1. Ciclo de vida tradicional de los sistemas de software

En ingeniería de software, la descripción tradicional del ciclo de vida del software está basada en un modelo conocido como el modelo de cascada (*waterfall model*). Inicialmente el modelo planteaba la discretización de las actividades involucradas en el desarrollo de software y las presentaba como una serie lineal de tareas, donde cada tarea debía ser completada antes de comenzar la siguiente (figura 1).

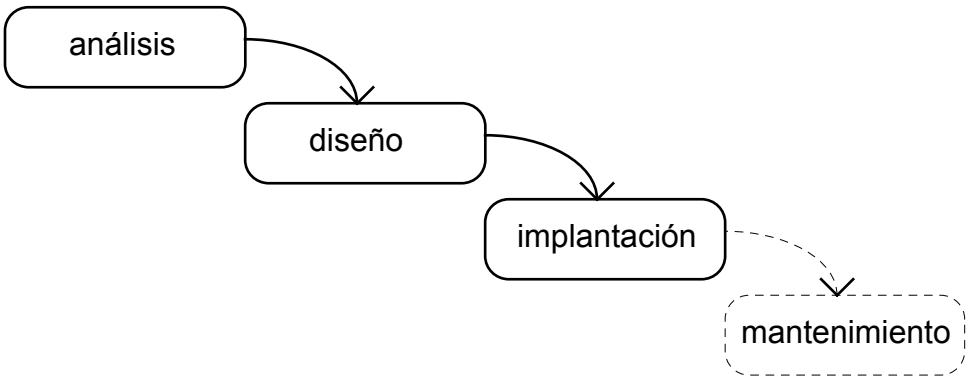

Fig. 1. Ciclo de vida tradicional

Frente al hecho de que en la realidad no es posible lograr este grado de completitud, posteriores variaciones del modelo reconocen la necesidad de iterar a través de las diferentas etapas (figura 2).

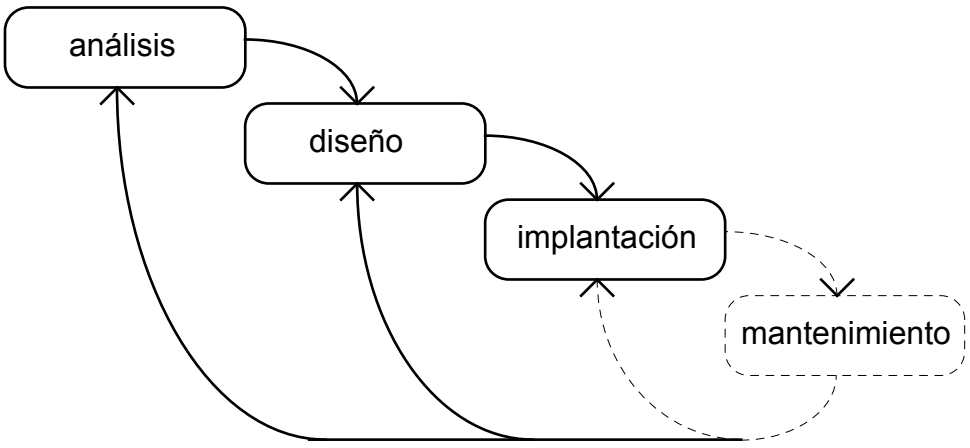

Fig. 2. Modelo de cascada revisado

Todos los autores en el área coinciden en cuatro etapas: análisis, diseño y construcción. La fase de análisis cubre basicamente el análisis de los requerimientos del usuario (dominio del problema) y un estudio de factibilidad/viabilidad. La fase de diseño cubre los diferentes conceptos del diseño de sistemas (dominio de la solución): diseño global o

diseño lógico y diseño del programa, conocido como diseño detallado o diseño físico. Una vez diseñada la aplicación se escribe el código correspondiente y se pone a prueba el programa. Aquí comienza la fase de mantenimiento donde se corrigen fallas y se adecúa el programa a nuevos requerimientos. A un nivel más detallado se identifican nuevas actividades tal como se ve en la siguiente figura.

