CAPÍTULO III -

UML Y LOS PROCESOS DE DESARROLLO

DE SOFTWARE

3.1 Paradigma orientado a objetos

A medida que pasa el tiempo los sistemas de software se vuelven cada vez más complejos. Para ayudarnos a lidiar con la complejidad, se comenzó a utilizar la programación estructurada, donde la programación se basaba en una secuencia esperada de instrucciones de ejecución. El esfuerzo por diseñar y depurar programas, pensando en el orden que la computadora sigue para hacer las cosas desembocó en un software que nadie entendía del todo [Martin, 94].

El mundo de las técnicas OO (orientadas a objetos) con herramientas CASE es muy diferente, pues el diseñador piensa en términos de objetos y su comportamiento. Las técnicas OO nos brindan un conjunto de clases reutilizables, donde la mayor parte del proceso de construcción de software consiste en el ensamblaje de clases ya existentes y probadas. Otra aportación importante es que el paradigma OO cambia nuestra forma de pensar sobre los sistemas:

> "Para la mayoría de las personas, la forma de pensar OO es más natural que las técnicas del análisis y diseño estructurado, después de todo, el mundo está formado por objetos. Desde una etapa muy temprana categorizamos los objetos y descubrimos su comportamiento" [Martin, 94].

3.1.1 Ventajas del paradigma orientado a objetos

Las técnicas orientadas a objetos fueron creadas para mejorar la capacidad del

profesional de la computación en diversos modos. Es recomendable comprender todos los beneficios potenciales de las técnicas OO e intentar conseguir todos, en vez de sólo un subconjunto de ellos. A continuación se enlistan los beneficios del análisis y diseño orientado a objetos [Martin, 94]:

- *Reutilización*: Las clases pueden ser diseñadas para que se reutilicen en muchos sistemas.
- *Estabilidad*: Las clases diseñadas para una reutilización repetida se vuelven estables.
- *El diseñador piensa en términos del comportamiento de objetos y no en detalles de bajo nivel*: El encapsulado oculta los detalles y hace que las clases complejas sean fáciles de utilizar (las clases pueden verse como cajas negras).
- *Se construyen clases cada vez más complejas*: Se construyen clases a partir de clases ya existentes y probadas. Esto permite construir componentes complejos de software, que a su vez se convierten en bloques de construcción de software más complejo.
- *Confiabilidad*: Es probable que el software construido a partir de clases estables ya probadas tenga menos fallas que el software elaborado a partir de cero.
- *Diseño de mayor calida:* Los diseños suelen tener mayor calidad, puesto que se integran a partir de componentes probados, que han sido verificados y pulidos varias veces.
- *Integridad*: Las estructuras de datos sólo pueden utilizar métodos específicos. Esto tiene particular importancia en los sistemas cliente-servidor y los sistemas distribuidos, en los que usuarios desconocidos podrían intentar el acceso al sistema.
- *Programación más sencilla:* Los programas se elaboran a partir de piezas pequeñas, cada una de las cuales, en general, se puede crear fácilmente. El programador crea un

método para una clase a la vez. El método cambia el estado de los objetos en formas que suelen ser sencillas cuando se les considera en sí mismas.

- *Mantenimiento más sencillo*: El programador encargado del mantenimiento del sistema cambia un método de cada clase a la vez. Cada clase efectúa sus funciones independientemente de las demás.
- *Modelado más realista*: El análisis OO modela la empresa o área de aplicación de manera que sea lo más cercana posible a la realidad de lo que se logra con el análisis convencional.
- *Mejor comunicación entre los profesionales de los sistemas de información y los empresarios*: Los empresarios comprenden más fácilmente el paradigma OO. Piensan en términos de eventos, objetos y políticas empresariales que describen el comportamiento de los objetos.
- *Independencia del diseño:* Las clases están diseñadas para ser independientes del ambiente de plataformas, hardware y software.

3.2 El lenguaje unificado de construcción de modelos, UML

El UML (*Unified Modeling Language*) se define como un "lenguaje que permite especificar, visualizar y construir los artefactos de los sistemas de software" [Booch et al, 97]. Es un lenguaje notacional (que, entre otras cosas, incluye el significado de sus notaciones) destinado a los sistemas de modelado que utilizan conceptos orientados a objetos.

