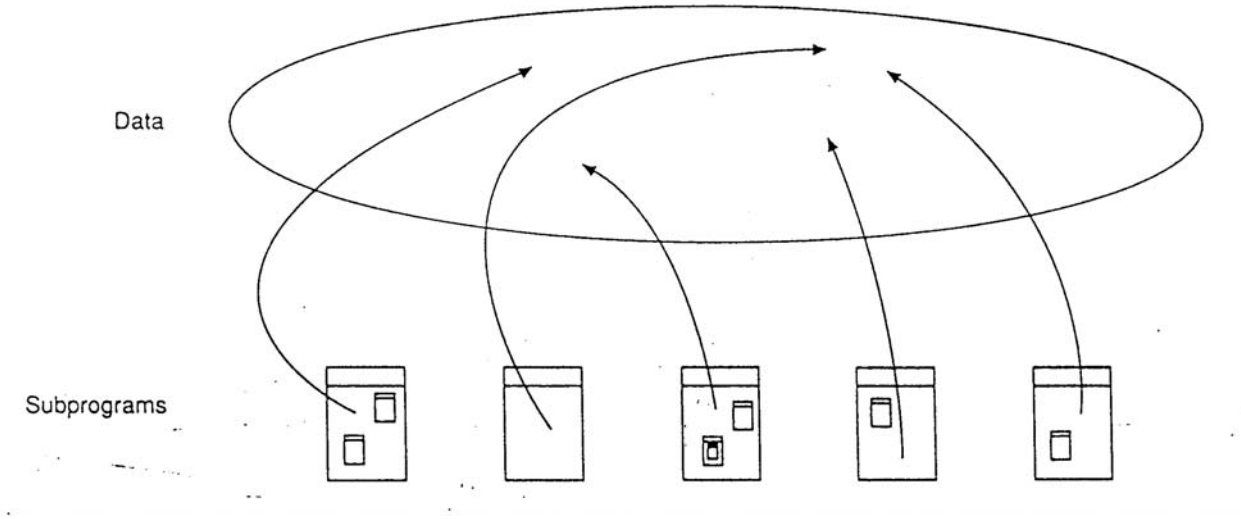

Figura # 3, Segunda Topología.

La tercera topología se muestra en la Figura # 4.