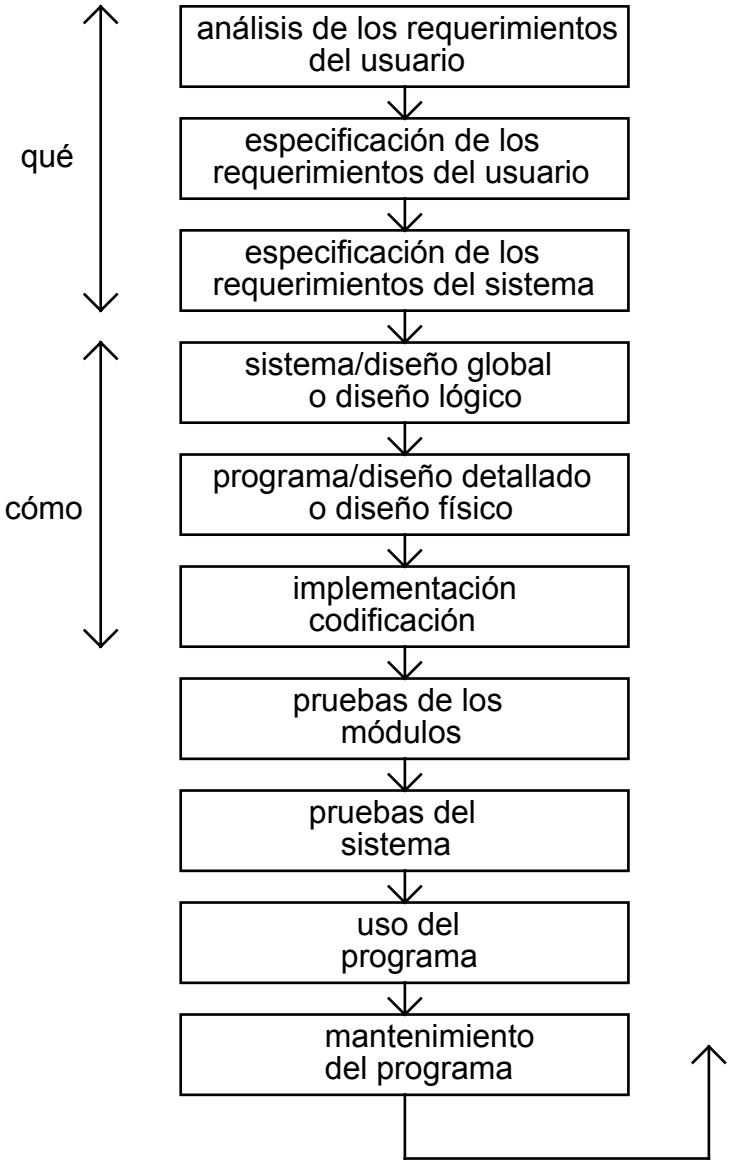

Fig. 4. Modelo de cascada a nivel detallado

Se comienza con la definición del problema y se identifican los requerimientos de los usuarios actuales y futuros, usualmente por medio de preguntas, discución con expertos en el área, etc.. Dependiendo de la importancia y volumen del sistema esta fase puede incluir un estudio de factibilidad/viabilidad. A partir de esto se escriben sendos

documentos con las especificaciones de requerimientos del usuario y con las especificaciones de requerimientos de la aplicación. La especificación de requerimientos del usuario se escribe en el lenguaje de los usuarios, lo que permite establecer un acuerdo sobre esta especificación entre los usuarios del sistema y los ingenieros del sistema. La especificación de requerimientos de la aplicación se escribe en el lenguaje de los programadores y establece en forma precisa los detalles del sistema. Estas actividades comprenden el análisis del espacio del problema (qué). A medida que avanzamos en el diagrama pasamos del análisis al diseño del sistema, esto es, del espacio del problema al espacio de la solución.

La etapa de diseño es la que está definida en forma más imprecisa, quizás porque es un proceso creativo, no mecánico, donde se hace una descomposición cada más detallada y progresiva del sistema. El diseño de sistemas es generalmente referido como "diseño global" o "diseño lógico" y el diseño del programa como "diseño detallado" o "diseño físico". En el ciclo de vida tradicional estas dos actividades de diseño pueden llegar a traslaparse y ser realizadas en forma iterativa. En el ciclo de vida orientado por objetos la diferenciación desaparece casi totalmente.

El ciclo de vida del software, tal como se ha descrito en los párrafos anteriores, es frecuentemente implementado basándose en una visión del mundo interpretada en términos de **descomposición funcional**, donde la pregunta fundamental a responder mediante las etapas de análisis y diseño es **qué** hace el sistema, esto es, cuál es su función. De esta manera, el diseño funcional (y las técnicas de descomposición funcional utilizadas para llevarlo a cabo) está basado en una interpretación funcional del espacio del problema y su traducción al espacio de la solución como un grupo de procedimientos interrelacionados. El sistema final es visto entonces como un conjunto de funciones que operan sobre los datos, aparentemente secundarios.

La descomposición funcional es también una metodología de análisis y diseño arriba-abajo (*top-down*) y es, en consecuencia, demasiado restrictiva para dar soporte al diseño de las aplicaciones contemporáneas. Meyer [Meyer] resume las desventajas de la metodología de diseño arriba-abajo de la siguiente forma:

1. La metodología arriba-abajo no tiene en cuenta el carácter evolutivo de los sistemas (i.e. cambio en las especificaciones, extensión, ...).

2. En la metodología arriba-abajo el sistema está caracterizado por una sola función, un concepto cuestionable.

3. En la metodología arriba-abajo el punto de vista es basicamente funcional y por lo tanto las estructuras de datos que el sistema manipula son dejadas en un plano secundario.

4. La metodología arriba-abajo no promueve la reutilización. (!)

Hemos hecho una revisión del enfoque tradicional del ciclo de vida del software, una metodología basicamente funcional. Ahora haremos una breve descripción de las tres opciones principales para el análisis y diseño de sistemas.

