Introducción a la Programación Orientada a Objetos

El paradigma de objetos se constituyó en la metodología de desarrollo de software más utilizado en la industria del software. La Tecnología Orientada a Objetos (TOO) no solo se aplica a los lenguajes de programación, sino que también se ha propagado a los métodos de análisis y diseño y a otras áreas tales como las bases de datos y las comunicaciones.

La OO permitió mejoras de amplio alcance en el diseño, desarrollo y mantenimiento del software, ofreciendo una solución a largo plazo a los problemas y preocupaciones que han existido desde el comienzo en el desarrollo de software, como ser:

- la falta de portabilidad del código y reusabilidad
- código difícil de modificar
- ciclos de desarrollo excesivamente largos y plazos no cumplidos
- técnicas de codificación no intuitivas

Un lenguaje orientado a objetos ataca estos problemas.

Si bien el paradigma de objetos tiene características bien definidas, no está libre de la influencia de técnicas anteriores, tales como la programación estructurada, los conceptos de abstracción y de tipos abstractos de datos (TAD).

Las características sobresalientes de las dos técnicas de desarrollo más conocidas son:

- **Programación Estructurada**

  Se emplea desde principios de la década de los 70. Es aún uno de los métodos más utilizados en el campo de la programación. Centrado en la descomposición funcional. Se basa en el lema "Divide y vencerás": descompone de manera sucesiva un problema en subproblemas. Los programas se componen de bloques (procedimientos y funciones) que pueden o no comunicarse entre sí.

  En un programa estructurado, lo importante es ***qué se hace*** y en segundo plano se piensa en ***sobre qué estructuras se hace*** (es decir, los procedimientos son más relevantes que las estructura de datos). El programador se concentra en el procesamiento, en el algoritmo requerido para llevar a cabo el cómputo deseado, dando mayor énfasis al diseño de procedimientos y relegando la organización de la información.

  Presenta problemas en desarrollos realizados por muchas personas, o que estén geográficamente distribuidas. Es compleja la coordinación y organización entre programadores en la creación de aplicaciones de media y gran envergadura, dado que la comunicación entre los módulos no es eficiente.

  Existe reutilización basada en bibliotecas de funciones, sin embargo éstas siempre tienen dependencia de los datos.

  Es difícil modificar y extender los programas, pues suele haber datos compartidos por varios subprogramas, que introducen interacciones ocultas entre ellos. Esto hace difícil reutilizar los programas.

- **La Programación Orientada a Objetos**

  Con esta metodología, cobra relevancia ***sobre qué se hace*** y no ***qué se hace*** (las estructuras de datos son más relevantes que los procedimientos). Los desarrollos se organizan alrededor de los datos manipulados, y no alrededor de las funcionalidades. Esta forma de construir el programa resulta mucho más eficaz puesto que en la vida de un programa los elementos más estables son los datos.

  En este paradigma, el software se empaqueta en forma de clases. Éstas pueden reutilizarse posteriormente en otros sistemas de software. En la comunidad del software, se cree que el factor más importante que afecta al futuro del desarrollo del software es la reutilización. Reutilizar clases existentes cuando se crean nuevas clases y programas es un proceso que ahorra tiempo y esfuerzo. Además, permite crear sistemas más confiables, ya que las clases existentes a menudo han pasado por un proceso extenso de prueba y depuración. La reutilización se realiza también a través de la derivación de nuevas clases a partir de clases ya existentes: las bibliotecas de clases predefinidas, que poseen los distintos lenguajes, y las definidas por los propios usuarios.

  El paradigma de objetos no es totalmente nuevo, sino que nació de la mano de los problemas de simulación, y de lenguajes como Flavors y Smalltalk, en la década de los ‘70. Sin embargo, su auge comenzó recién en los ‘90, siendo dos los aspectos que lo han impulsado especialmente:

  - Su uso en aplicaciones comerciales, saliendo del ámbito académico.
  - La aparición de metodologías avanzadas de desarrollo orientado a objetos.

La complejidad del software

En una definición simple se puede decir que software es el conjunto de instrucciones que permite al hardware de la computadora desempeñar trabajo útil. Un sistema de software no es sólo una suma de programas, y su complejidad crece más que linealmente con la cantidad de programas que contiene. Una definición clásica de sistema de software es: "conjunto de programas que, trabajando como conjunto, tienen un propósito específico y mayor a la suma de las partes".

Se puede desarrollar software para uso personal, o software industrial. El desarrollo industrial implica la construcción de productos complejos, para gran cantidad de usuarios, siendo personas distintas quienes los desarrollan o los mantienen. Estos productos deben cumplir requisitos de interfaz hombre-máquina (amigabilidad para el usuario), legibilidad, robustez, documentación, etc.

El desarrollo de este tipo de software implica algunos factores que impactan en su creación:

- Número de personas involucradas en el desarrollo
- Tamaño (líneas de código)
- Tiempo dedicado a la elaboración del producto
- Recursos para la construcción del software

Conforme un producto/sistema de software posea más cantidad de algunos de estos elementos, el producto será más complejo.