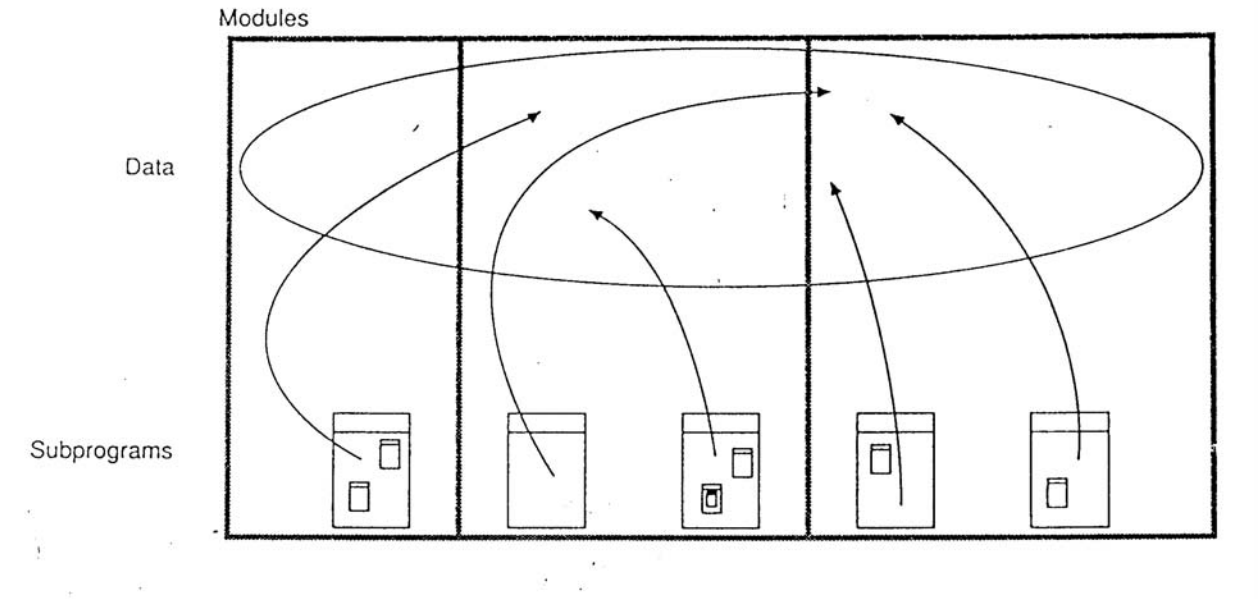

Figura # 4, Tercera Topología.

La Cuarta Topología, llamada Basada en Objetos y de Programación Orientada a Objetos. La importancia de la abstracción de datos es claramente explicada por Shankar, que dice lo siguiente: La naturaleza de la abstracción que puede ser llevada a cabo a través del uso de procedimientos, es útil para la descripción de operaciones abstractas, pero no así para la descripción de Objetos abstractos, esto es un problema

serio, ya que en muchas aplicaciones, la complejidad de los datos como objetos al ser manipulados contribuyen substancialmente en el aumento de complejidad del problema. Este planteamiento, lleva consigo 2 consecuencias importantes:

* Emergen métodos de diseño de manipulación de datos, los cuales nos brindan una metodología para resolver el problema de abstracción de datos en lenguajes orientados a algoritmos.
* La teoría desprecia el concepto de tipo, el cual eventualmente encuentra su definición en lenguajes como Pascal.

La conclusión de estas ideas tiene su respuesta, primero, en Lenguajes como Simula, y recientemente lleva al desarrollo de Lenguajes como Smalltalk, Objet Pascal, C++, CLOS y Ada, a los cuales se les llama, Lenguajes Basados en Objetos o Lenguajes Orientados a Objetos. En la Figura # 5 se muestra la Topología para estos lenguajes, para desarrollos pequeños o de tamaño medio. El bloque de estos lenguajes es el **módulo**, el que representa una colección lógica de clases y objetos en lugar de subprogramas como lo hacian los lenguajes anteriores.

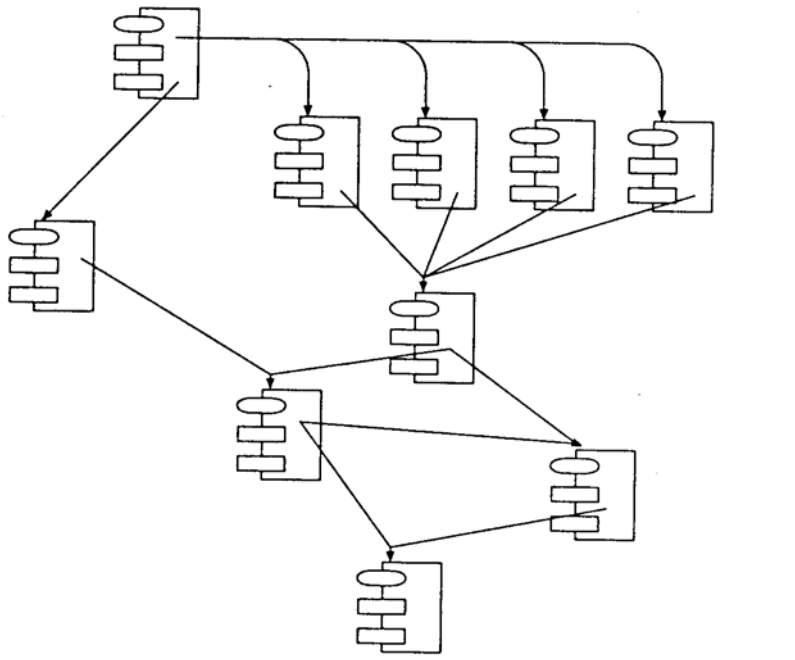

Figura # 5, Cuarta Topología, Para sistemas chicos o medianos.