El UML es un estándar para construir modelos orientados a objetos. Nació en 1994 por iniciativa de Grady Booch y Jim Rumbaugh para combinar sus dos famosos métodos: el de Booch y el OMT (*Object Modeling Technique*, Técnica de Modelado de Objetos). Más tarde se les unió Ivar Jacobson, creador del método OOSE (*Object-Oriented Software Engineering*, Ingeniería de Software Orientada a Objetos). En respuesta a una petición del OMG (*Object Management Group*, Grupo de Administración de Objetos) para definir un lenguaje y una notación estándar del lenguaje de construcción de modelos, en 1997 propusieron el UML como candidato [Larman, 99]. Ver Fig. 3.1.

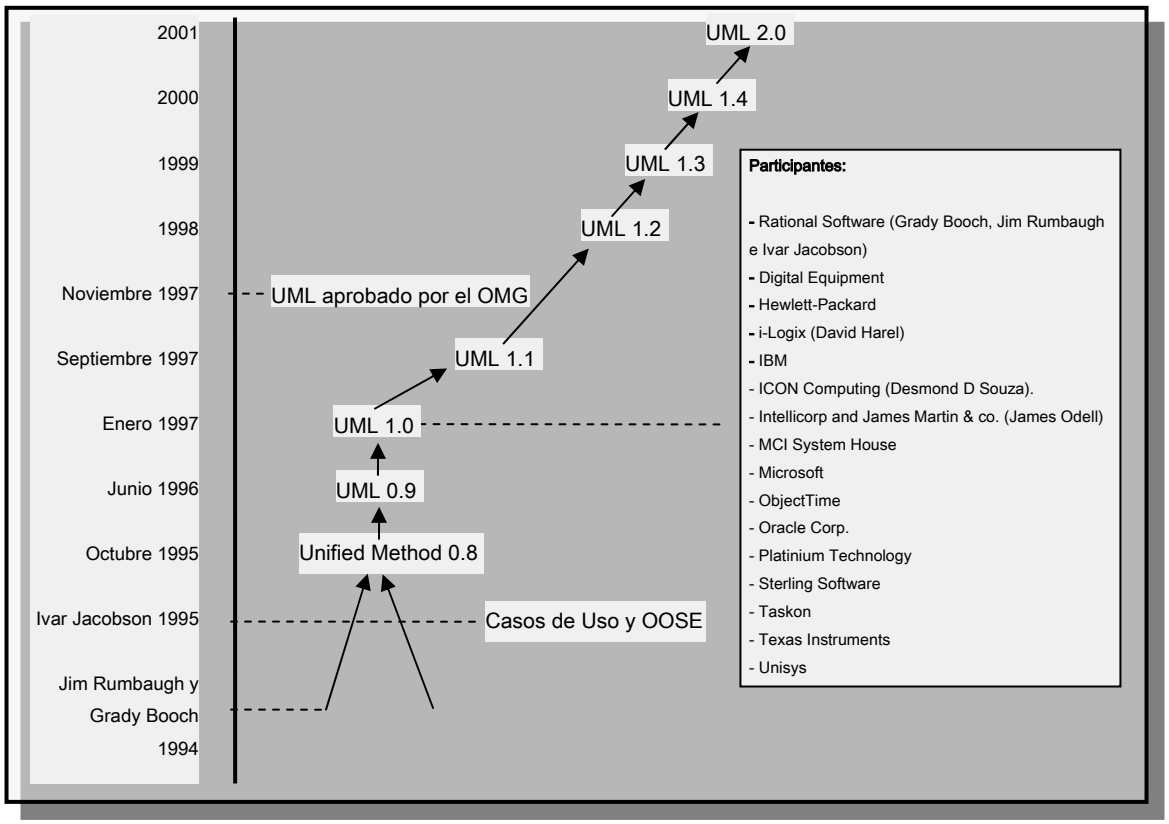

Fig. 3.1 Historia de UML [Letelier, 02]

3.2.1 Vistas del UML

En la construcción de software usando UML, existen cinco vistas para visualizar, especificar, construir y documentar la arquitectura del software. UML permite representar cada vista mediante un conjunto de diagramas, las vistas son las siguientes [Booch et al, 97]:

- *Vista de casos de uso:* Muestra la funcionalidad del sistema desde el punto de vista de un actor externo que interactúa con él. Esta vista es útil a clientes, diseñadores y desarrolladores.
- *Vista de diseño:* Muestra la funcionalidad del diseño dentro del sistema en términos de la estructura estática y comportamiento dinámico del sistema. Esta vista es útil a diseñadores y desarrolladores. Se definen propiedades tales como: persistencia, concurrencia, interfaces y estructuras internas a las clases.
- *Vista de procesos:* Muestra la concurrencia del sistema, comunicación y sincronización. Útil a desarrolladores e integradores.
- *Vista de implementación:* Muestra la organización de los componentes de código. Útil a desarrolladores.
- *Vista de implantación (también conocida como vista de despliegue):* Muestra la implantación del sistema en la arquitectura física. Útil a desarrolladores, integradores y verificadores.

3.2.2 Definición de modelo

Un sistema (tanto en el mundo real como en el mundo del software) suele ser extremadamente intrincado, por ello es necesario dividir el sistema en partes o fragmentos si queremos entender y administrar su complejidad. Estas partes podemos representarlas como modelos que describan y abstraigan sus aspectos esenciales [Rumbaugh, 97].

Un modelo captura una vista de un sistema del mundo real. Es una abstracción de dicho sistema considerando un cierto propósito. Así, el modelo describe completamente aquellos aspectos del sistema que son relevantes al propósito del modelo y a un apropiado nivel de detalle.

Los modelos se componen de otros modelos, de diagramas y documentos que describen detalles del sistema. El UML especifica varios diagramas. Si queremos caracterizar los modelos, podemos poner de manifiesto la información *estática* o *dinámica* de un sistema. Un *modelo estático* describe las propiedades estructurales del sistema; en cambio, un *modelo dinámico* describe las propiedades de comportamiento de un sistema. Es importante mencionar que el UML es un lenguaje para construir modelos; no guía al desarrollador en la forma de realizar el análisis y diseño orientado a objetos ni indica cuál proceso de desarrollo adoptar [Larman, 99].

Para modelar un sistema es suficiente utilizar una parte de UML, "el 80 por ciento de la mayoría de los problemas pueden modelarse usando alrededor del 20 por ciento de UML" [Grady Booch].

3.2.3 Diagramas UML

UML es un lenguaje notacional. Parte importante de esta notación son los diagramas que nos permiten modelar un sistema. Un diagrama es una representación gráfica de una colección de elementos de modelado, la mayoría de las veces mostrados como grafo conexo de vértices (cosas) y arcos (relaciones). Los buenos diagramas hacen el sistema que se está desarrollando más comprensible y cercano a los objetivos. En UML se definen nueve diagramas, los cuales se pueden mezclar en cada vista (ver figura 3.2 y 3.3) [Booch et al, 97].

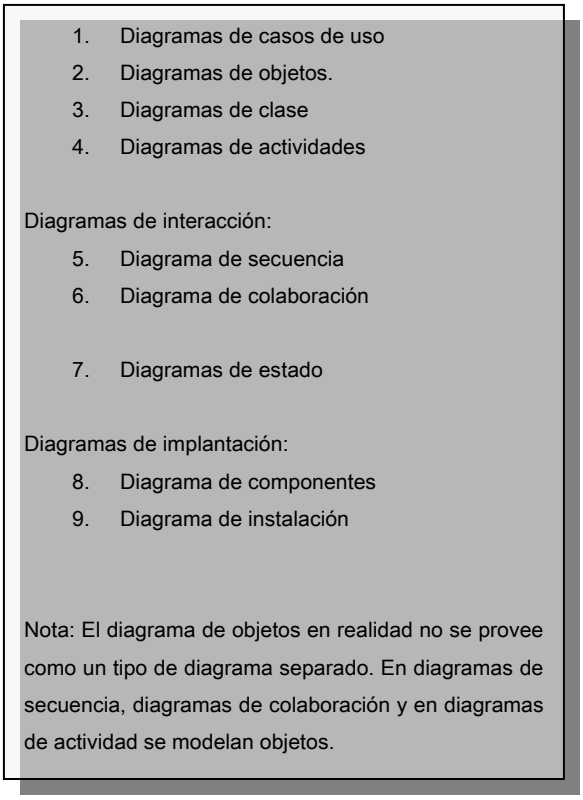

Fig. 3.2 Diagramas empleados por UML [Booch et al, 97]