La complejidad del software se deriva de dos elementos principales:

1) **Complejidad del dominio del problema**: representada por
   - Cantidad muy grande de requisitos
     - Ejemplos: Sistema electrónico de un avión multimotor, sistema de conmutación para teléfonos celulares, robot autónomo
     - Requisitos implícitos: Facilidad de uso, óptimo rendimiento, costo accesible, fiabilidad.
   - Problemas de comunicación entre usuario y desarrollador
     - Hablan diferente lenguaje: usuario tiene dificultad para expresar con precisión sus necesidades
     - Realizan suposiciones distintas sobre la naturaleza de la solución
     - En casos muy extremos, el usuario tiene una débil idea de lo que necesita que contenga el software
   - Cambio de requisitos durante el desarrollo del sistema: Se modifican reglas en el dominio del problema (Ej: normas del Estado, adecuación a la competencia, etc)
   - Criticidad del sistema: Software para equipos médicos, transacciones bancarias, software relacionado al lanzamiento espacial, para atención de desastres naturales, etc.
2) **Dificultad para gestionar el proceso de desarrollo**: El desarrollo del software industrial requiere gran cantidad de escritura de código, y probablemente se reutilicen módulos de software existente. Este último punto se vuelve cada vez más importante, dado que también es creciente la necesidad de obtener resultados en el menor tiempo posible. Estas características implican:
   - Software de gran tamaño
     - Inicialmente, los programas se desarrollaban en lenguaje ensamblador, y contaban en general con un centenar de líneas
     - En la actualidad, se utilizan lenguajes de alto nivel con hasta millones de líneas de código, dado que cada vez existe un mayor grado de complejidad en las aplicaciones demandadas
     - Se generan cientos y a veces miles de módulos, lo cual significa que se complica la integración de los distintos módulos
   - Imposibilidad de que un solo desarrollador pueda comprender todas las sutilezas del sistema y fundamentalmente llevarlo a cabo en el tiempo requerido.
     - Necesidad de un equipo de desarrolladores (un equipo tan pequeño como sea posible).
     - Más miembros implica una comunicación más compleja entre los integrantes del equipo, y por tanto, una coordinación más difícil, agravada si el equipo está disperso geográficamente.
   - Ciclos de desarrollo largos

Todas estas características representan un reto a la gestión del proyecto, y el director del equipo se convierte en la persona clave, que debe lograr la concreción del proyecto en tiempo y forma.

La crisis del software

Al principio, la producción de software era un problema secundario. Las computadoras eran muy pocas y el costo de fabricarlas era muy superior al de programarlas.

Pero a medida que avanzaba el tiempo, éstas se fueron haciendo más económicas. Esto desencadenó un aumento en la demanda de productos software, que creció en mayor proporción que la capacidad de producirlos y mantenerlos. Esta situación es la que se conoce como **crisis del software.** Esta crisis se profundiza por la creciente complejidad del software. Los desarrolladores de sistemas y los usuarios fueron conscientes de esta crisis. El desarrollo de software a menudo se hacía fuera de plazo y de presupuesto, de forma no fiable y, sobre todo, no cumplía los requerimientos desde el punto de vista del usuario.

Un estudio de la época del Standish Group[1] hecho sobre 352 compañías de software, donde se estudiaron más de 8.000 proyectos de software, revelaron lo siguiente:

- El 31% de los proyectos de software fueron cancelados antes de terminar (US$81 billones perdidos)
- El 53% de los proyectos tuvieron un costo 189% mayor de lo estimado.
- El 9% de los proyectos se terminaron a tiempo y dentro del presupuesto (compañías grandes).
- El 16% de los proyectos se terminaron a tiempo y dentro del presupuesto (compañías pequeñas).

A raíz de estos datos, se les preguntó a las empresas sobre las causas de estos problemas. Las tres principales razones expuestas fueron las siguientes:

- Falta de información por parte de los usuarios (12.8%)
- Especificaciones y requerimientos incompletos (12.3%)
- Cambios en las especificaciones y requerimientos (11.8%)

Del análisis de éstos y otros estudios, se ha llegado a la siguiente conclusión: si se asigna una unidad de costo de **uno** al esfuerzo requerido para detectar y reparar un error durante la etapa de codificación, entonces el costo para detectar y reparar un error durante la etapa de requerimientos es entre **cinco y diez** veces menor. Más aun, el costo de detectar y reparar un error durante la etapa de mantenimiento es **veinte** veces más alto.

¿Cómo cambiar esa imagen desoladora? Era evidente la necesidad de hallar nuevas y mejores técnicas de producción de software. El problema es causado por la complejidad inherente al software. Por tanto, lo más acertado es estudiar cómo se organizan los sistemas complejos en otras disciplinas. Existen muchos sistemas extraordinariamente complejos realizados por el hombre: el túnel bajo el Canal de la Mancha, la administración de grandes organizaciones como Microsoft o General Electric.

Ingeniería del Software

A mediados de la década de los ‘60, aparece el concepto de Ingeniería de Software, como un enfoque más metodológico para el desarrollo de software. En primer lugar surgieron las llamadas técnicas estructuradas de análisis, diseño y programación, que recién cobraron relevancia a mediados de la década de los ‘80.

