1. INTRODUCCIÓN

El desarrollo de software es un proceso muy complejo requiriendo de una metodología eficiente y sistemática. El ciclo de vida de un sistema de computación comienza con la formulación de un problema, seguido de análisis, diseño, implementación, verificación y validación del software. A continuación prosigue una fase operacional durante la cual se mantiene y extiende el sistema. El modelo orientado a objetos presenta un enfoque evolucionado para la ingeniería de software.

Existen diversas metodologías para el desarrollo de sistemas orientados a objetos, todos sustentados en principios similares aunque no idénticos. En este libro nos apoyaremos en el modelo y metodología OMT (Object Modeling Technique) [Rumbaugh et al. 1991], uno de los más importantes en el área.

1.1 Programación Orientada a Objetos

Una diferencia importante entre la programación tradicional y la programación orientada a objetos es que la programación tradicional separa los datos de las funciones, mientras que la programación orientada a objetos define un conjunto de objetos donde se combina de forma modular los datos con las funciones.

Los aspectos principales de la programación orientada a objetos son:
1) Objetos
2) Clasificación
3) Instanciación
4) Generalización
5) Abstracción
6) Encapsulación
7) Modularidad
8) Extensibilidad
9) Polimorfismo
10) Reuso de Código

1.1.1 Objetos

- El *objeto* es la entidad básica del modelo orientado a objetos,
- El objeto integra una estructura de datos (*atributos*) y un comportamiento (*operaciones*)
- Los objetos se distinguen entre sí por medio de su propia identidad, aunque internamente los valores de sus atributos sean iguales.

1.1.2 Clasificación

- Las *clases* describen posibles objetos, comunes en su estructura y comportamiento.
- Los objetos que contienen los mismos atributos y operaciones pertenecen a la misma clase.
- La estructura de clases integra las operaciones con los atributos a los cuales se aplican.

1.1.3 Instanciación

- El proceso de crear objetos pertenecientes a cierta clase se llama *instanciación*.
- El objeto es la instancia de una clase.
- Pueden ser instanciados un número indefinido de objetos de cierta clase.

1.1.4 Generalización

- En una jerarquía de clases, se comparten atributos y operaciones entre clases basados en la *generalización* de clases.
- El mecanismo para describir la jerarquía de generalización de clases es la *herencia*.
- Las clases más generales se conocen como *superclases*.
- Las clases más especializadas se conocen como *subclases*.
- La herencia puede ser *sencilla* o *múltiple*.

1.1.5 Abstracción

- Se concentra en lo primordial de una entidad y no en sus propiedades secundarias.
- Se concentra en lo que el objeto hace y no en cómo lo hace.
- Se da énfasis a cuales son los objetos y no cómo son usados, logrando el desarrollo de sistemas más estables.

1.1.6 Encapsulación

- *Encapsulación* o *encapsulamiento* es la separación de las propiedades externas de un objeto de los detalles de implementación internos del objeto.
- Se separa la interfase del objeto de sus implementación, limitando la complejidad al mostrarse sólo la informacion relevante.
- Se limita el impacto a cambios en la implementación, ya que los cambios a las propiedades internas del objeto no afectan su interacción externa.
- Se reduce el esfuerzo en migrar el sistema a diferentes plataformas.
- Se protege al objeto contra posibles errores, y se permite hacer extensiones futuras en su implementación.

1.1.7 Modularidad

- El encapsulamiento de los objetos da lugar a gran *modularidad*.
- Cada módulo se concentra en una sola clase de objetos.
- Los módulos tienden a ser pequeños y concisos.
- La modularidad facilita encontrar y corregir problemas.
- La complejidad del sistema se reduce facilitando su mantenimiento.

1.1.8 Extensibilidad

- La *extensibilidad* permite hacer cambios en el sistema sin afectar lo que ya existe.
- Nuevas clases pueden ser definidas sin tener que cambiar la interfase del resto del sistema.
- La definición de los objetos existentes puede ser extendida sin necesidad de cambios más allá del propio objeto.