Su conceptualización se maneja de otra manera; Si los procedimientos y las funciones son **verbos**, y los datos son **sustantivos**, un programa orientado a procedimientos es organizado alrededor de **verbos**, mientras un programa orientado a objetos es organizado alrededor de **sustantivos**. Por esta razón la estructura física de una aplicación de tamaño medio se representa como una gráfica no como un árbol, él

cual corresponde tipicamente a lenguajes orientados a algoritmos. Adicionalmente hay muy pocos o ningún dato global, en vez de ésto, los datos y las operaciones se encapsulan de tal forma que la construcción del bloque lógico de estos sistemas, no tiene un algoritmo de desarrollo grande, pero sí la definición de clases y objetos.

En sistemas muy grandes, encontramos que; las clases los objetos y los módulos, no son suficientes para descomponer el sistema, afortunadamente el modelo de Objeto puede ser escalado hacia arriba, encontrando agrupamientos de abstracción (clusters) construidos en capas uno encima del otro. A cualquier nivel dado de abstracción nos encontramos una colección de objetos que contribuyen a realizar una actividad de alto nivel. Si vemos hacia el interior de cualquier agrupamiento (cluster) observamos otro conjunto de agentes de colaboración.

En la Figura # 6 podemos observar la Cuarta Topología para sistemas muy grandes.

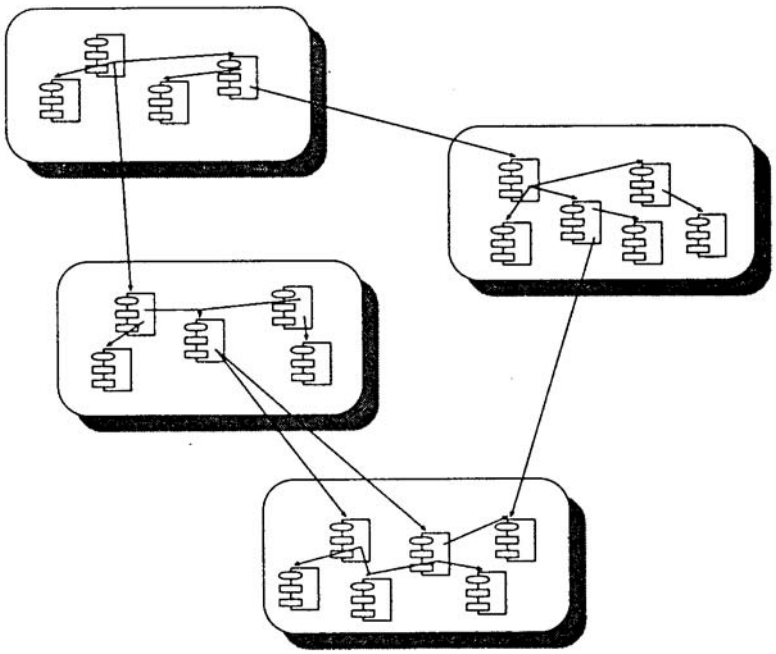

Figura # 6, Cuarta Topología.
Para sistemas muy grandes.

2.b) Fundamentación del Modelo de Objeto.

El método de Diseño Estructurado establece como guía de desarrollo para la construcción de sistemas complejos el uso de algorítmos como la parte fundamental de la construcción de sus bloques.

El método de Diseño Orientado a Objetos explota como guía de desarrollo la potencia del desarrollo basado en objetos y los Lenguajes de Programación Orientados a Objetos, usando las clases y objetos como base de construcción de sus bloques.

Actualmente, el modelo de objeto, esta influenciado por varios factores, no siendo uno de estos la programación orientada a objetos. El modelo de Objeto es un concepto de la ciencia computacional, aplicable no solo a los lenguajes de programación, sino también a, interfaces de usuario, base de datos, bases de conocimiento y a la arquitectura de computadoras, la razón para un campo de aplicación tan extenso como éste, es tan simple como que una orientación a objetos nos ayuda a manejar la complejidad inherente en diferentes campos de sistemas.

