Índice general

1. Contexto de la asignatura en la Ingeniería de Software **1**
1.1. Necesidad de una metodología 1
1.1.1. Sistemas 1
1.1.2. La crisis del software 2
1.1.3. Definición de metodología 2
1.1.4. Finalidad de una metodología 3
1.1.5. Taxonomía de las metodologías 3
1.2. Ciclo de vida del software 5
1.2.1. Ciclos de vida en cascada 7
1.2.2. Modelo de ciclo de vida en espiral 10
1.2.3. Ciclos de vida orientados a objetos 13
1.3. Notaciones de especificación y diseño (UML) 14
1.3.1. Introducción 14
1.3.2. Modelo de casos de uso 17
1.3.3. Modelo estructural estático 19
1.3.4. Modelo de comportamiento 30
1.3.5. Modelo estructural de implementación 36
Ejercicios y actividades 39

2. Fase de especificación **41**
2.1. Obtención de requisitos 41
2.1.1. Introducción 41
2.1.2. Técnicas de obtención de requisitos 43
2.2. Análisis de requisitos 57
2.3. Representación de requisitos 58
2.4. Análisis orientado a objetos 58
2.5. Bases de documentación 58
Ejercicios y actividades 64

3. Fase de diseño **65**
3.1. Conceptos y elementos del diseño 65
3.1.1. Patrones 65
3.2. Diseño estructurado 72

3.3. Diseño orientado a objetos . . . 72
3.4. Validación y confirmación del diseño . . . 75
3.4.1. Revisión del diseño . . . 75
3.4.2. Verificación del diseño . . . 75
3.4.3. Validación del diseño . . . 75
3.5. Documentación: especificación del diseño . . . 75
Ejercicios y actividades . . . 78

4. Fase de implementación 79
4.1. Técnicas de depuración . . . 79
4.2. Documentación del código . . . 79
4.2.1. Tipos de comentarios . . . 79
4.2.2. Consideraciones generales . . . 80
Ejercicios . . . 81

5. Fases de pruebas 83
5.1. Técnicas y métodos de prueba . . . 83
5.2. Documentación de pruebas . . . 83
Ejercicios . . . 86

6. Fase de entrega y mantenimiento 89
6.1. Finalización del proyecto . . . 89
6.1.1. Aceptación . . . 90
6.1.2. Informe de cierre del proyecto . . . 90
6.1.3. Indicadores del proyecto . . . 90
6.2. Planificación de revisiones y organización del mantenimiento . . . 92
6.3. Recopilación y organización de documentación . . . 92
6.3.1. Motivos de la documentación . . . 92
6.3.2. Directrices que se deben seguir para la redacción de un documento . . . 93
6.3.3. Tipos de documentos . . . 93
6.3.4. Manual de usuario . . . 95
6.3.5. Manual del sistema . . . 97
6.3.6. Manual de mantenimiento . . . 100
Ejercicios y actividades . . . 102

7. Metodologías de desarrollo 103
7.1. Introducción a las metodologías de desarrollo . . . 103
7.2. Proceso unificado de Rational . . . 103
7.2.1. Introducción . . . 104
7.2.2. Las cuatro “P”: Personas, Proyecto, Producto y Proceso . . . 106
7.2.3. Proceso dirigido por casos de uso . . . 107
7.2.4. Proceso centrado en la arquitectura . . . 108
7.2.5. Proceso iterativo e incremental . . . 109
7.2.6. Captura de requisitos . . . 111