1.1.9 Polimorfismo

- *Polimorfismo* permite definir las mismas operaciones con diferente comportamiento en diferentes clases.
- Se permite llamar una operación sin preocuparse de cuál implementación es requerida en que clase, siendo la responsabilidad de la jerarquía de clases y no del usuario

1.1.10 Reuso de Código

- La orientación a objetos apoya el *reuso de código* en el sistema.
- Los componentes orientados a objetos se pueden utilizar para estructurar librerías reusables.
- El reuso reduce el tamaño del sistema durante la creación y ejecución.
- Al corresponder varios objetos a una misma clase, se guardan los atributos y operaciones una sola vez por clase, y no por cada objeto.
- Por medio de la generalización y especialización, las subclases heredan atributos y operaciones comunes, definidos en las superclases.
- La herencia es uno de los factores más importantes contribuyendo al incremento en el reuso de código dentro de un proyecto o entre múltiples proyectos.
- Nuevas subclases de clases previamente definidas se pueden crear en el reuso de una clase, pudiéndose crear nuevas operaciones, o modificar las ya existentes.

1.2 Lenguajes de Programación Orientados a Objetos

Los aspectos fundamentales de la programación orientada a objetos se originó de dos motivaciones:
1) La dificultad en el diseño y mantenimiento de sistemas grandes de software, lo cual introdujo la noción de programación estructurada, representada por Algol (Algol 60 [Naur 1963], Algol 68 [Algol 1975], o Pascal [Wirth 1971]).
2) El modelo del razonamiento humano en inteligencia artificial, ha mostrado la necesidad de estructurar cantidades grandes de datos encapsulando la información y las propiedades asociadas, con un concepto particular en una sola entidad [Minsky 1975].

Simula I [Dahl 1966] fue originalmente diseñada para problemas de simulación y fue el primer lenguaje en el cual los datos y procedimientos estaban unificados como una sola entidad. Simula [Birtwistle et al. 1973], su sucesor, derivó definiciones formales a los conceptos de objeto y clase, donde un programa es una colección de objetos activos y autónomos guardando sus propios procedimientos y variables locales. Cada objeto tiene un comportamiento particular

permitiendo simular procesos concurrentes. La clase es utilizada para definir comportamiento común de un conjunto de objetos, llamados instancias de la clase, y es incorporado en una jerarquía de herencia.

Otros lenguajes, como Ada [Ada 1983] (*packages*) y CLU [Liskov 1986] (*clusters*), se derivan de conceptos similares, con la excepción de la incorporación de una jerarquía de herencia.

Simula sirvió como modelo para una generación completa de lenguajes orientados a objetos, caracterizados por su naturaleza de tipo estáticos, en particular C++ [Stroustrup 1986; 1991], Eiffel [Meyer 1986], y Beta [Kristensen et al. 1987].

Smalltalk, el descendiente directo de Simula, generaliza el concepto de un objeto como única entidad manipulada en los programas. Tres versiones sucesivas de este lenguaje han sido desarrolladas. Smalltalk-72 [Kay and Goldberg 1976], introdujo el *paso de mensajes* para permitir la comunicación entre objetos. Herencia apareció en Smalltalk-76 [Ingalls 1978], donde cada clase es definida con respecto a otra, cuyas propiedades hereda, y el conjunto de clases es organizada en una estructura jerárquica con una sola raíz. Smalltalk-80 [Goldberg and Robson 1983] se estableció como un uso estándar y sistemático de metaclases, una idea ya presente de forma rudimentaria en Smalltalk-76. Smalltalk-80 es a la vez un lenguaje y un ambiente de programación muy sofisticado, cuya interactividad es inspirada por Lisp. Es el fruto del trabajo original sobre la máquina FLEX [Kay 1968; 1969], el cual fué el primer prototipo de una computadora personal corriendo un lenguaje de alto nivel. La máquina fué, en particular, provista con el primer sistema de ventanas, el ancestro de la estación de trabajo Xerox Star [Smith et al. 1983].

Lisp fué otro idioma que contribuyó de forma importante a la evolución de la programación orientada a objetos. Flavors [Moon 1986], incluyó herencia múltiple apoyada con facilidades para la combinación de métodos heredados. Finalmente CLOS [Keene 1989], es el estándar del sistema de objetos para Common Lisp.

Otros lenguajes también han sido extendidos con objetos, como es el caso de Objective-C [Cox 1983], integrado con C.

1.3 OMT - Object Modeling Technique

OMT (Object Modeling Technique) es una metodología de ingeniería de software para sistemas orientados a objetos. OMT apoya un modelo y una notación particular de orientación a objetos.

1.3.1 Modelo

Las características principales de los modelo de computación son las siguientes:

- ofrecen una forma de pensar más que una forma de programar,
- reducen la complejidad en el diseño de software,
- sirven para la especificación, análisis, documentación, y programación de un sistema,

- permiten atacar los errores durante el diseño en lugar de durante la implementación, donde el costo de reparación es bastante más costoso.