El Diseño Orientado a Objetos entonces es un desarrollo evolutivo, no un desarrollo revolucionario; no es un descubrimiento sorpresivo, es una evolución continua, construida sobre bases firmes. Desafortunadamente la mayoria de los programadores en la actualidad son entrenados, ya sea formal o informalmente en los principios del diseño estructurado. Ciertamente muchos buenos ingenieros han desarrollado y liberado sistemas de software útiles, usando estas técnicas, sin embargo, hay limites al nivel de complejidad que podemos manejar utilizando solo descomposición algoritmica; cuando lleguemos a este nivel debemos de voltear la mirada a la descomposición orientada a objetos.

Si usamos lenguajes como C++ o Ada, de la manera tradicional; como lenguajes orientados a algoritmos, sin pensar en un Diseño Orientado a Objetos, no tiene sentido, es similar a estar usando un Lenguaje "C" o Pascal, es como querer matar hormigas a cañonasos.

Informalmente podemos definir un objeto como una entidad tangible que presenta un comportamiento bien definido.

Stefik y Bobrow definen al objeto como una entidad que combina las propiedades de procedimientos y datos ya que los objetos realizan operaciones y guardan su estado actual.

La idea básica sobre los objetos es que nos sirven para unificar las ideas de algoritmos y abstracción de datos. Jones dá un poco más de claridad a esta idea, haciendo notar que en el modelo del objeto, se pone especial enfasis en resaltar las características de los componentes del sistema físico o abstracto a ser modelado por el programador... Los objetos tienen una integridad, la cual no puede ser violada. Un objeto solo puede cambiar estado, conducta, etc. Los objetos existen con propiedades invariantes que los caracteriza y define su comportamiento. Por ejemplo, como similitud podemos pensar en un elevador, este dispositivo solo tiene movimiento hacia arriba y hacia abajo, pero nunca tendrá un movimiento lateral.

2.c) Algunas definicionnes.

1) ANALISIS ORIENTADO A OJETOS. (**OOA**)

Tradicionalmente la técnica del análisis estructurado fué bien tipificada, se enfocó sobre el flujo de datos en un sistema. El análisis orientado a objetos enfatiza la construcción de modelos del mundo real, usando un punto de vista del mundo orientado a objetos. Puede ser definido como sigue:

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

De acuerdo a la definición de OOA, OOD y OOP, el resultado de un Análisis Orientado a Objetos nos puede servir como el modelo desde el cual podemos partir para un Diseño Orientado a Objetos y el resultado del Diseño Orientado a Objetos puede entonces ser usado como punto de partida para completar la implementación de un sistema usando métodos de Programación Orientada a Objetos.

2) DISEÑO ORIENTADO A OBJETOS. (**OOD**)

Mientras el enfasis en métodos de programación estriba en el uso adecuado y eficiente de un mecanismo particular del lenguaje, El enfasis en los métodos de diseño estriba es la estructuración adecuada y eficiente de un sistema complejo. Puede ser definido como sigue:

Es un método de diseño que involucra el proceso de descomposición orientada a objetos y una notación para describir tanto modelos lógicos como físicos, así como estáticos y dinámicos del sistema bajo diseño.

Encontramos dos partes importantes en la definición.

1) Nos lleva a una descomposición orientada a objetos.

2) Usa diferentes notaciones para expresar diferentes modelos lógicos (clases y estructuras de objetos) y físicos (módulos y arquitectura de procesos) de un sistema.

Es necesario no confundir el Diseño Orientado a Objetos y el Diseño estructurado, el primero usa abstracción de clases y objetos para estructurar logicamente un sistema, y el segundo usa abstracción de algoritmos.

3) PROGRAMACION ORIENTADA A OBJETOS. (**OOP**)

Es un método de implementación en el cual los programas son organizados como una colecc6n cooperativa de objetos, donde cada uno representa una instancia de una clase y cuyas clases son todas miembros de una jerarquía de clases unidas via relaciones de herencia.