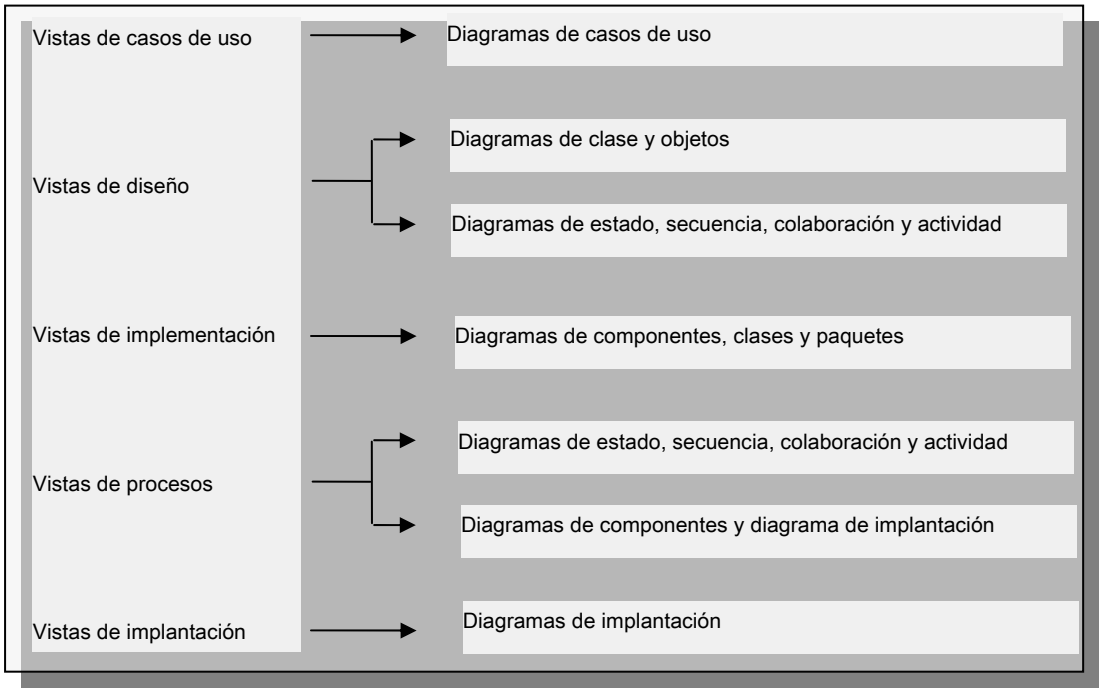

Fig. 3.3 Vistas de un software y sus respectivos diagramas UML [Booch et al, 97]

3.2.3.1 Diagramas estructurales

Los cuatro diagramas estructurales de UML existen para visualizar, especificar, construir y documentar los aspectos estáticos del sistema. Están organizados sobre grupos de objetos que se encontrarán cuando se esté modelando un sistema [Booch et al, 97].

1. *Diagrama de clases:* Un diagrama de este tipo muestra un conjunto de clases, interfaces, y sus relaciones.
2. *Diagrama de objetos:* Muestra un conjunto de objetos y sus relaciones. A diferencia de los diagramas anteriores, estos diagramas se enfocan en la perspectiva de casos de uso, y prototipos.
3. *Diagrama de componentes:* Muestra el conjunto de componentes y sus relaciones y se

utilizan para ilustrar la vista de la implementación estática de un sistema.

4. *Diagrama de implantación:* Muestra un conjunto de nodos y sus relaciones, se usan para ilustrar la vista de implantación estática de un sistema.

3.2.3.2 Diagramas de comportamiento

Los cinco diagramas de comportamiento de UML son usados para visualizar, especificar, construir y documentar los aspectos dinámicos de un sistema. Se puede pensar en los aspectos dinámicos como las representaciones de las partes cambiantes del sistema [Booch et al, 97].

1. *Diagrama de casos de uso:* Muestra el conjunto de casos de uso y actores (incluyendo sus relaciones). Estos diagramas se utilizan para ilustrar la vista del caso de uso del sistema.
2. *Diagrama de secuencia:* Es un diagrama de interacción que enfatiza el orden en tiempo de los mensajes.
3. *Diagrama de colaboración:* Es un diagrama de interacción que enfatiza la organización estructural de los objetos que envían y reciben mensajes. El diagrama de colaboración muestra un conjunto de objetos, las ligas entre ellos y los mensajes enviados y recibidos por dichos objetos.