Según el estándar IEEE**, la ingeniería de software es:**

"(1) la aplicación de un método sistemático, disciplinado y cuantificable al desarrollo, operación y mantenimiento de software, esto es, la aplicación de la ingeniería al software"

"(2) el estudio de los métodos de (1)",

La ingeniería del software estudia la aplicación de técnicas formales de desarrollo a la construcción de software, buscando obtener productos de calidad, con la menor relación costo/eficiencia

Los problemas que presenta el software se pueden resumir en:

- Alta complejidad
- Muy cambiante

1 Conocida firma de investigaciones que realiza bianualmente un estudio sobre el número de proyectos IT que culminan exitosamente y los que se quedan en el camino.

- Dificultad para hacerlo fiable
- Dificultad para probarlo
- Dificultad para la especificación de requisitos

Las nuevas metodologías suponen un enfoque integral del problema, abarcando todas las fases, que en su mayoría no se consideraba en los desarrollos tradicionales. **El propósito de aplicar metodologías es lograr procesos predecibles y repetibles que mejoren la productividad y la calidad**.

En particular se busca la reducción de costos y plazos, así como la calidad del producto final. Para ello se realiza un tratamiento sistemático del proceso de desarrollo.

El proceso de desarrollo de software "es aquel en que las necesidades del usuario son traducidas en requerimientos de software, estos requerimientos transformados en diseño y el diseño implementado en código, el código es probado, documentado y certificado para su uso operativo"[2].

La formalización del proceso de desarrollo se define como un marco de referencia denominado ciclo de desarrollo del software o ciclo de vida del software. Se puede describir como, "el período de tiempo que comienza con la decisión de desarrollar un producto software y finaliza cuando se ha entregado éste". Este ciclo, por lo general incluye las fases:

- requerimientos
- diseño
- implantación
- prueba
- instalación
- aceptación

El ciclo de desarrollo software se utiliza para estructurar las actividades que se llevan a cabo en el desarrollo de un producto software. A pesar de que no hay acuerdo acerca del uso y la forma del modelo, cualquiera de ellos sigue siendo útil para la comprensión y el control del proceso.

La ingeniería de software tiene varios modelos de desarrollo en los cuales se puede apoyar para la construcción de software, de los cuales se destacan los siguientes, por ser los más utilizados y los más completos:

- Modelo en cascada (ciclo de vida clásico)
- Modelo de mejora iterativa
- Modelo de prototipos

Factores de Calidad del software

El proceso de ingeniería de software se define como "un conjunto de etapas parcialmente ordenadas con la intención de lograr un objetivo, en este caso, la obtención de un producto de software de calidad" [Jacobson 1998].

Hay muchas cualidades de software que son deseables. Algunas de éstas se aplican tanto al producto como al proceso llevado a cabo para producir el producto. El concepto de calidad varía según las expectativas de los involucrados en el proceso (stakeholders[3]): el **usuario** quiere que el producto de software sea confiable, eficiente y fácil de usar. El **productor** del software quiere que sea verificable, mantenible (que se pueda mantener), portable y extensible. El **director** del proyecto de software quiere que el proceso de desarrollo del software sea productivo y fácil de controlar.

Cualidades Externas Versus Internas

Las cualidades externas son visibles a los usuarios del sistema; las cualidades internas son aquellas que incumben a los que desarrollan el sistema. En general, a los usuarios del software sólo les preocupan las cualidades externas, pero son las cualidades internas (las que se vinculan con gran parte de la estructura del

[2] Applying UML in The Unified Process". Jacobson. 1998.

[3] Todas las personas que afectan o son afectadas por el proyecto, ya sea de forma positiva o negativa.

software) las que ayudan a los que las desarrollan a obtener las cualidades externas. En muchos casos, incluso, las cualidades se relacionan estrechamente y la distinción entre internas y externas no es nítida.

Para ser considerado software de calidad, debe cumplir con la mayoría de los siguientes requisitos:

- **Performance** (Eficiencia): Capacidad de un software para hacer un buen uso de los recursos que manipula. En la actualidad ya no se da tanta importancia al almacenamiento como recurso, pero sí cobra relevancia el tiempo de respuesta, lo que implica un eficaz uso del procesador.
- **Portabilidad** (Transportabilidad): Facilidad con la que un software puede ser transportado a diferentes ambientes físicos o lógicos, es decir, diferentes plataformas. En esto precisamente radica el éxito de Java, dado que mediante la JVM (Máquina Virtual Java), logra ser ampliamente portable
- **Integridad**: Capacidad de un software para proteger sus propios componentes contra los procesos que no tengan derecho de acceso a ellos. Este factor es de vital importancia en la actualidad, dado que muchas aplicaciones se encuentran implementadas en Internet, multiplicándose las posibilidades de acceso no deseado.
- **Facilidad de uso** (Amigabilidad/Usabilidad): Un software es fácil de utilizar si se puede comunicar con él de manera cómoda. Se debe tener en cuenta que cada día más personas sin formación informática acceden a distintas aplicaciones, y éstas deben tener una interfaz amigable.
- **Confiabilidad:** Informalmente, el software es confiable si el usuario puede contar con él. La literatura especializada sobre confiabilidad del software define a la confiabilidad en términos de comportamientos estadísticos - la probabilidad de que el software va a funcionar como debe en un intervalo de tiempo especificado.
- **Corrección**: Capacidad de los productos software de realizar exactamente las tareas definidas por su especificación. Un programa es **funcionalmente correcto** si se comporta de acuerdo a la especificación de las funciones que debería proveer (llamadas '**especificaciones de requerimientos funcionales'**).
- **Robustez**: Capacidad para funcionar incluso en situaciones anormales. Por ejemplo, cuando hay mucha carga de trabajo, o en situaciones imprevistas. Java maneja la primera situación mediante el uso de multitarea, y el segundo caso, mediante el manejo de excepciones.
- **Extensibilidad** (Evolucionabilidad): Se refiere a la facilidad que tienen los productos de adaptarse a cambios en su especificación, por ejemplo, agregar nuevas funcionalidades. Esto se logra mediante dos principios fundamentales: diseño simple y descentralización. La descentralización es una capacidad intrínseca de la OO, ya que se basa en la interacción entre clases a través de mensajes. Es decir que la responsabilidad del sistema está repartida entre las clases.
- **Reparabilidad:** Un sistema de software es reparable si permite la corrección de sus defectos, con una cantidad limitada de trabajo. En muchos productos ingenieriles, la reparabilidad es la meta mayor del diseño. Por ejemplo, los motores de automóviles se construyen con las partes que generalmente parecen fallar como las más accesibles. La reparabilidad es también afectada por el número de partes del producto. Por ejemplo, es más difícil reparar un defecto en el cuerpo de un auto monolítico que si el cuerpo está hecho de distintas partes. En este caso, se puede reemplazar una única parte más fácilmente que el cuerpo completo. Una situación análoga se aplica al software; un producto de software que consiste en módulos bien diseñados, es mucho más fácil de analizar y reparar que uno monolítico. La modularización correcta, promueve la reparabilidad permitiendo que los errores sean confinados en unos pocos módulos, haciendo más fácil la localización y remoción de la misma
- **Reutilización**: Capacidad de los productos de ser reutilizados, en su totalidad o en parte, en nuevas aplicaciones. La reusabilidad es el factor clave que caracteriza la madurez del campo industrial. Se observa un alto grado de reusabilidad en áreas como la industria automotriz y los consumos electrónicos. Por ejemplo, en la industria automotriz, el motor es a menudo reusado de modelo en modelo. Además, el auto es construido por ensamble de componentes que están altamente estandarizados y son usados para muchos modelos producidos por la misma industria. El bajo grado de reusabilidad en el software es un claro indicador que el campo debe evolucionar para conseguir el status de una bien establecida disciplina ingenieril.

- **Interoperabilidad**: Se refiere a la habilidad de un sistema de coexistir y cooperar con otros sistemas - por ejemplo, la habilidad de un procesador de palabras para incorporar un gráfico producido por un graficador, o la habilidad del graficador para graficar los datos producidos por una planilla de cálculo, o la habilidad de la planilla de cálculo para procesar una imagen escaneada por un scanner

La orientación a objetos - Surgimiento

A raíz de la aparición del concepto de Ingeniería del software, surgieron en las décadas de los '70 y '80 metodologías y técnicas de desarrollo, sin embargo éstas no eliminaron el problema de la crisis del software. La razón era que la demanda de software se incrementaba de forma tan rápida al menos como la aparición de las técnicas que ayudaban a su producción. Por lo tanto, cuando una técnica estaba resultando útil, el enorme crecimiento de la complejidad la hacía obsoleta y, fundamentalmente, no resolvía el problema de la reutilización.

El aumento de la complejidad no es un asunto trivial. Si se supone que cada nueva funcionalidad para un producto de software agrega un nuevo módulo, se puede medir el aumento de complejidad en cuanto a la interacción creciente de los módulos (los nuevos con los existentes). Esto agravado por el hecho de que generalmente los desarrolladores no intentan escribir la menor cantidad de código posible, sino que se agregan parches, en vez de mantener una alta calidad y funcionalidad.

Es posible afrontar problemas de mayor complejidad haciendo abstracciones jerárquicas de la misma. Los sistemas complejos presentan jerarquías diferentes:

- Jerarquía estructural o <<**parte de**>> (Agregación)
- Jerarquía de tipo <<**es un**>> (Herencia)

Esta separación facilita el estudio de cada parte en forma relativamente aislada.

La descomposición orientada a objetos hace precisamente eso: descomponer un problema complejo en abstracciones jerárquicas, que permiten manejar módulos en forma individual. Esto permite generar software resistente al cambio.

Otro modo de manejar la complejidad es realizando modelos (abstracciones o "representaciones de los conceptos del mundo real[4]) que estén estrechamente relacionados con el problema a resolver. Todas las técnicas usadas hasta ese momento requerían hacer una transformación del modelo del problema (o espacio del problema) hacia el modelo de la solución (o espacio de la solución). La orientación a objetos intenta acercar el modelo del problema al de la solución. La idea es crear "objetos" que representan elementos tanto del espacio del problema como de la solución, es decir, se hace una **reducción del salto de representación**. De este modo se describe el problema en términos del problema, y cuando se lee código se leen palabras que expresan el problema. Esto facilita en gran medida la comunicación y la legibilidad, y en consecuencia la mantenibilidad.