En esta definición encontramos tres partes importantes:

* Programación Orientada a Objetos, usa objetos, no algoritmos como la parte fundamental de construcción de bloques.

* Cada objeto es una instancia de alguna clase.

* Las Clases se relacionan por medio de relaciones de herencia.

Si en un programa se omite cualquiera de los puntos anteriores, no se considera como un programa orientado a objetos. Especificamente programando sin considerar herencia podemos decir que es una programación con tipos abstractos de datos.

Debido a esta definición algunos lenguajes son orientados a objetos y otros no lo son. Stroustrup sugiere que el término lenguaje orientado a objetos deberá representar un lenguaje que cuenta con mecanismos que soportan bien el estilo de programación orientado a objetos... Un lenguaje soporta adecuadamente un estilo de programación si cuenta con facilidades que hacen uso adecuado de dicho estilo. Un lenguaje no soporta una técnica si se requiere un esfurzo excepcional para escribir programas dentro de dicha técnica.

Cardelli y Wegner dicen que un lenguaje es orientado a objetos si satisface los siguientes requerimientos.

1) Soportan objetos que son abstracción de datos con una interface de operadores y un ocultamiento de su estado local.

2) Los objetos cuentan con un tipo asociado [**clase**]

3) Los tipos [**clase**] pueden heredar atributos de supertipos [**superclases**].

Para que un lenguaje soporte herencia significa que es posible expresar "genero de" relaciones entre tipos, así podriamos decir; que una Rosa Roja es un género de flores y una flor es un género de Planta. Si un lenguaje no proporciona un soporte directo de herencia, no es un lenguaje orientado a objetos, Cardelli y Wegner le llaman a tales lenguajes; Basados en Objetos en vez de Orientado a Objetos. Bajo esta distinción Smalltalk, Object Pascal, C++ y CLOS son Orientados a Objetos y Ada es Basado en Objetos.

3.- **ELEMENTOS DEL MODELO DE OBJETO.**

3.a) Esencia del Paradigma de Programación.

Jenkins y Glasgow observaron que la mayoria de los programadores trabajan con un lenguaje y usan un estilo de programanción, programan en un paradigma forzado por el lenguaje que usan. Nunca ven caminos alternativos en la solución de un problema en cuanto a un estilo de programación más apropiado al problema que tienen entre manos.

Bobrow y Stefik definen un estilo de programación como "una manera de organizar los programas sobre la base de algunos modelos conceptuales de programación y un lenguaje apropiado para realizar programas escritos un limpio estilo. mencionan cinco estilos de programación,los cuales se listan abajo, mencionando el genero de abstracción que usan.

* Orientado a Procedimientos	Algoritmos.
* Orientado a Objetos	Clases y Objetos.
* Orientado a Lógica	La meta, se expresa en un cálculo de predicado.
* Orientado a Reglas	Reglas if-then
* Orientado a restricciones	Relaciones invariantes.

No existe un solo estilo de programación que sea mejor para todos los tipos de aplicaciones. Por ejemplo, el estilo Orientado a Reglas es mejor para el diseño de una Base de Conocimientos. El estilo Orientado a Objetos, es mejor para un amplio rango de aplicaciones de desarrollo de software en donde la complejidad es el elemento dominante.

Cada estilo de programación trata el problema de manera diferente, partiendo de su propia base conceptual. El estilo Orientado a Objetos su base conceptual es el modelo de Objeto, él cual involucra cuatro elementos primordiales:

* Abstracción.
* Encapsulamiento.
* Modularidad.
* Jerarquía.

Por primordial, entendemos, que el modelo de Objeto sin cualquiera de estos cuatro elementos no es Orientado a Objetos.

Involucra también tres elementos menores:

* Tipo
* Concurrencia.
* Persistencia.

Por menores, entendemos, que cada uno de estos elementos es útil pero no una parte esencial del Modelo de Objeto.