   Nota: Los diagramas de secuencia y de colaboración son isomórficos, i.e. se puede hacer la conversión de uno a otro sin perder información.
4. *Diagrama de estado:* Muestra una máquina de estado, consistente en estados, transiciones, eventos y actividades. Estos diagramas enfatizan el comportamiento

ordenado por eventos de un objeto.

5. *Diagrama de actividad:* Muestra el flujo de una actividad a otra dentro del sistema. Ha sido diseñado para mostrar una visión simplificada de lo que ocurre durante una operación o suceso.

3.3 UML y su relación con los procesos de desarrollo de software

Un proceso de desarrollo de software es un método de organizar las actividades relacionadas con la creación, presentación y mantenimiento de los sistemas de software. El lenguaje UML no define un proceso oficial de desarrollo, en realidad UML se combina con un proceso de desarrollo para obtener un producto final (ver figura 3.4). Craig Larman [Larman, 99] da dos razones importantes que explican esto:

1. Aumentar la posibilidad de una aceptación generalizada de la notación estándar del modelado, sin la obligación de adoptar un proceso oficial.
2. La esencia de un proceso apropiado admite mucha variación y depende de las habilidades del personal, de la razón investigación-desarrollo, de la naturaleza del problema y de las herramientas.

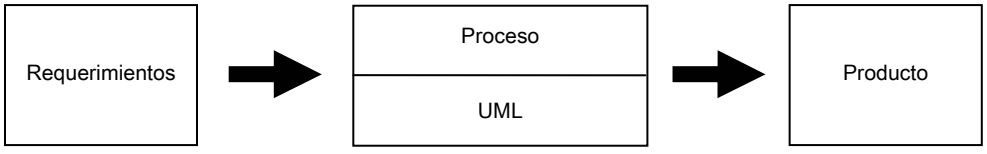

Fig. 3.4 UML y el proceso de desarrollo [Larman, 99]

3.3.1 Proceso unificado de desarrollo de software

Uno de los procesos más ocupados y recomendados para trabajar con UML es el proceso unificado de desarrollo de software (*The Unified Software Development Process)*. Este proceso fue elaborado por los creadores del UML (Jacobson, Booch y Rumbaugh). Sus características principales son [Booch et al, 99]:

- Es dirigido por los *casos de uso;* acciones realizadas (interacción) entre los usuarios y el sistema.
- Se centra en el diseño de una *arquitectura central*, la cual guía el proceso de construcción de software.
- Es un proceso que utiliza un desarrollo *iterativo e incremental*:
  - Las *iteraciones* son controladas sobre los diferentes pasos del proceso.
  - Es *incremental* porque en cada iteración el software se va ampliando y mejorando.

3.3.2 Desarrollo iterativo e incremental

Un ciclo de vida iterativo se basa en el agrandamiento y perfeccionamiento secuencial de un sistema a través de múltiples ciclos de desarrollo, análisis, diseño, implementación y pruebas. El sistema crece al incorporar nuevas funciones en cada ciclo de desarrollo. "En cada ciclo se aborda un conjunto relativamente pequeño de requerimientos, pasando por el análisis, el diseño, la construcción y las pruebas. El sistema va creciendo con cada ciclo que concluye " [Larman, 99]. Ver Figura 3.5.

Ventajas del desarrollo iterativo e incremental:

- Reducción de los riesgos basándose en una retroalimentación temprana.
- Mayor flexibilidad para manejar cambios nuevos o modificaciones a los mismos.
- La complejidad nunca resulta abrumadora.
- Se produce retroalimentación en una etapa temprana, porque la implementación se efectúa rápidamente con una parte pequeña del sistema.