El modelo OMT está compuesto por tres modelos ortogonales, los cuales sirven para describir de forma completa un sistema. En orden de importancia y desarrollo, los tres modelos son:
1) el modelo de objetos,
2) el modelo dinámico,
3) el modelo funcional.

1.3.1.1 Modelo de Objetos

En el *modelo de objetos* se describe el "que" del sistema:
- Se describen las estructuras estáticas: principalmente *clases* y *objetos*.
- Se describe la relación entre las estructuras estáticas.
- Se representa el modelo de objetos por medio de *diagramas de clases*, donde los nodos en los diagramas corresponden a las *clases* y los arcos corresponden a relaciones entre las *clases*.
- Los diagramas de clases están acompañados de *diagramas de objetos*, donde los nodos en los diagramas corresponden a los *objetos* y los arcos corresponden a las relaciones entre los *objetos*.

1.3.1.2 Modelo Dinámico

En el *modelo dinámico* se describe el "cuando" del sistema:
- Se describen los aspectos de control: los *eventos* externos del sistema.
- Se describen los aspectos que varían con el tiempo: las secuencias de *operaciones*.
- Se describen las interacciones entre los *objetos*.
- Se describen los estados de los *objetos*.
- Se representa el modelo dinámico por medio de *diagramas de estado*, donde los nodos corresponden a los estados de los *objetos* y los arcos corresponden a las transiciones entre estados de los *objetos*.

1.3.1.3 Modelo Funcional

En el *modelo funcional* se describe el "como" del sistema:
- Se describen las dependencias de datos en el sistema.
- Se describen las transformaciones de datos en el sistema.
- Se representa el modelo funcional por medio de *diagramas de flujo de datos*, donde los nodos corresponden a los procesos de los *objetos*, y los arcos corresponden a los flujos de datos entre *objetos*.

1.3.2 Metodología

La metodología OMT para el desarrollo de software se basa en los tres modelos anteriormente descritos, y consiste en tres etapas de desarrollo:
1) Análisis,

2) Diseño,
3) Implementación.

Estas tres etapas se siguen de forma iterativa agregando cada vez más detalles. Aunque presentaremos las diferentes etapas de forma independiente, en general, los desarrolladores experimentados pueden combinar las diferentes etapas.

1.3.2.1 Análisis

El proceso de análisis se puede resumir según el diagrama de la Figura 1.1.

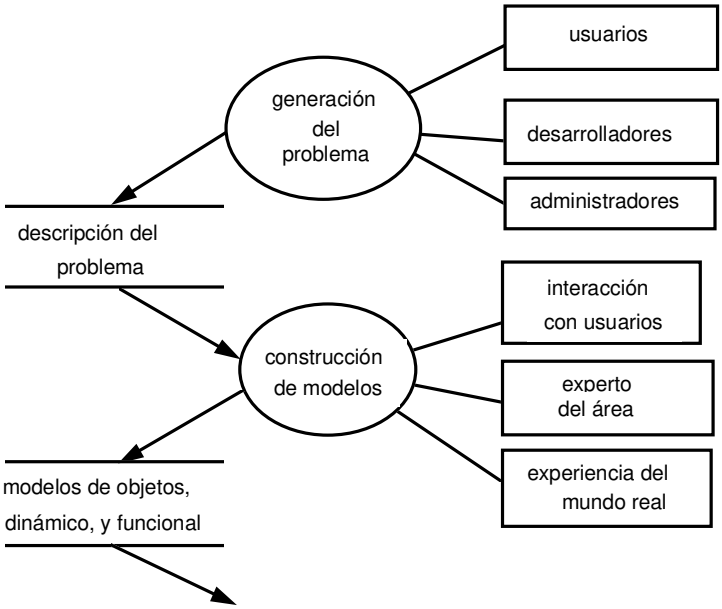

Figura 1.1. Diagrama de Análisis.

El primer paso del análisis es generar la *descripción del problema* por medio de interacciones con los usuarios, desarrolladores, administradores, y expertos en el área del problema. El análisis sobre la descripción del problema da lugar a la construcción de los tres modelos anteriormente definidos en la metodología OMT:

Durante el *análisis* :

- Se muestra los aspectos más importantes del sistema sin importar su implementación final.
- El análisis sirve para comunicarse con los clientes y los expertos en el área, sin necesidad de tener un conocimiento de la computación.
- El análisis es una representación concisa de lo "que" el sistema va a hacer pero no "como".
- El analista trabaja con el cliente para clarificar la descripción del problema, la cual originalmente puede ser incompleta, e incluso tener inconsistencias.