7.2.7. Diseño . . . 117
7.2.8. Implementación . . . 120
7.2.9. Prueba . . . 122
7.3. Método extreme programming . . . 124
7.3.1. Historias de usuario . . . 125
7.3.2. Plan de publicación de versiones . . . 125
7.3.3. Tarjetas CRC: Clases, Responsabilidades y Colaboraciones . . . 126
7.3.4. Planificación de cada iteración . . . 126
7.3.5. Integración . . . 127
7.3.6. Codificación de cada unidad . . . 128
7.3.7. Recomendaciones generales . . . 130
7.4. Métrica 3 . . . 132
7.4.1. Introducción . . . 132
7.4.2. Objetivos . . . 132
7.4.3. Estructura . . . 132
7.4.4. Planificación de Sistemas de Información . . . 133
7.4.5. Estudio de la viabilidad del sistema . . . 135
7.4.6. Análisis del sistema de información . . . 136
7.4.7. Diseño del sistema de información . . . 138
7.4.8. Construcción del sistema de información . . . 141
7.4.9. Implantación y Aceptación del Sistema . . . 143
7.4.10. Mantenimiento de Sistemas de Información . . . 144
7.5. Métodos de software libre: “cathedral” vs. “bazaar” . . . 154
7.5.1. La catedral . . . 154
7.5.2. El bazar . . . 155
Ejercicios y actividades . . . 157

8. Herramientas de desarrollo y validación 159
8.1. Herramientas CASE . . . 159
8.1.1. Funciones de las herramientas CASE . . . 159
8.1.2. Clasificación de las herramientas CASE . . . 160
8.2. Gestión de la configuración . . . 162
8.2.1. Terminología y definiciones básicas . . . 162
8.2.2. Identificación de la configuración . . . 163
8.2.3. Control de cambios . . . 165
8.2.4. Generación de informes de estado . . . 165
8.2.5. Auditoría de la configuración . . . 166
8.2.6. Plan de gestión de la configuración . . . 166
8.2.7. Herramientas de la Gestión de la Configuración . . . 167
8.2.8. Software libre . . . 169
8.3. Entornos de desarrollo de interfaces . . . 171
8.3.1. Introducción . . . 171
8.3.2. Componentes . . . 172

8.3.3. Creación de interfaces de usuario 173
8.3.4. Metodología 174
8.3.5. Heurísticas de usabilidad 175
8.3.6. Glade 175
8.3.7. GTK+ 176
8.3.8. Anjuta 176
Ejercicios y actividades 178

Bibliografía **181**

Índice alfabético **182**

Capítulo 1

Contexto de la asignatura en la Ingeniería de Software

1.1. Necesidad de una metodología

El proceso de construcción del software requiere, como en cualquier otra ingeniería, identificar las tareas que se han de realizar sobre el software y aplicar esas tareas de una forma ordenada y efectiva. Adicionalmente y aunque no es el objeto principal de esta asignatura, el desarrollo del software es realizado por un conjunto coordinado de personas simultáneamente y, por lo tanto, sus esfuerzos deben estar dirigidos por una misma metodología que estructure las diferentes fases del desarrollo. En temas posteriores, en primer lugar se explicarán en detalle las mencionadas fases y a continuación las metodologías usuales que las gestionan.

En esta asignatura se hará especial énfasis en los temas relacionados con el análisis, diseño y mantenimiento del software, mencionando apenas los temas específicos de la organización y gestión de proyectos que conciernen a la distribución de medios y tareas entre las personas a cargo del desarrollo, ya que estos últimos temas son objeto de otras asignaturas de la titulación.

1.1.1. Sistemas

La **Ingeniería de Sistemas** es el contexto genérico en que se sitúan las herramientas y metodologías usadas para crear sistemas. Un sistema puede estar formado por subsistemas de diferentes tipos. La ingeniería de sistemas constituye el primer paso de toda ingeniería: proporciona una visión global de un sistema en su conjunto para que, posteriormente, cada uno de sus subsistemas se analice con la rama de la ingeniería adecuada. Una de esas ramas es la ingeniería del software.

La **Ingeniería del Software** trata, según Bauer [Bau72], del establecimiento de los principios y métodos de la ingeniería, orientados a obtener software económico, que sea fiable y funcione de manera eficiente sobre máquinas reales.

El sistema es un conjunto de elementos que cooperan entre sí para proporcionar una funcionalidad. En el caso de un sistema informático, consta de dos tipos de elementos: hardware y software.

1.1.2. La crisis del software

Desde el momento en que se introdujeron computadores con capacidad para soportar aplicaciones de tamaño considerable (años 60), las técnicas de desarrollo para los hasta entonces pequeños sistemas dejaron progresivamente de ser válidas. Estas técnicas consistían básicamente en codificar y corregir (*code-and-fix*) que se resume en lo siguiente:

> No existe necesariamente una especificación del producto final, en su lugar se tienen algunas anotaciones sobre las características generales del programa. Inmediatamente se empieza la codificación y simultáneamente se va depurando. Cuando el programa cumple con las especificaciones y parece que no tiene errores se entrega.