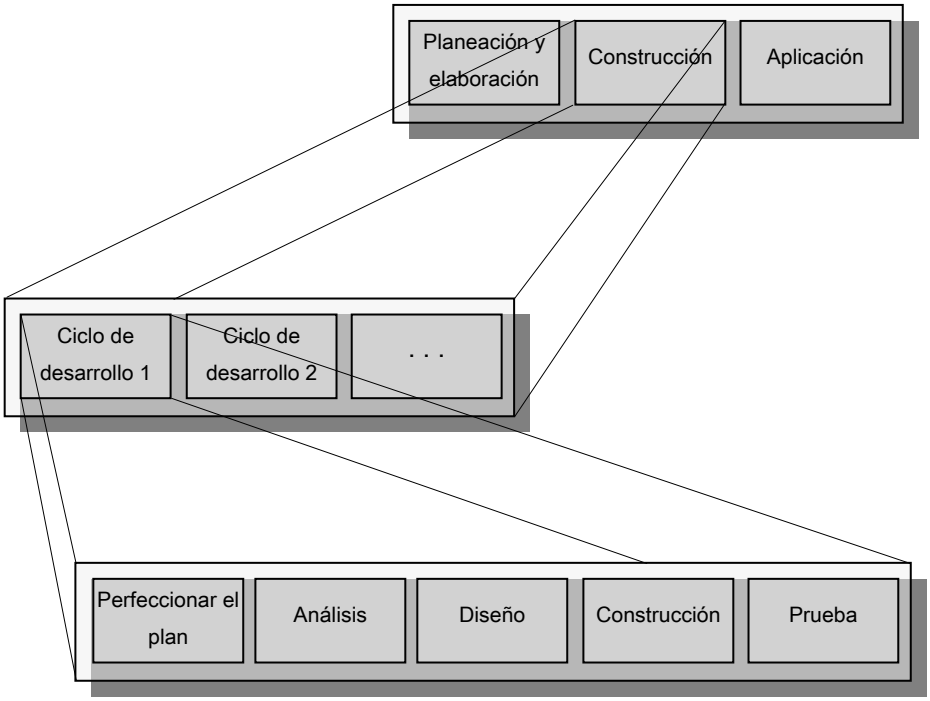

Fig. 3.5 Ciclos iterativos de desarrollo [Larman, 99]

Cada iteración es un ciclo de desarrollo que termina en la liberación de una versión parcial del producto final y cada iteración pasa a través de todos los aspectos del desarrollo de software (Ver Figura 3.6):

– Análisis de requerimientos.

– Diseño.

– Implementación.

– Pruebas.

– Documentación.

– Evaluación.

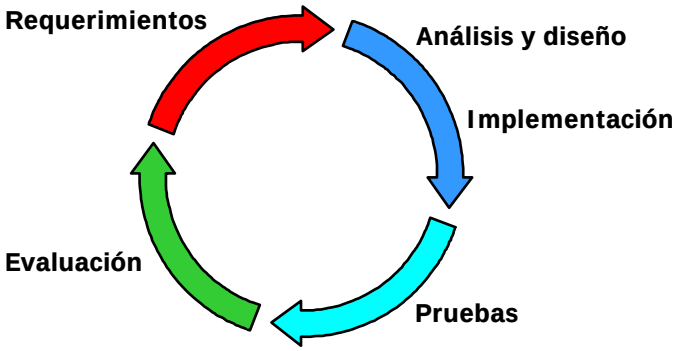

Fig. 3.6 Desarrollo iterativo [Letelier, 02]

3.3.3 Fases en el desarrollo iterativo

En el proceso unificado de desarrollo de software existen cuatro fases por las que se debe pasar en el desarrollo iterativo (ver figura 3.7). Estas fases son [Booch et al, 99]:

***Concepción**:* Durante esta fase se desarrollan ideas pensando en el producto final. Esencialmente, en esta fase se responden las siguientes preguntas:

- ¿Qué es lo que el sistema hará principalmente para cada uno de los usuarios potenciales?
- ¿Cuál es la forma de la arquitectura central del sistema?
- ¿Cuál es el plan y el costo de desarrollo del sistema?

Un modelo simplificado de casos de uso que contenga los casos más críticos responderá a la primera pregunta. En esta etapa, la arquitectura es tentativa, contiene solamente los subsistemas cruciales para su funcionamiento. En esta fase, se identifican los riesgos más importantes y se toman las medidas necesarias para resolverlos de alguna manera.

Elaboración*:* En esta fase se especifican en detalle la mayoría de los casos de uso y se diseña la arquitectura central. Al final de esta fase, el desarrollador está en la posición de planear las actividades y estimar los recursos necesarios para completar el proyecto. Los casos de uso, la arquitectura y los planes deben estar lo suficientemente estables y los riesgos deben estar bien controlados.