Sin esta base conceptual podemos programar en lenguajes como Smalltalk, Objet Pascal, C++, CLOS o Ada pero nuestro diseño tiene la estructura manejada por lenguajes como FORTRAN, Pascal o C.

*** Abstracción.**

El significado de Abstracción. La abstracción es una de las formas en que nosotros como seres humanos lideamos con la complejidad.

Hoare sugiere que: "La abstracción parte del reconocimiento de similitudes entre ciertos objetos, situaciones o procesos en el mundo real, y la decisión se concentra sobre estas similitudes e ignora en ese momento las diferencias."

Shaw define la abstracción como: "Una descripción simplificada o especificación simplificada de un sistema que enfatiza algunos de los detalles o propiedades del sistema mientras ignora otras. Una buena abstracción es aquella que enfatiza que detalles le son significativos para el lector o usuario y suprime detalles que son, por el momento innecesarios o diversificantes.

Combinando estos puntos de vista podemos establecer nuestra propia definición de abstracción.

> La abstracción nos proporciona las características esenciales de un objeto que lo distinguen de los otros generos de objetos y nos brinda un concepto bien delimitado de él, relativo a la prespectiva del observador.

*** Encapculamiento.**

El significado de Encapsulamiento. La abstracción de un objeto deberá preceder a la decisión acerca de su implementación. Una vez que una implementación es seleccionada, ésta deberá tratarse como un secreto de la abstracción y ocultarla de su entorno. Como sugiere Ingalls, "Ninguna parte de un sistema complejo deberá depender de los detalles internos de otra parte".

Encapsulamiento y abstracción son conceptos complementarios: La abstracción se enfoca sobre la parte exterior de un objeto y el encapsulamiento, se enfoca sobre la parte interior de un objeto.

Liskow, va todavía más lejos al sugerir que "Para que la abstracción funcione, la implementación deberá ser encapsulada". En la práctica ésto significa que cada clase

deberá tener dos partes: una interface y una implementación. La **interface** de una clase contiene solo su punto de vista externo. La **implementación** de una clase comprende la representación de la abstracción así como el mecanismo que lleva a cabo el comportamiento deseado.

El concepto de encapsulamiento usado en el contexto Orientado a Objetos, no difiere esencialmente de la definición del diccionario. Se refiere a la construcción de una cápsula, en este caso una barrera conceptual, alrededor de una colección de cosas. En el caso del Diseño de software, funciona como una ayuda de conceptualización y abstracción, nos permite dibujar un círculo alrededor de ideas y operaciones. (información y operaciones)

Lo anterior nos permite establecer la definición de encapsulamiento.

> Encapsulamiento es el proceso de ocultar los detalles de un objeto que no contribuyen a sus características esenciales.

*** Modularidad.**

Como observa Myers, "El acto de particionar un programa en componentes individuales puede reducir su complejidad en algún grado... La idea de particionar un programa es útil por esta razón y más importante aún, resulta la idea de que al particionar un programa se crea un conjunto de partes bien documentadas dentro de él. Estas porciones o interfaces son invaluables en la comprensión del programa. En lenguajes como C++, el módulo es un bloque separado y por ello nos garantiza un conjunto separado de desiciones de diseño. En estos lenguajes las clases y los objetos forman la estructura lógica de un sistema. Ponemos estas abstracciones en módulos para generar la arquitectura física del sistema. Especialmente para grandes aplicaciones en las cuales se pueden tener cientos de clases el uso de módulos es esencial para ayudar en el manejo de la complejidad.

Liskow establece que: "La modularidad consiste en dividir un programa en módulos, los cuales pueden ser compilados por separado, pero los cuales tienen conexión con otros módulos". Los lenguajes que soportan el módulo como un concepto separado, distinguen también entre la interface del módulo y su implementación, entonces podemos decir que modularidad y encapsulamiento van de la mano. Cada lenguaje soporta la modularidad en diferentes maneras. Por ejemplo para C++, la modularidad no es más que archivos separados. La práctica tradicional en el C++, es poner las interfaces de los módulos en archivos con extensión (.h) llamados archivos de cabecera, y la implementación de los modulos se coloca en archivos (.c), las dependencias entre estos archivos pueden ser manejadas usando la directiva #include.