La ventaja de este enfoque es que no se gasta tiempo en planificación, gestión de los recursos, documentación, etc. Si el proyecto es de un tamaño muy pequeño y lo realiza un equipo pequeño (una sola persona o dos) no es un mal sistema para producir un resultado pronto. Hoy en día es un método de desarrollo que se usa cuando hay plazos muy breves para entregar el producto final y no existe una exigencia explícita por parte de la dirección de usar alguna metodología de ingeniería del software. Puede dar resultado, pero la calidad del producto es imprevisible. Las consecuencias de este tipo de desarrollo son:

1. El costo es mucho mayor de lo originalmente previsto.
2. El tiempo de desarrollo excede lo proyectado.
3. La calidad del código producido es imprevisible y en promedio baja.

Aunque se han desarrollado técnicas para paliar estos problemas, hoy en día aún se considera normal que una aplicación rebase sus proyecciones iniciales de tiempo y dinero y que se descubran *bugs* (errores informáticos) en su ejecución. Esto se debe a que todavía no se aplica de un modo sistemático la ingeniería del software durante el desarrollo.

1.1.3. Definición de metodología

En la literatura sobre este tema existen muchas definiciones sobre lo que es una metodología. El común denominador de todas ellas es la siguiente lista de características:

1. Define cómo se divide un proyecto en fases y las tareas que se deben realizar en cada una de ellas.
2. Especifica para cada una de las fases cuáles son las entradas que recibe y las salidas que produce.
3. Establece alguna forma de gestionar el proyecto.

Resumiendo lo anterior definimos *metodología* como un modo sistemático de producir software.

1.1.4. Finalidad de una metodología

La diferencia entre *code-and-fix* (no usar ninguna metodología) y usar una es que se pueden alcanzar los siguientes atributos en el producto final:

1. *Adecuación*: el sistema satisface las expectativas del usuario.
2. *Mantenibilidad*: facilidad para realizar cambios una vez que el sistema está funcionando en la empresa del cliente.
3. *Usabilidad*: facilidad de aprender a manejar el sistema por parte de un usuario que no tiene por qué ser informático. La resistencia de los usuarios a aceptar un sistema nuevo será mayor cuanto peor sea la usabilidad.
4. *Fiabilidad*: capacidad de un sistema de funcionar correctamente durante un intervalo de tiempo dado. La diferencia con la corrección es que en este atributo interesa el tiempo, es decir, no se trata del número absoluto de defectos en el sistema sino de los que se manifiestan en un intervalo de tiempo. Interesan sobre todo:

   a) *MTBF* (*Mean Time Between Failures*): tiempo medio entre fallos.

   b) *Disponibilidad*: probabilidad de que el sistema esté funcionando en un instante dado.

5. *Corrección*: baja densidad de defectos.
6. *Eficiencia*: capacidad del sistema de realizar su tarea con el mínimo consumo de recursos necesario.

1.1.5. Taxonomía de las metodologías

Existen dos grupos de metodologías en función de la mentalidad con la que aborda un problema: metodología estructurada y metodología orientada a objetos.

Metodología estructurada

Constituyó la primera aproximación al problema del desarrollo de software. Está orientada a procesos, es decir, se centra en especificar y descomponer la funcionalidad del sistema. Se utilizan varias herramientas:

- *Diagramas de flujo de datos (DFD)*: representan la forma en que los datos se mueven y se transforman. Incluyen:

  - Procesos
  - Flujos de datos
  - Almacenes de datos

  Los procesos individuales se pueden a su vez ”explosionar” en otros DFD de mayor nivel de detalle.