Descomposición Funcional

Según esta metodología, el diseñador debe centrar su atención en los procedimientos y algoritmos necesarios para que el sistema lleve a cabo su tarea. Inicialmente, el sistema es visto en términos de lo *que* debe hacer y luego en la etapa de diseño se determina *como* lo va a hacer. Las herramientas de diseño que dan soporte a esta metodología están basadas en el *flujo de datos* de la aplicación, e incluyen los Diagramas de Flujo de Datos (DFDs), los Diccionarios de Datos (DDs) y los Diagramas de Estructuras.

Existen varios lenguajes de programación que dan soporte a la descomposición funcional: Fortran, Cobol, Pascal, C, etc.. La tendencia en estos lenguajes de colocar los datos compartidos en áreas globales (e.g. *COMMON* en Fortran, *extern* en C) produce los llamados 'efectos colaterales' (*stack of dominoes*), tan conocidos por aquellos que trabajan en el desarrollo y mantenimiento de programas, donde los cambios hechos en una parte del sistema es fuente de problemas en otras áreas aparentemente disociadas. Por otra parte, un sistema diseñado con una única función en mente, dificilmente podrá asimilar los cambios evolutivos causados por modificaciones o extensiones.

Desarrollo Estructurado de Jackson

La Programación Estructurada de Jackson (Jackson Structured Programming) y el Desarrollo Estructurado de Jackson (JSD, Jackson Structured Development) [Jackson] son técnicas que modelan el mundo real con un efoque parcialmente orientado por objetos, donde se utilizan métodos basados en las estructuras de datos. Sin embargo, son esencialmente metodologías de descomposición funcional [Sommerville, p.185] en donde las estructuras de datos son utilizadas como soporte a esta descomposición. A diferencia de la descomposición funcional en donde las estructuras de datos están fundamentalmente determinadas por la estructura funcional, se puede decir que JSD es una metodología parcialmente orientada por objetos o de centro-afuera (*middle-out*), que hace menos énfasis en la parte funcional y comienza el análisis del sistema como un proceso de modelamiento.

Desarrollo Orientado por Objetos

El paradigma Orientado por Objetos (OO) utiliza los mismos componentes de los sistemas de software: datos y procedimientos. Sin embargo, desenfatiza los procedimientos y hace énfasis en el encapsulamiento de datos y procedimientos en entidades con una interfaz claramente definida. En la descomposición de sistemas basada en un enfoque OO, el sistema es visto como una colección de entidades (objetos) que encapsulan cierto conocimiento (datos) y cierto comportamiento (procedimientos), y que interactúan para realizar la tarea o tareas del caso. Las etapas de análisis y diseño se llevan a cabo con base en estos objetos y en los servicios que éstos prestan, utilizando un

modelo cliente-servidor donde los objetos interactúan por medio de mensajes. El uso de este modelo cliente-servidor hace que el sistema sea descrito como 'orientado a responsabilidades' (*responsibilty-driven*). El diseño detallado, incluyendo la implementación de procedimientos y definición de estructuras de datos, se difiere hasta una etapa más avanzada del proceso de desarrollo y es privada a cada clase de objetos, adhiriendo estrictamente al concepto de ocultamiento de información. De esta forma los procedimientos y estructuras de datos no quedan 'congelados' desde las primeras etapas del diseño, cuando todavía no se tiene un conocimiento profundo del sistema. Un sistema representado en función de objetos es por lo tanto más flexible y la vez menos sensible a los cambios estructurales. Es importante que las estructuras de datos no sean especificadas muy temprano en el proceso de diseño. En consecuencia el desarrollo orientado por objetos se centra en la abstracción de datos en vez de fijar una estructura determinada en la especificación de los objetos que componen el sistema.

En contraste con el enfoque de descomposición funcional arriba-abajo, el enfoque OO si bien tiene atributos arriba-abajo está caracterizado por una metodología abajo-arriba, donde el fin principal está en generar clases de objetos que puedan ser incluídas en bibliotecas de uso general. Un enfoque que involucra una técnica de análisis arriba-abajo y una técnica de diseño abajo-arriba es probable que dé como resultado sistemas más robustos. En esta metodología la línea divisoria entre las diferentes etapas del ciclo de vida del software (A/D/I) desaparecen, tornándose un proceso contínuo e iterativo donde analistas, diseñadores y programadores trabajan con entidades comunes: objetos.

Por último, una de las características que diferencian radicalmente la metodología OO de la metodología tradicional basada en la descomposición funcional está en los objetivos de cada una. Mientras que en el enfoque tradicional el fin es construir un sistema que de solución a un problema determinado (y este sistema se construye desde cero), en el enfoque OO el fin principal está en generar clases de objetos, que en conjunto den la solución requerida, pero que además puedan ser incluídas en bibliotecas para su futura **reutilización**. La reutilización de componentes probados y optimizados es lo que marca la verdadera diferencia entre las metodologías y se traduce en un aumento considerable de la productividad.