Construcción*:* Se empieza a construir cada parte del sistema siguiendo la arquitectura central antes diseñada. En esta fase, el sistema debe crecer lo suficiente para ponerlo a disposición de los usuarios para obtener opiniones y hacer los cambios pertinentes. La arquitectura debe ser estable, pero es posible que durante esta fase se descubran formas mejores de estructurarla; por lo que puede haber cambios menores. El sistema debe resolver satisfactoriamente todos los casos de uso que se hayan planteado al

principio del proyecto; aunque puede haber algunos defectos que se resolverán en la fase de transición. La pregunta es: ¿Cumple el sistema con la mayoría de las necesidades de los usuarios para liberar una primera versión?.

Transición: Esta fase cubre el período durante el cual el sistema se convierte en una versión beta. Durante esta fase, un cierto número de usuarios experimentados prueban el sistema y reportan los defectos y deficiencias encontradas. Los desarrolladores corrigen los problemas reportados e incorporan algunas mejoras sugeridas.

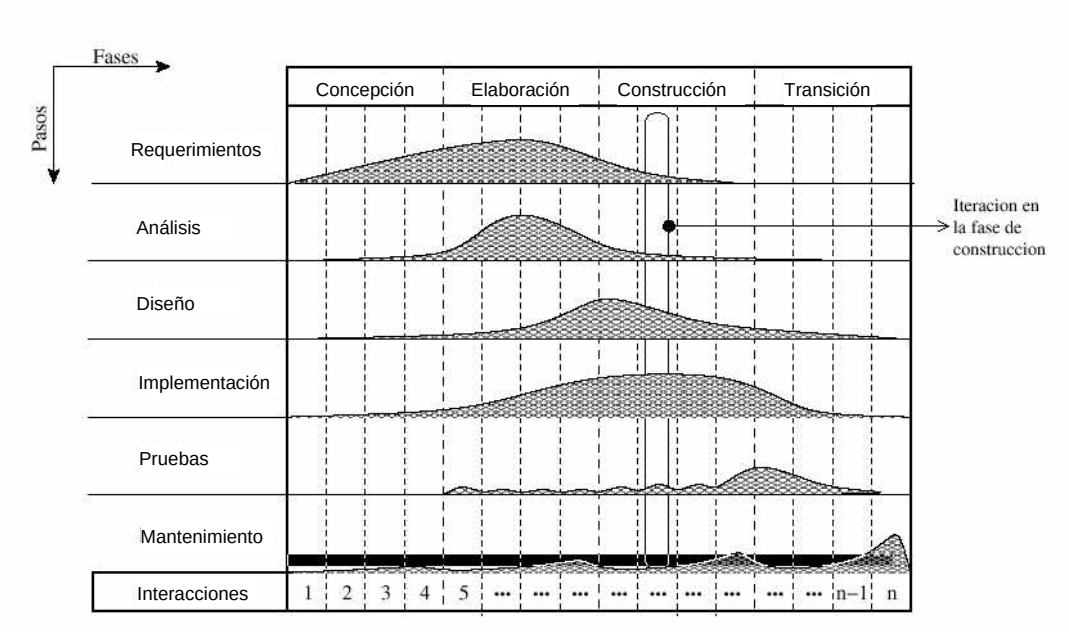

Fig. 3.7 Fases en el desarrollo iterativo incremental [Booch et al, 99]

3.4 Conclusiones

La tecnología orientada a objetos persigue el antiguo principio del divide y

vencerás. Su objetivo es descomponer la complejidad en partes más manejables y comprensibles. No parece que esto sea algo novedoso con respecto a la tradicional descomposición funcional de los métodos estructurados. Sin embargo, la gran diferencia reside en aplicar la dualidad estructura-función en pequeñas unidades capaces de comunicarse y reaccionar en base a la aparición de una serie de eventos.

Esté capitulo mostró las características que convierten a UML en un estándar en la construcción de modelos, y resalta la necesidad de combinarlo con un proceso de desarrollo de software para poder realizar un análisis y diseño que aproveche las ventajas que proporciona el paradigma OO. Este proyecto fue desarrollado combinando UML con el proceso unificado de desarrollo de software, siendo muy favorables los resultados obtenidos.