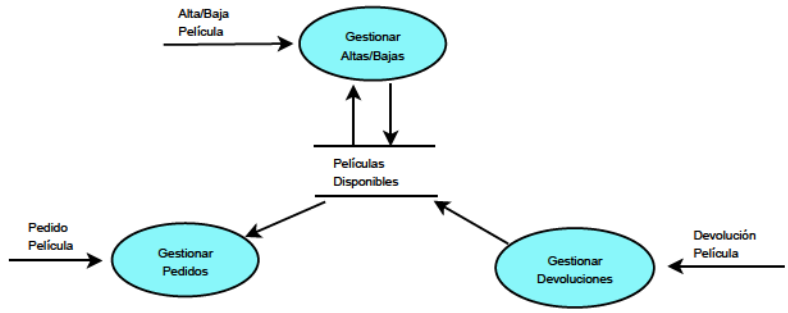

Figura 1.1: Ejemplo de DFD

- *Especificaciones de procesos*: descripciones de los procesos definidos en un DFD que no se puede descomponer más. Pueden hacerse en pseudocódigo, con tablas de decisión o en un lenguaje de programación.
- *Diccionario de datos*: nombres de todos los tipos de datos y almacenes de datos junto con sus definiciones.
- *Diagramas de transición de estados*: modelan procesos que dependen del tiempo.
- *Diagramas entidad-relación*: los elementos del modelo E/R se corresponden con almacenes de datos en el DFD. En este diagrama se muestran las relaciones entre dichos elementos.

Los lenguajes de programación también reflejan esta dicotomía que existe entre las metodologías; así, existen lenguajes para la programación estructurada. Los más famosos son: Cobol (destinado a aplicaciones financieras, en su mejor momento el 90 % del código estaba escrito en este lenguaje), Fortran (usado en aplicaciones matemáticas), C (aplicaciones de propósito general en los años 80), Pascal y Modula 2.

Metodología orientada a objetos

Constituye una aproximación posterior. Cuenta con mayor número de desarrolladores y es previsible que termine sustituyendo a la anterior. Además, es ampliamente admitido que proporciona estas ventajas:

1. Está basada en componentes, lo que significa que facilita la reutilización de código. De todos modos hay que añadir que la reutilización de código es un tema complejo y que requiere en cualquier caso un diseño muy cuidadoso. Una metodología orientada a objetos no garantiza por si misma la producción de código reutilizable, aunque lo facilita.
2. Simplifica el mantenimiento debido a que los cambios están más localizados.

La mentalidad que subyace al diseño estructurado es: ¿Cómo se puede dividir el sistema en partes más pequeñas que puedan ser resueltas por **algoritmos** sencillos y qué información se intercambian?. En el diseño orientado a objetos la idea es sin embargo: ¿Cuáles son los tipos de **datos** que hay que utilizar, qué características tienen y cómo se relacionan?.

La orientación a objetos supone un paradigma distinto del tradicional (no necesariamente mejor o peor) que implica focalizar la atención en las estructuras de datos. El concepto de objetos tuvo sus orígenes en la inteligencia artificial como un modo de representación del conocimiento. El primer lenguaje orientado a objetos fue **Simula67**, desarrollado por Kristen Nggaard y Ole-Johan Dahl en el Centro de Cálculo noruego, pero el que se considera el primer lenguaje orientado a objetos puro fue **Smaltalk**, donde todos los elementos del lenguaje son objetos. El lenguaje **C++** fue una ampliación de C para que soportara objetos; resultó muy eficiente pero también muy complejo. **Java** es otro lenguaje orientado a objetos derivado del C++ pero más sencillo y concebido con la idea de minimizar los errores relativos a punteros.

Objetos y clases Un objeto consta de una estructura de datos y de una colección de métodos (antes llamados procedimientos o funciones) que manipulan esos datos. Los datos definidos dentro de un objeto son sus atributos. Un objeto sólo puede ser manipulado a través de su interfaz, esto es, de una colección de funciones que implementa y que son visibles desde el exterior.

Los conceptos importantes en el contexto de clases y objetos se pueden consultar en [Pre01, sec. 20.1] y [Pre01, sec. 20.2]. En resumen son:

1. *Clases*: describen abstracciones de datos y operaciones necesarias para su manipulación. Dentro de una clase se definen:
   - *Atributos*: los datos contenidos en la clase.
   - *Métodos*: los algoritmos internos de la clase.
2. *Polimorfismo*: capacidad de un objeto de presentar varios comportamientos diferentes en función de cómo se utilice; por ejemplo, se pueden definir varios métodos con el mismo nombre pero diferentes argumentos.
3. *Herencia*: relación de generalización; cuando varias clases comparten características comunes, éstas se ponen en una clase antecesora.
4. *Asociación*: relación entre clases que cooperan con alguna finalidad.