El modelo de objetos se fundamenta en principios de ingeniería robustos, como ser: abstracción, encapsulación, modularidad y jerarquía, que se conjugan de forma sinérgica, favoreciendo desarrollos productivos y la construcción de productos de calidad, objetivos perseguidos por la ingeniería del software.

En los últimos años se observa un abandono progresivo de las metodologías estructuradas, que están siendo desplazadas por las orientadas a objetos. Una de las causas más fuertes de esta situación es indirecta, y tiene que ver con la tendencia cada vez mayor a desarrollar aplicaciones distribuidas sobre Internet y la Web. En realidad, este modelo de computación distribuida no está necesariamente ligado a los objetos, pero los actores más importantes del mercado de software han decidido lanzar herramientas de desarrollo que utilizan el paradigma de objetos (Oracle/Java – Microsoft/.NET), y esto parece ser un camino sin retorno.

Por otra parte, se debe tener en cuenta que las nuevas metodologías de desarrollo de software bregan por una menor separación entre las fases de desarrollo. Tanto analistas como diseñadores y programadores deben manejar un lenguaje común, y todos deben aprender de diseño, arquitecturas y patrones. Esto justifica la importancia de estudiar la orientación a objetos como conjunto, y no abordarla sólo desde un lenguaje de programación.

[4] UML y Patrones. Craig Larman. 2003.

Los métodos modernos de diseño de software orientado a objetos incluyen mejoras entre las que están el uso de los patrones de diseño, diseño por contrato, y lenguajes de modelado (ej: UML).

Este enfoque se encuentra en una etapa avanzada de madurez como paradigma de desarrollo de sistemas de información. El Object Management Group (OMG) es un consorcio a nivel internacional que integra a los principales representantes de la industria de la tecnología de información OO. Tiene como objetivo central la promoción, fortalecimiento e impulso de la industria OO. Propone y adopta por consenso especificaciones entorno a la misma. Una de las especificaciones más importantes es la adopción en 1998 del Lenguaje de Modelado Unificado o UML (del inglés *Unified Modeling Language*) como un estándar, que junto con el Proceso Unificado (PU) están consolidando la tecnología OO.

Características de la programación orientada a objetos

Si bien hay un cierto desacuerdo sobre exactamente qué características de un método de programación o lenguaje le definen como "orientado a objetos", hay un consenso general en un conjunto básico de éstas propiedades. Los cuatro elementos o propiedades más importantes de este modelo según Booch son: abstracción, modularidad, encapsulamiento y jerarquía. Este autor sostiene que si alguno de estos elementos no existe, se dice que el modelo no es orientado a objetos. Sin embargo, existen híbridos que surgen como extensiones, y que implementan algunas de las propiedades mencionadas.

1) **ABSTRACCION:**

En un sentido amplio, el proceso de supresión de detalles respecto a un fenómeno, entidad o concepto.

Según Booch (1996), la abstracción manifiesta las características esenciales de un objeto que lo distinguen de todos los demás tipos de objetos, y proporciona así fronteras conceptuales nítidamente definidas **respecto a la perspectiva del observador**.

Es el modo de representar un objeto del mundo real como un objeto abstracto, o como la información constructiva de una clase u objeto que sirva como componente de la solución a un problema particular. Es una descripción especial simplificada, que hace énfasis en ciertos rasgos y suprime otros, **desde un punto de vista particular**. Permite ignorar aquellos aspectos que no sean relevantes al propósito actual para concentrarse más profundamente en aquellos que lo son.

La buena abstracción es aquella que logra hacer énfasis en los detalles significativos o relevantes de la solución y discrimina cualquier otra característica, **en el dominio del problema**. Con esto se consigue un mapeo de los objetos del mundo real a los objetos del sistema.

Los humanos gestionamos la complejidad a través de la abstracción. Por ejemplo, nosotros no vemos a un automóvil como una lista de partes insondables; lo vemos como un objeto bien definido con un comportamiento único: sirve para desplazarse de un lugar a otro. Esta abstracción nos permite trasladarnos en un vehículo a cualquier lugar sin que nos desborde la complejidad de lo que éste representa. Ignoramos lo que representa el motor y cómo funciona, y los detalles de funcionamiento de la transmisión o del sistema de frenos. En cambio, sí tratamos con una noción abstracta, fiable e idealizada de cómo funciona el coche en su conjunto. Tenemos una capacidad poderosa para gestionar esa abstracción de una forma jerárquica que nos permite, a nuestra voluntad, dividir en partes los sistemas complejos. Por ejemplo, un auto es una "unidad" que nos resulta útil para desplazarnos. Una vez dentro de él, podemos utilizar el volante, los frenos, la radio, los cinturones de seguridad, etc, todos módulos o componentes con una cierta funcionalidad, que colaboran con el objetivo: desplazarse.