1.3.2.2 Diseño

La etapa de diseño consiste de dos fases:
1) Diseño de sistemas,
2) Diseño de objetos.

1.1.2.2.1 Diseño de Sistemas

Durante el *diseño de sistemas* :

- Se toman decisiones de alto nivel sobre la arquitectura del sistema a desarrollarse.
- Se diseña el ambiente de implementación, incluyendo decisiones sobre software y hardware.
- El sistema de divide en subsistemas basados en los modelos de análisis y la arquitectura propuesta.
- Se identifica la concurrencia en el sistema.
- Los subsistemas se asignan a los procesadores y tareas según la arquitectura propuesta.
- Se escoge el manejo de almacenamientos de datos.
- Se escogen los mecanismos para coordinar el acceso a recursos globales
- Se escoge la implementación del control del software.
- Se escoge el enfoque para el manejo de condiciones de borde, como errores.
- Se deciden las prioridades durante el diseño, incluyendo
    - Rendimiento
    - Memoria
    - Protocolos de comunicación
    - Flexibilidad
    - Extensibilidad

1.1.2.2.2 Diseño de Objetos

Durante el *diseño de objetos* :

- Se añaden detalles para la implementación final, basados en el análisis, tales como:
    - Extensión y refinamiento de las estructura de datos
    - Descripción final de algoritmos
- Se combinan los tres modelos para determinar con más detalle las operaciones en las clases.
- Se escoge o diseña los algoritmos para implementar los métodos.
- Se escogen las estructuras de datos.
- Se añaden clases, operaciones, y objetos internos del sistema.
- Se hacen optimizaciones al sistema.
- Se ajusta la herencia creando superclases abstractas para incrementar el reuso de codigo.
- Se determina la representacion final de los objetos.
- Se empacan las clases y operaciones en módulos.

1.3.2.3 Implementación

Durante la *implementación*:
- Se traduce el diseño a un lenguaje de programación particular, bases de datos o hardware.
- La implementacion debe ser relativamente sencilla y directa, ya que todas las decisiones han sido hechas en las etapas previas.

1.1.2.3.1 Implementación con Lenguajes de Programación

- El lenguaje de programación no tiene que ser necesariamente orientado a objetos.
- El uso de un lenguaje de programación orientado a objetos hace más fácil la implementacion de un diseño orientado a objetos
- La elección del lenguaje influye en el diseño, pero el diseño no debe depender de los detalles del lenguaje.
- Si se cambia de lenguaje de programación no debe requerirse el re-diseño del sistema.
- Los lenguajes de programación y sistemas operativos difieren mucho en su organizacion, aunque la mayoría de los lenguajes tienen la habilidad de expresar los tres aspectos de la especificación de software expresados en los tres modelos OMT
    - Estructura de datos estáticas: Modelo de Objetos
    - Flujo dinámico de control: Modelo Dinámico
        - procedural (ciclos, condiciones) o declarativo (reglas, tablas)
        - secuencial o concurrente
    - Transformaciones funcionales: Modelo Funcional

1.1.2.3.2 Implementación con Base de Datos

- Se pueden utilizar bases de datos orientadas a objetos u otros tipos de bases de datos.
- Si los aspectos dinámicos y funcionales del sistema son pocos, comparados con las estructuras del sistema, una base de datos relacional puede que sea suficiente.
- Una base de datos relacional se puede utilizar para implementar las estructuras del sistema.
- Una base de datos *persistente* se puede utilizar para guardar las estructuras fuera del programa.

1.4 Comparación de Metodologías

En esta sección se hará una comparación entre OMT y:
1) Otras metodologías orientadas a objetos.
2) Las metodologías estructurales (SA/SD: Structural Analisis/Structural Design).

1.4.1 Comparación con otras Metodologías Orientadas a Objetos

Otras de las más importantes metodologías orientadas a objetos:
- OOSE/Objectory (Object Oriented Software Engineering) [Jacobson 92]
- OOA&D/ROSE (Object-Oriented Analysis & Design) [Booch 94]
- FUSION (Object-Oriented Development) [Coleman et al. 94]

- OOK/MOSES (Object-Oriented Knowledge) [Henderson-Sellers et al. 92]
- OBA (Object Behavior Analysis) [Rubin y Goldberg 92]
- OOA&D (Object-Oriented Analysis and Design) [Coad y Yourdon 91]
- RDD (Responsibility-Driven Design) [Wirfs-Brock 90]
- OOSA (Object-Orientd System Analysis) [Schlaer y Mellor 92]
- OOSD (Object-Oriented System Development) [de Champeaux 93]
- OOA&D (Object-Oriented Analysis and Design) [Martin y Odell 92]
- OOSA (Object-Oriented Systems Analysis) [Embley et al. 92]
- OORA (Object-Oriented Requirements Analysis) [Firesmith 93]
- Synthesis [Page-Jones y Weiss 93]

1.4.2 Comparación con el Análisis y Diseño Estructurado

Las metodologías de ingeniería de software más utilizadas en la actualidad están basadas en los modelos estructurados de Yourdon, Constantine, DeMarco, Page-Jones, Martin, Jackson.