Durante la etapa de análisis se **identifican** los objetos del dominio del problema. En el diseño se **definen** las características de los objetos.

1.2. Ciclo de vida del software

Al igual que otros sistemas de ingeniería, los sistemas de software requieren un tiempo y esfuerzo considerable para su desarrollo y deben permanecer en uso por un periodo mucho mayor. Durante este tiempo de desarrollo y uso, desde que se detecta la necesidad de construir un sistema de software hasta que éste es retirado, se identifican varias etapas (o fases) que en conjunto se denominan ciclo de vida del software. En cada caso, en función de cuáles sean las características del proyecto, se configurará el ciclo de vida de forma diferente. Usualmente se consideran las fases: especificación y análisis de *requisitos*, *diseño* del sistema, *implementación* del software, aplicación y *pruebas*, entrega y *mantenimiento*. Un aspecto esencial dentro de las

tareas del desarrollo del software es la *documentación* de todos los elementos y especificaciones en cada fase. Dado que esta tarea siempre estará influida por la fase del desarrollo en curso, se explicará de forma distribuida a lo largo de las diferentes fases como un apartado especial para recalcar su importancia en el conjunto del desarrollo del software.

Tal como ya hemos mencionado, las fases principales en cualquier ciclo de vida son:

1. *Análisis*: se construye un modelo de los requisitos
2. *Diseño*: partiendo del modelo de análisis se deducen las estructuras de datos, la estructura en la que descompone el sistema, y la interfaz de usuario.
3. *Codificación*: se construye el sistema. La salida de esta fase es código ejecutable.
4. *Pruebas*: se comprueba que se cumplen criterios de corrección y calidad.
5. *Mantenimiento*: se asegura que el sistema siga funcionando y adaptándose a nuevos requisitos.

Las etapas se dividen en actividades que constan de tareas. La *documentación* es una tarea importante que se realiza en todas las etapas y actividades. Cada etapa tiene como entrada uno o varios documentos procedentes de las etapas anteriores y produce otros documentos de salida según se muestra en la figura 1.2.

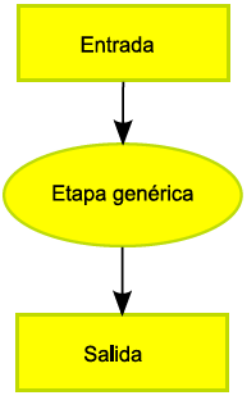

Figura 1.2: Etapa genérica

Algunos autores dividen la fase del diseño en dos partes: *diseño global* o arquitectónico y *diseño detallado*. En el primero se transforman los requisitos en una arquitectura de alto nivel, se definen las pruebas que debe satisfacer el sistema en su conjunto, se esboza la documentación y se planifica la integración. En el detallado, para cada módulo se refina el diseño y se definen los requisitos del módulo y su documentación.

Las formas de organizar y estructurar la secuencia de ejecución de las actividades y tareas en las diferentes fases de cada uno de los métodos pueden dar lugar a un tipo de ciclo de vida diferente. Los principales ciclos de vida que se van a presentar a continuación realizan todas las fases, actividades y tareas. Cada uno de ellos tiene sus ventajas e inconvenientes.

1.2.1. Ciclos de vida en cascada

El ciclo de vida inicialmente propuesto por Royce en 1970, fue adaptado para el software a partir de ciclos de vida de otras ramas de la ingeniería. Es el primero de los propuestos y el más ampliamente seguido por las organizaciones (se estima que el 90 % de los sistemas han sido desarrollados así). La estructura se muestra en la figura 1.3.