La programación que utiliza abstracción de datos se basa en el hecho de que en un programa se deben integrar y combinar los tipos básicos de datos, como números y caracteres, para formar estructuras de datos más complejas y así representar (modelar) información del problema, dentro del computador.

Por ejemplo, para definir el objeto automóvil, hay que ser capaces de abstraer las funciones y atributos de un automóvil; es decir, hay que ser capaz de definir automóvil en términos de **qué puede hacer** y **qué características lo distinguen de otros objetos**.

Abstracción funcional: Funcionalmente, un coche puede realizar las siguientes acciones:

- Avanzar
- Parar
- Girar a la derecha
- Girar a la izquierda

Hay cosas que se sabe que los coches hacen, pero el cómo lo hacen, la implementación de *avanzar*, *parar*, *girar* (a la derecha, a la izquierda) es irrelevante desde el punto de vista del diseño. Esto es lo que se conoce como ***abstracción funcional***.

Abstracción de datos: un coche tiene las siguientes características o atributos:

- Color
- Velocidad
- Tamaño

La manera en que se almacenan o definen esos atributos, también es irrelevante para el diseño del objeto. Por ejemplo, el color puede definirse como la palabra *rojo*, o como un vector RGB (255,0,0). La forma en que el objeto almacena el atributo *color* es irrelevante para el programador. Este proceso de despreocupación de cómo se almacena el color es lo que se llama ***abstracción de datos***.

2) **ENCAPSULAMIENTO**:

Es el proceso de almacenar en un mismo módulo los elementos de una abstracción que constituyen su estructura y comportamiento. Sirve para separar la interfaz contractual de una abstracción y su implantación (Booch, 1996).

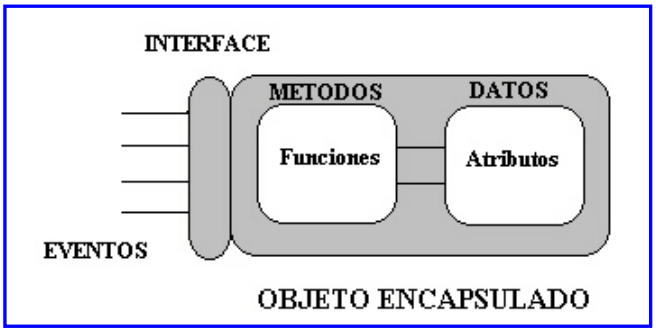
Al abstraer se enuncia comportamiento y atributos relevantes que describen a un objeto. Encapsular significa agrupar y manejar el grupo resultante (métodos y datos) como una unidad (una cápsula), y no cada parte por separado.

El encapsulamiento está muy relacionado con el concepto de abstracción. Es complementario a la misma, porque sin encapsulación no hay abstracción, ya que si no se encapsulan los componentes no se podría dar una abstracción alta del objetos al cual se hace referencia. Es decir, sirve para ocultar de la simple vista los componentes internos del objeto. Se logra por medio de la **ocultación** de la información de un objeto hacia los demás. El objeto esconde sus datos de los demás objetos y permite el acceso a los mismos mediante sus propios métodos. Esto recibe el nombre de **ocultamiento de información**.

Un objeto sólo muestra lo necesario para poder interactuar con otros objetos. La interfaz de cada componente se define de forma que revele tan poco como sea posible de sus particularidades interiores. Actúa como una "caja negra". Este concepto se utiliza ampliamente en física, electrónica e informática, y consiste en esconder todos los detalles internos del sistema que se estudia bajo una caja negra imaginaria, en los casos en que es más importante entender **qué** hace el sistema que **cómo** lo hace.

El ocultamiento de la información se traduce en determinar qué cosas deben ser vistas por los demás, lo que constituye la interfaz pública del objeto. En los lenguajes OO la encapsulación garantiza que los usuarios de un objeto sólo pueden modificar el estado del objeto e interactuar con él a través de su interface.

Interface representa la frontera y el lugar de paso en la comunicación del objeto con el mundo exterior. Ante la activación de ciertos Eventos tan solo se podrán consultar y modificar los Datos almacenados en los Atributos exclusivamente a través de las Funciones que determinen los Métodos correspondientes al Objeto en cuestión.



El encapsulamiento evita la corrupción de los datos de un objeto. Si todos los programas pudieran tener acceso a los datos de cualquier forma que quisieran los usuarios, los datos se podrían corromper o utilizar de mala manera. El encapsulado protege los datos del uso arbitrario y no pretendido.
El encapsulado también oculta los detalles de la implementación de sus métodos. Los usuarios de un objeto conocen las operaciones que pueden solicitar de un objeto, pero desconocen los detalles de cómo se lleva a cabo la operación.
El encapsulado, al separar el comportamiento del objeto de su implementación, permite la modificación de ésta sin que se tengan que modificar las aplicaciones que lo utilizan.
La gran ventaja del encapsulamiento consiste en que si un módulo u objeto cambia internamente sin modificar su interfaz, el cambio no desencadenará ninguna otra modificación en el sistema.
En general existe una fuerte relación entre todos los datos manipulados por un programa, por lo que es conveniente que esa relación esté claramente especificada y controlada, de forma que cada parte del programa "vea" sólo lo que necesita.