Hay varias técnicas que nos pueden ayudar a realizar una modularización inteligente de clases y objetos. Como observa Britton y Parnas: "La finalidad de la descomposición en módulos es la reducción del costo del software, creando módulos que puedan ser diseñados y revisados independientemente...". Cada estructura de un módulo deberá ser simple de tal manera que pueda ser comprendido con relativa facilidad, asi mismo deberá ser posible cambiar la implementación de un módulo sin la

necesidad de conocer la implementación de los otros módulos y sin afectar su comportamiento.

La modularidad deberá consistir de construir módulos que sean coherentes (agrupando logicamente abstracciones relacionadas) y libertad de acoplamiento (minimizando las dependencias entre módulos). Desde esta prespectiva podemos formularnos una definición de modularidad.

> Modularidad es la propiedad de un sistema que le permite ser descompuesto en un conjunto de modulos acoplados de una forma coherente y libre.

*** Jerarquía.**

El significado de Jerarquía. La abstracción es un excelente mecanismo, pero en la mayoria de las aplicaciones, exceptuando las más triviales, nos encontramos más abstracciones que podemos comprender a la vez. El encapsulamiento nos ayuda a manejar esta complejidad ocultadndo la parte interna de nuestras abstracciones. La modularidad también nos ayuda proporcionando una manera para agrupar abstracciones relacionadas. Esto no es suficiente. Un conjunto de abstracciones frecuentemente involucra jerarquias, identificando estas jerarquias en nuestro diseño podemos entender y simplificar el problema.

Podemos definir Jerarquía como sigue:

> La jerarquía es una clasificación u ordenación de la información.

Herencia es la forma más importante de jerarquía, y como lo acabamos de notar, es un elemento esencial en sistemas Orientados a Objetos. Basicamente la herencia define relaciones entre clases, donde una clase comparte la estriuctura o comportamiento definido en una o más clases, (se le llama herencia simple o herencia multiple respectivamente) se dice que hereda ciertas características.

La herencia entonces representa una jerarquía de abstracción en donde una subclase hereda de una o más superclases. Tipicamente una subclase agrega o redefine la estructura y características de la superclase.

Definición de los elementos menores.

*** tipo**

El significado de tipo. El concepto de un tipo deriva primeramente de la teoría de abstracción de tipos de datos. Deutsch sugiere "Un **tipo** es una caracterízación precisa de propiedades las cuales van a ser compartidas por una colección de entidades.". Para nuestro propósito tipo y clase son términos similares.

Con lo anterior podemos proponer la siguiente definición.

tipo es la definición hecha a la clase de un objeto de tal manera que objetos de diferente tipo no puedan ser intercambiados, o por lo menos, existen muchas restricciones.

La declaración de tipos trae un número importante de beneficios.

+ Sin verificación de tipos, un programa puede fallar de manera misteriosa en tiempo de ejecución.

+ En la mayoría de los sistemas el ciclo de; Edición, compilación y depuración es tedioso, por lo que nos conviene detectar la mayor cantidad de errores.

+ La declaración de tipos nos ayuda a documentar los programas.

+ La mayoria de los compiladores pueden generar código objeto más eficiente si se declaran los tipos.

El **polimorfismo** es un concepto en la teoría de tipos en la cual un solo nombre (semejante a una declaración de variable) puede denotar objetos de muy diferentes clases los cuales son relacionados por una superclase común. Cualquier objeto denotado por su nombre es capaz de responder a un conjunto común de operaciones.

Lo opuesto a polimorfismo es monomorfismo, él cual se encuentra en todos los lenguajes fuertemente tipados y estaticamente limitados como Ada.