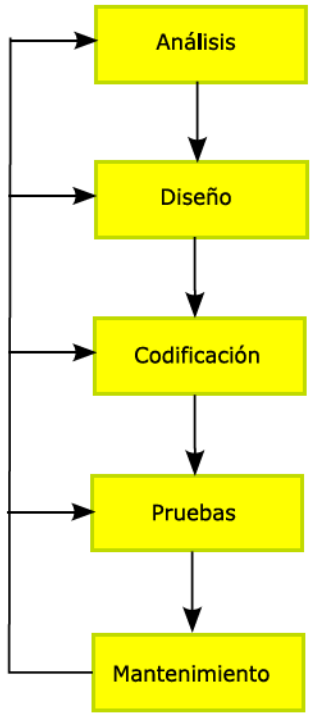

Figura 1.3: Ciclo de vida en cascada

Descripción

Este modelo admite la posibilidad de hacer iteraciones. Así por ejemplo, si durante el mantenimiento se ve la necesidad de cambiar algo en el diseño se harán los cambios necesarios en la codificación y se tendrán que realizar de nuevo las pruebas, es decir, si se tiene que volver a una de las etapas anteriores al mantenimiento se recorrerán de nuevo el resto de las etapas.

Después de cada etapa se realiza una revisión para comprobar si se puede pasar a la siguiente.

Trabaja en base a documentos, es decir, la entrada y la salida de cada fase es un tipo de documento específico. Idealmente, cada fase podría hacerla un equipo diferente gracias a la documentación generada entre las fases. Los documentos son:

- Análisis: Toma como entrada una descripción en lenguaje natural de lo que quiere el cliente. Produce el S.R.D. (Software Requirements Document).
- Diseño: Su entrada es el S.R.D. Produce el S.D.D. (Software Design Document)

- Codificación: A partir del S.D.D. produce módulos. En esta fase se hacen también pruebas de unidad.
- Pruebas: A partir de los módulos probados se realizan la integración y pruebas de todo el sistema. El resultado de las pruebas es el producto final listo para entregar.

Ventajas

- La planificación es sencilla.
- La calidad del producto resultante es alta.
- Permite trabajar con personal poco cualificado.

Inconvenientes

- Lo peor es la necesidad de tener todos los requisitos al principio. Lo normal es que el cliente no tenga perfectamente definidas las especificaciones del sistema, o puede ser que surjan necesidades imprevistas.
- Si se han cometido errores en una fase es difícil volver atrás.
- No se tiene el producto hasta el final, esto quiere decir que:
  - Si se comete un error en la fase de análisis no se descubre hasta la entrega, con el consiguiente gasto inútil de recursos.
  - El cliente no verá resultados hasta el final, con lo que puede impacientarse .
- No se tienen indicadores fiables del progreso del trabajo (síndrome del 90 %).[1]
- Es comparativamente más lento que los demás y el coste es mayor también.

Tipos de proyectos para los que es adecuado

- Aquellos para los que se dispone de todas las especificaciones desde el principio, por ejemplo, los de reingeniería.
- Se está desarrollando un tipo de producto que no es novedoso.
- Proyectos complejos que se entienden bien desde el principio.

Como el modelo en cascada ha sido el más seguido ha generado algunas variantes. Ahora veremos algunas.

Ciclo de vida en V

Propuesto por Alan Davis, tiene las mismas fases que el anterior pero tiene en consideración el nivel de abstracción de cada una. Una fase además de utilizarse como entrada para la siguiente, sirve para validar o verificar otras fases posteriores. Su estructura está representada en la figura 1.4.

[1] Consiste en creer que ya se ha completado el 90 % del trabajo, pero en realidad queda mucho más porque el 10 % del código da la mayor parte de los problemas