3) **MODULARIDAD:**

Es una propiedad que permite subdividir una aplicación en partes más pequeñas (llamadas módulos), más sencillas y manejables, cada una las cuales debe ser tan independiente como sea posible de la aplicación en sí y de las restantes partes.
La Modularidad es una partición funcional de todo el sistema. Cada módulo o parte del sistema debe contar tanto con una funcionalidad clara y relativamente sencilla como con una facilidad de combinarse con otros módulos.
La complejidad de un programa crece en forma exponencial con respecto a su tamaño. Esto significa que si se tiene un programa grande, su complejidad será mucho mayor que la suma de la complejidad de varios programas más pequeños (módulos) que cumplan el mismo objetivo.
Se espera que cada módulo pueda ser desarrollado con cierta independencia de los demás, y que incluso sea compilado en forma separada. Con esto se busca lograr una mayor productividad al desarrollar software en gran escala, en donde los módulos son asignados a las personas que componen el equipo de trabajo y los desarrollan en forma independiente y paralela. Cada módulo debe estar bien definido, y su acoplamiento con los demás debería ser el mínimo posible. Entre menos dependencias existan entre ellos, mas consistente será la aplicación final.
Las características deseables de un módulo (las que aumentan la modularidad), son las siguientes:

- **Alta cohesión**: la cohesión mide el grado de relación funcional interna de los componentes de un módulo. En la ingeniería del software, la cohesión se traduce en una medida del grado en que las líneas de código dentro del módulo colaboran para ofrecer una función concreta. Cada objeto debe tener una única funcionalidad, bien definida.
  Según Booch, existe alta cohesion funcional cuando los elementos de un componente (como una clase) "trabajan todos juntos para proporcionar algún comportamiento bien delimitado".
- **Bajo acoplamiento**: el acoplamiento mide la interconexión entre módulos. En la ingeniería del software, el acoplamiento se traduce en una medida de la relación entra las líneas de código pertenecientes a módulos diferentes. Un bajo acoplamiento es deseable, para evitar la dependencia entre módulos, que impliquen cambios en cascada.

La modularidad es una de las medidas de calidad en el desarrollo de software.
El propósito de la división en módulos es reducir el costo del software al permitir el diseño y revisión independiente de los módulos y la reutilización de los mismos. Puede llevar a que el programador tarde más en terminar un programa, sin embargo, el beneficio es que permite que el mantenimiento del programa sea mucho más simple.

4) **JERARQUIA:**

La jerarquía es una clasificación u ordenación de abstracciones (Booch, 1996).
El ser humano siempre tiende a clasificar los objetos que lo rodean, creando clasificaciones de especies, de elementos químicos, de materiales, etc. Esas clasificaciones forman jerarquías que permiten concentrarse en subconjuntos de elementos, lo que facilita el estudio de los elementos como un todo.
Los sistemas complejos presentan jerarquías diferentes:

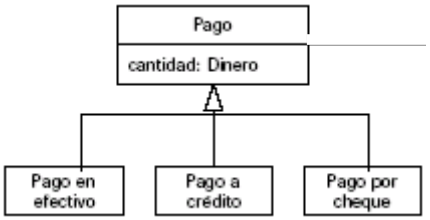
- Jerarquía estructural o <<**parte de**>> (Agregación)
- Jerarquía de tipo <<**es un**>> (Herencia)

Esta separación facilita el estudio de cada parte en forma relativamente aislada.

En el modelo de objetos el concepto mas aprovechado es el de herencia. La herencia permite a los objetos ser definidos y creados como tipos especializados de objetos preexistentes. Estos pueden compartir (y extender) su comportamiento sin tener que reimplementarlo. Por ejemplo, los elefantes, tigres, vacas, y caballos son todos mamíferos de sangre caliente. Existe una clase de animales denominados mamíferos que comparten un conjunto determinado de características. Sin embargo, hay clases de animales, como las aves, que son de sangre caliente pero no son mamíferos. O sea, la clase de animales de sangre caliente contempla tanto los mamíferos como las aves. Todos los mamíferos y todas las aves heredan las características de los animales de sangre caliente, pero pertenecen a clases diferentes de animales.

La herencia define una jerarquía de abstracciones, en donde una subclase hereda de una superclase, y además puede agregar o redefinir elementos a la estructura o al comportamiento previo. Esto hace posible definir software nuevo de la misma manera en que se introduce un concepto a una persona que lo desconoce, o sea, comparándolo con algo que ya existe y con lo que ya está familiarizado. La idea es tomar software existente, ya depurado, y construir a partir de él las nuevas soluciones, reutilizando y modificando la jerarquía de clases según se requiera.

La jerarquía de herencia se representa agrupando los objetos en *clases* y las clases en *árboles* que reflejan un comportamiento común. En la figura adjunta, los distintos tipos de pagos (efectivo, crédito, cheque), tienen la característica común de tener un monto (cantidad), y de ser un medio de pago.



La herencia es una de las propiedades más valoradas del modelo de objetos, dado que permite lograr el objetivo de construir sistemas de software a partir de partes reutilizables.