- El análisis y diseño estructurado (SA/SD) se concentra principalmente en especificar y descomponer la funcionalidad del sistema total, donde en general, los cambios a los requisitos cambiarían totalmente la estructura del sistema.
- En ambas metodologías, SA/SD y OMT, podemos encontrar componentes similares, los cuales apoyan las tres aspectos generales del sistema: modelo de objetos, dinámico y funcional.
- La diferencia principal entre las dos metodologías es de estilo y en el énfasis que se le da a cada modelo. OMT está dominado por el modelo de objetos, que da el contexto para los modelos dinámico y funcional, mientras que en SA/SD el modelo funcional domina, el modelo dinámico es el siguiente en importancia, y el modelo de objeto es el menos importante.
- SA/SD organiza el sistema alrededor de procedimientos. En contraste, los sistemas orientados a objetos se organizan alrededor del concepto de objetos.
- Si la mayoría de las modificaciones en los requisitos son cambios en la funcionalidad más que en los objetos, entonces los cambios pueden ser críticos para sistemas basados en funciones, mientras que las modificaciones en la funcionalidad son rapidamente acomodados en los sistemas orientados a objetos. SA/SD es útil para problemas donde las funciones son más importantes y complejas que los datos.
- Un diseño orientado a objetos es generalmente más resistente a cambios y más extensible.
- En SA/SD la descomposición de un proceso en subprocesos es bastante arbitraria. Diferentes personas producirían diferente descomposiciones. En cambio, en el diseño orientado a objetos, la descomposición está basada en los objetos del dominio del problema. Por lo tanto los desarrolladores de diferentes programas en el mismo dominio tienden a descubrir objetos similares, lo cual incrementa el reuso de componentes de un proyecto a otro.
- El enfoque orientado a objetos integra de mejor forma las bases de datos con código de programación. En cambio, un enfoque de diseño procedural tiene grandes problemas al tratar con las bases de datos, ya que es difícil integrar código de programación organizado alrededor de funciones con base de datos organizadas alrededor de datos.

1.5 CASE - Computer Aided Software Engeneering

CASE (Computer Aided Software Engeneering) son herramientas para asistir al ingeniero de software en las diferentes fases del ciclo de vida del desarrollo del software. Existen muchos productos comerciales de CASE, con distintas funciones.

Las principales etapas en el desarrollo del software son las siguientes:

- Planeación
- Análisis
- Diseño
- Implementación
- Verificación
- Validación
- Mantenimiento
- Administración

Las productos CASE se pueden clasificar de la siguiente forma:
1) *Herramientas* específicas apoyando actividades particulares en el desarrollo de software.
2) *Bancos de trabajo* ("Workbenches") apoyando una o dos actividades en el desarrollo de software.
3) *Ambientes* apoyando una gran parte del proceso de desarrollo del software.

1.5.1 Herramientas

Las *herramientas* más comunes en el desarrollo del software son las siguientes:

- Editores: textuales o gráficos.
- Herramientas para la programación: codificadores, depuradores ("debuggers"), compiladores, ensambladores.
- Herramientas para la verificación y validación: analizadores estáticos y dinámicos, diagramas de flujos.
- Herramientas para la medición: monitores.
- Herramientas para la administración de la configuración: versiones, librerías.
- Herramientas para la administración del proyecto: estimación, planeación, costo.

1.5.2 Bancos de Trabajo

Los *bancos de trabajo* más comunes en el desarrollo del software son las siguientes:

- Planeación y modelo.
- Análisis y diseño.
- Interfases de usuario.
- Programación.
- Verificación y validación.

- Mantenimiento e ingeniería en reversa.
- Administración de la configuración.
- Administración del proyecto.

1.5.3 Ambientes

Los *ambientes* más comunes en el desarrollo del software son los siguientes:

- Juego de herramientas ("toolkits").
- Ambientes centrados en lenguajes.
- Ambientes integrados con lenguajes.
- Lenguajes de cuarta generación.
- Ambientes centrados en procesos.