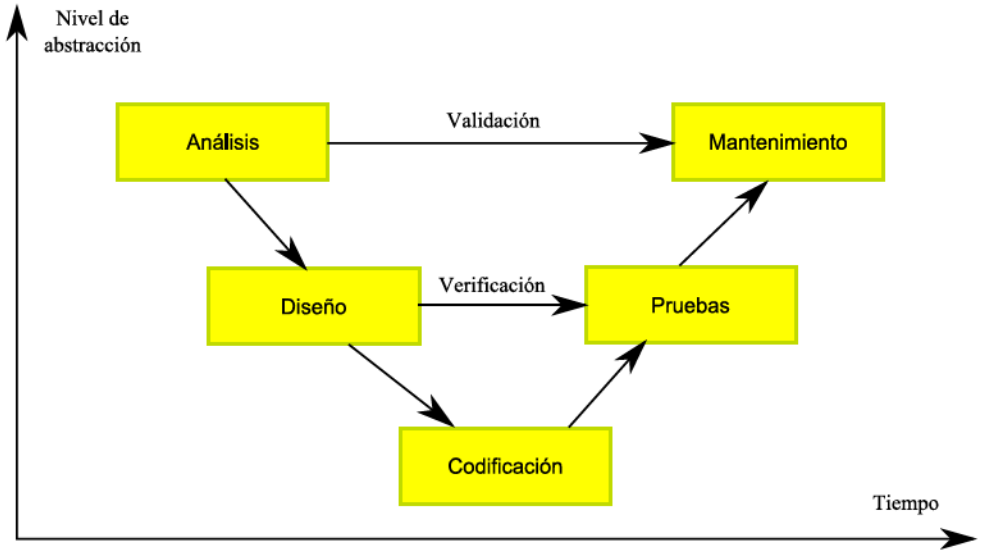

Figura 1.4: Ciclo de vida en V

Ciclo de vida tipo sashimi

Según el modelo en cascada puro una fase sólo puede empezar cuando ha terminado la anterior. En este caso, sin embargo, se permite un solapamiento entre fases. Por ejemplo, sin tener terminado del todo el diseño se comienza a implementar. El nombre "sashimi" deriva del estilo de presentación en rodajas de pescado crudo en Japón. Una ventaja de este modelo es que no necesita generar tanta documentación como el ciclo de vida en cascada puro debido a la continuidad del mismo personal entre fases. Los problemas planteados son:

- Es aún más difícil controlar el progreso del proyecto debido a que los finales de fase ya no son un punto de referencia claro.
- Al hacer cosas en paralelo, si hay problemas de comunicación pueden surgir inconsistencias.

La fase de "concepto" consiste en definir los objetivos del proyecto, beneficios, tipo de tecnología y tipo de ciclo de vida. El diseño arquitectónico es el de alto nivel, el detallado el de bajo nivel. En la figura 1.5 se ha representado la estructura del ciclo de vida sashimi.

Ciclo de vida en cascada con subproyectos

Si, una vez que se ha llegado al diseño arquitectónico, se comprueba que el sistema se divide en varios subsistemas independientes entre sí, sería razonable suponer que a partir de ese punto cada uno se puede desarrollar por separado y en consecuencia en paralelo con los demás. Cada uno tendrá seguramente fechas de terminación distintas. Una vez que han terminado todos se integran y se prueba el sistema en su conjunto. La ventaja es que se puede tener a más gente

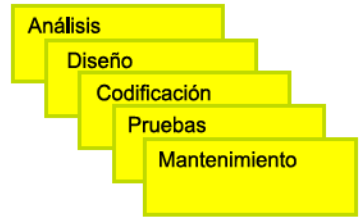

Figura 1.5: Ciclo de vida sashimi

trabajando en paralelo de forma eficiente. El riesgo es que existan interdependencias entre los subproyectos.

Ciclo de vida en cascada incremental

En este caso se va creando el sistema añadiendo pequeñas funcionalidades. Cada uno de los pequeños incrementos es comparable a las modificaciones que ocurren dentro de la fase de mantenimiento. La ventaja de este método es que no es necesario tener todos los requisitos en un principio. El inconveniente es que los errores en la detección de requisitos se encuentran tarde.

Hay dos partes en el ciclo de vida, que lo hacen similar al ciclo de vida tipo sashimi. Por un lado están el análisis y el diseño global. Por otro lado están los pequeños incrementos que se llevan a cabo en las fases de diseño detallado, codificación y mantenimiento.