Beneficios de la programación orientados a objetos

Los principios de la definición de objetos ayudan a los programadores a hacer código más robusto, mantenible y seguro; porque se puede aislar cada uno de los objetos y tratarlo como un ente único, con su propia personalidad, sin que haya cientos de características que se deban tener presentes. Desde el punto de vista económico, esto resulta muy beneficioso, ya que los objetos bien diseñados pueden ser utilizados en muy diversas aplicaciones, con lo cual el tiempo de desarrollo total se reduce.

Existen dos beneficios fundamentales que surgen de la aplicación de la TOO en los procesos de desarrollo:

1) Mejora de la Productividad:
   - las clases de objetos son bloques básicos, modulares, que permiten la reutilización de modo similar a la construcción de cualquier objeto complejo (tal como un automóvil) que se construye ensamblando sus partes. **Esto acorta los tiempos de desarrollo, disminuyendo en consecuencia los costos**.
   - Cada objeto es una "caja negra" con respecto a los objetos externos con los que debe comunicarse. Esto significa que las estructuras de datos internos y métodos se pueden modificar sin afectar a otras partes. Por lo tanto, **el mantenimiento es más sencillo y rápido**.
   - Las tecnologías de objetos ayudan a los desarrolladores a **tratar la complejidad**., lo cual redunda en beneficio al facilitar el desarrollo
2) Mejora de la Fiabilidad:
   - También basada en la reusabilidad, dado que si es un módulo que ya se está usando, se tiene la seguridad de que ya fue probado y funciona bien.

Lenguajes de programación orientados a objetos

La programación orientada a objetos puede llevarse a cabo con lenguajes convencionales, pero esto exige al programador la construcción de los mecanismos de que disponen los lenguajes orientados a objetos. Por ello, lo más apropiado es utilizar directamente un lenguaje orientado a objetos, ya que éstos soportan los mecanismos y las características que anteriormente se han expuesto, tales como objetos, clases,

métodos, mensajes, herencia, polimorfismo, etc. La herencia constituye uno de los mecanismos más potentes de la programación orientada a objetos.

El primer lenguaje que introdujo los conceptos de orientación a objetos fue SIMULA 67 creado en Noruega, por un grupo de investigadores del Centro de Cálculo Noruego, con el fin de realizar simulaciones discretas de sistemas reales.

En estos tiempos no existían lenguajes de programación que se ajustaran a sus necesidades, así que se basaron en el lenguaje ALGOL 60 y lo extendieron con conceptos de objetos, clases, herencia y polimorfismo.

El lenguaje fue utilizado sobre todo en Europa y no tuvo mucho impacto comercial, sin embargo los conceptos que se definieron en él, se volvieron sumamente importantes para el futuro del desarrollo de software.

Alrededor de los años '70 fue desarrollado el lenguaje SMALLTALK en los laboratorios Xerox en Palo Alto, E.U.A.. Éste lenguaje adoptó los conceptos nombrados anteriormente como su fundamento. El hecho de ser creado en E.U.A., ayudo a que se introdujera a nivel mundial el término de Orientación a Objetos y que cobrara importancia entre los diseñadores de lenguajes de programación.

Los puntos importantes de este lenguaje fueron, por un lado, adoptar el concepto de objeto y clase como núcleo del lenguaje y la programación interactiva, incorporando las ideas ya conocidas de lenguajes funcionales. Es decir que se tuviese un lenguaje interpretado y no compilado.

En 1985, E. Stroustrup extendió el lenguaje de programación C a C++, es decir C con conceptos de clases y objetos. También por esta fecha se creó desde sus bases el lenguaje EIFFEL por B. Meyer. Ambos manejan conceptos de objetos y herencia de clases. Permiten herencia múltiple, que se introduce pensando en dar mayor flexibilidad a los desarrolladores. Sin embargo, actualmente la herencia múltiple se ha evitado por agregar complejidad en las estructura de clases.

Ambos lenguajes tuvieron importancia entre 1985 y hasta la primera mitad de los noventa.

En 1995 apareció JAVA, el más reciente lenguaje OO, desarrollado por SUN, que hereda conceptos de C++, pero los simplifica y evita la herencia múltiple. A cambio se introduce el término de interfaz, y la herencia múltiple de interfaces. Obtiene una rápida aceptación gracias a los applets, que son unos programas en JAVA insertado en páginas WEB dentro del código HTML.

Estos programas pueden viajar a través de la Internet y brindarle al usuario mayor interactividad con las páginas WEB. JAVA introduce también, la programación concurrente y distribuida. El lenguaje es mitad compilado y mitad interpretado dando como resultado la portabilidad a distintas plataformas. JAVA aun sigue evolucionando y se espera que en los próximos años logre la madurez adecuada para volverse un lenguaje de desarrollo de mayor importancia.

Los lenguajes orientados a objetos puros implementan todas las propiedades de POO mencionadas y utilizan todas las ventajas de esta tecnología. Son más difíciles de aprender en un principio. Sin embargo, ofrecen mayor facilidad de reutilización.

Los lenguajes híbridos tienen como base algún lenguaje tradicional, al que agregan los conceptos de objetos, mensajes y clases.

Más popular de los lenguajes puros es Smalltalk y entre los híbridos sobresale Java.