El **polimorfismo** se dá cuando interactuan las características de herencia y enlace dinámico

El **polimorfismo** nos permite reconocer y explotar similaridades entre diferentes clases de objetos, nos permite que diferentes generos de objetos puedan responder al mismo mensaje, reconociendo la diferencia entre el nombre del mensaje y el método. Un objeto envia un mensaje: si el receptor implementa un método con la misma firma, responderá. Son posibles diferentes respuestas; de esta forma diferentes métodos responderan a instancias de diferentes clases.

Un ejemplo tipico sería la impresión de un entero, un flotante o una cadena por una función llamada imprime(), en realidad van a existir tres funciones llamadas imprime(), una para cada caso, siendo escogida la función de acuerdo al argunento que le fué pasado.

El **polimorfismo** representa una de las cualidades más sobresalientes de la Programación Orientada a Objetos después de la abstracción.

*** Concurrencia**

El significado de concurrencia. Para un cierto género de problemas un sistema automatizado puede manejar varios eventos simultamenamente, donde a cada evento o programa le podemos llamar enlace de control.

Mientras el enfoque de la Programación Orientada a objetos se basa en la abstracción de datos, encapsulamiento y herencia, la concurrencia se enfoca sobre abstracción de procesos y sincronización. El objeto es un concepto que unifica estos dos puntos de vista, cada objeto (representación de una abstracción del mundo real) puede representar un enlace de control (abstracción de proceso) diferente. A estos objetos se les llama **activos** en un sistema basado en Diseño Orientado a Objetos podemos conceptualizar el mundo como consistente de un conjunto de objetos cooperativos, donde algunos de ellos estan activos y sirven como centros de actividad independiente.

En base a lo anterior podemos proponer la siguiente definición:

> Concurrencia es la propiedad que distingue un objeto activo de uno que no esta activo.

*** Persistencia.**

Un objeto en software ocupa alguna cantidad de espacio y existe durante una cantidad de tiempo dada. Podemos mencionar algunos ejemplos de persistencia de objetos.

+ Resultados transitorios de evaluación de expresiones.

+ Variables locales activadas en procedimientos.

+ Variables globales y elementos en el heap.

También podemos mencionar la persistencia a través del espacio, la cual se dá en sistemas con multiples procesadores, en donde el objeto puede ser movido de máquina en máquina.

Podemos resumir la persistencia como sigue:

> La persistencia es la propiedad de un objeto a trevés de la cual trasciende en tiempo y/o espacio.

4.- **BENEFICIOS DEL MODELO DE OBJETO.**

1) Nos permite explotar la potencia de lenguajes basados en objetos y de programación Orientada a Objetos.

2) Podemos reusar no solamente parte del software, sino todo el diseño.

3) Los sistemas son construidos sobre plataformas estables, siendo más facilmente actualizados, lo que significa que los sistemas pueden evolucionar con el tiempo, en vez de ser abandonados y rediseñados completamente, como consecuencia de algún cambio en sus requerimientos.

5.- **ALGUNAS DEFINICIONES IMPORTANTES.**

agente Un objeto que puede operar sobre otros objetos y también puede ser operado por otros objetos.

clase Conjunto de objetos que comparten una estructura común y tienen la misma conducta, los términos clase y tipo son intercambiables, a excepción de cuando se enfatiza la jerarquía de clases.

cardinalidad El número de instancias que puede tener una clase.

cliente Un objeto que usa los recursos de otro, ya sea, operando sobre él o haciendo referencia a su estado.

interface Vista externa de una clase, objeto o módulo, la cual enfatiza su abstracción, mientras oculta su estructura y proceder.

método Una operación sobre un objeto, definida como parte de la declaración de una clase, todos los métodos son operaciones, pero no todas las operaciones son métodos. Los términos; mensaje, método y operación son normalmente intercambiables.

objeto Un objeto tiene estado, conducta e identidad. La estructura y conducta de objetos similares se encuentra definida en su clase común. Los términos instancia y objeto son intercambiables.

servidor Un objeto que nunca opera sobre otros objetos, y si es operado por otros objetos.

subclase Una clase que hereda de una o más clases.

superclase La clase de la cual otras clases heredan.