Ciclo de vida en cascada con reducción de riesgos

Como se ha comentado anteriormente, uno de los problemas del ciclo de vida en cascada es que si se entienden mal los requisitos esto sólo se descubrirá cuando se entregue el producto. Para evitar este problema se puede hacer un desarrollo iterativo durante las fases de análisis y diseño global. Esto consistiría en:

1. Preguntar al usuario.
2. Hacer el diseño global que se desprende del punto 1.
3. Hacer un prototipo de interfaz de usuario, hacer entrevistas con los usuarios enseñándoles el prototipo y volver con ello al punto 1 para identificar más requisitos o corregir malentendidos.

El resto es igual al ciclo de vida en cascada.

1.2.2. Modelo de ciclo de vida en espiral

Propuesto inicialmente por Boehm en 1988. Consiste en una serie de ciclos que se repiten. Cada uno tiene las mismas fases y cuando termina da un producto ampliado con respecto al ciclo anterior. En este sentido es parecido al modelo incremental, la diferencia importante es que tiene en cuenta el concepto de riesgo. Un riesgo puede ser muchas cosas: requisitos no comprendidos,

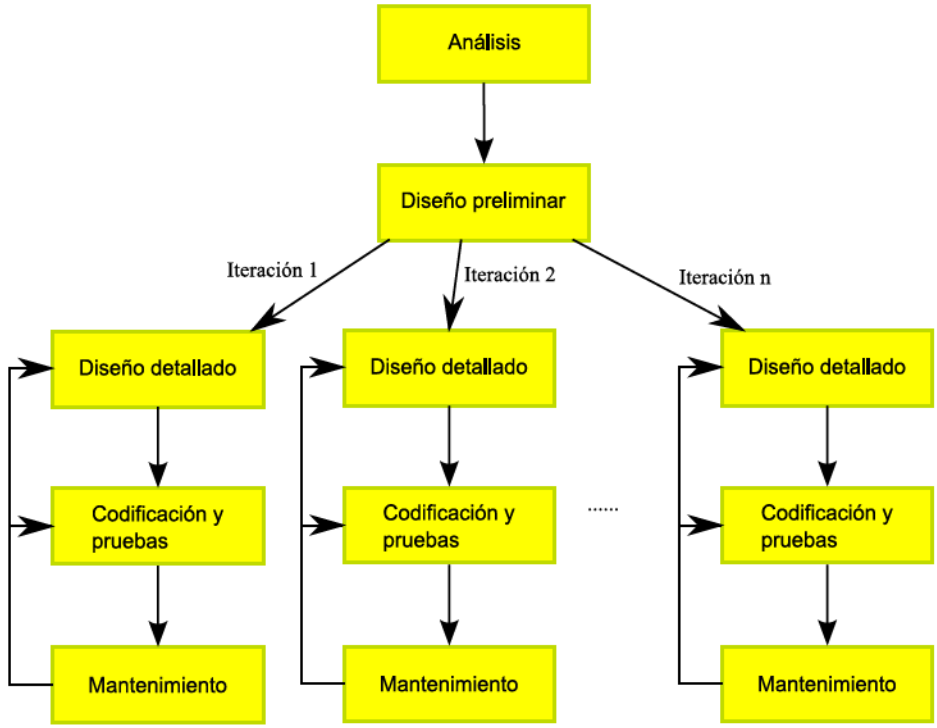

Figura 1.6: Cascada incremental

mal diseño, errores en la implementación, etc. Un representación típica de esta estructura se muestra en la figura 1.7.

En cada iteración Boehm recomienda recopilar la siguiente lista de informaciones:

- *Objetivos*: se obtienen entrevistando a los clientes, haciendo cuestionarios, etc.
- *Alternativas*: son las diferentes formas posibles de conseguir los objetivos. Se consideran desde dos puntos de vista
  - Características del producto.
  - Formas de gestionar el proyecto.
- *Restricciones*: son condiciones o limitaciones que se deben cumplir. Hay dos tipos:
  - Desde el punto de vista del producto: interfaces de tal o cual manera, rendimiento, etc.
  - Desde el punto de vista organizativo: coste, tiempo, personal, etc.
- *Riesgos*: es una lista de peligros de que el proyecto fracase.
- *Resolución de riesgos*: son las posibles soluciones al problema anterior. La técnica más usada es la construcción de prototipos.
- *Resultados*: es el producto que queda después de la resolución de riesgos.