

FORMALIZACIÓN DEL PROCESO DE DESARROLLO

El ciclo de vida del Software. Modelos de los ciclos de vida.

El ciclo de vida del software se define como el conjunto de etapas por las que pasa el sistema desde su concepción hasta su retirada de servicio, pasando por su desarrollo y mantenimiento.

NO EXISTE un único modelo de ciclo de vida, no obstante todo ciclo de vida debe cubrir los siguientes objetivos básicos:

- Definir las actividades a realizar y en que orden
- Establecer criterios de transición para pasar a la fase siguiente.
- Proporcionar puntos de control para la gestión del proyecto.
- Asegurar la consistencia con el resto de los sistemas de información.

Los modelos de ciclo de vida se pueden clasificar, según algunos autores en dos grandes grupos:

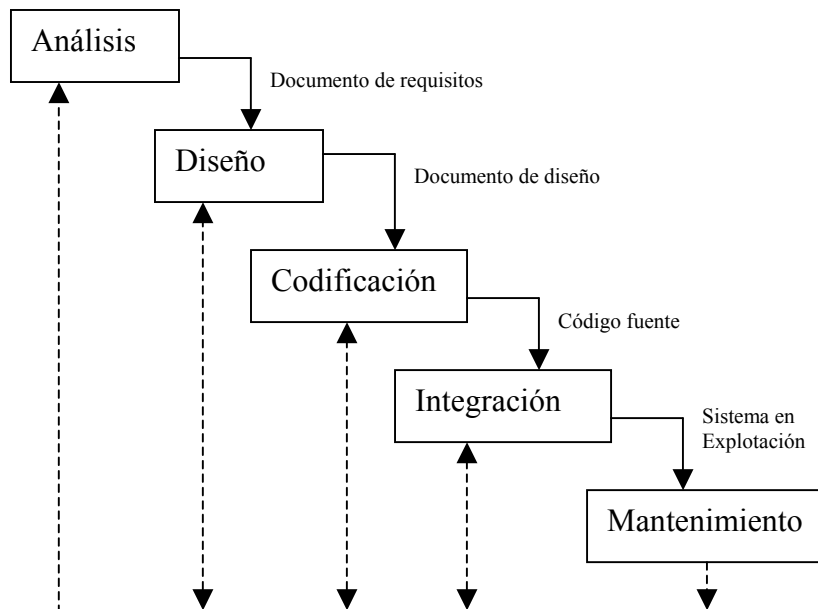
- Modelos tradicionales
- Modelos alternativos

Modelos Tradicionales

Modelo de ciclo de vida clásico o en cascada

Antes de que Royce presentara el modelo de ciclo de vida en cascada, los modelos de desarrollo del software se basaban principalmente en el modelo CODE & FIX (codificar y mejorar) y en el modelo por etapas, que recibe el nombre de (stage-wise-Model). El modelo en cascada introduce una serie de mejoras respecto al modelo por etapas, tales como:

- La realización de bucles de realimentación entre etapas, permitiendo que se puedan resolver los problemas detectados que provengan de una fase anterior
- En el se identifican ya prácticamente todas las clases de actividades distintas que intervienen en el desarrollo y explotación del software.



Existen diferentes variantes de este modelo, que se diferencian en el reconocimiento o no como fases separadas de ciertas actividades, de manera que lo que una variante se palntea globalmente como una sola fase, en otra puede desglosarse en una secuencia de dos o tres fases consecutivas. Según este modelo tendríamos las siguientes fases:

1ª Fase Análisis: Consiste en la recopilación de los requisitos del software. Se debe comprender el ámbito de información del software así como la función, el rendimiento y las interfaces requeridas. Estos requisitos se deben documentar y revisar de tal manera que los entiendan tanto los usuarios como el equipo de desarrollo del software.. En esta fase se desarrollará el documento de requisitos del software que consistirá en una especificación precisa y completa de lo que debe hacer el sistema

2ª Fase Diseño: Consiste en descomponer y organizar el sistema en elementos componentes que puedan ser desarrollados por separado. El resultado del diseño es la colección de especificaciones de cada elemento componente. En esta fase se desarrollará el Documento del diseño del Software que será una descripción del estructura global del sistema.

3ª Fase Codificación: En esta fase se traduce el diseño a un lenguaje legible para el computador. También se harán las pruebas o ensayos necesarios para garantizar que dicho código funciona correctamente. La documentación de esta fase será el Código fuente

4ª Fase Integración: Consiste en probar el sistema completo para garantizar el funcionamiento correcto del conjunto antes de ser puesto en explotación. Aquí tendremos el Sistema Software ejecutable

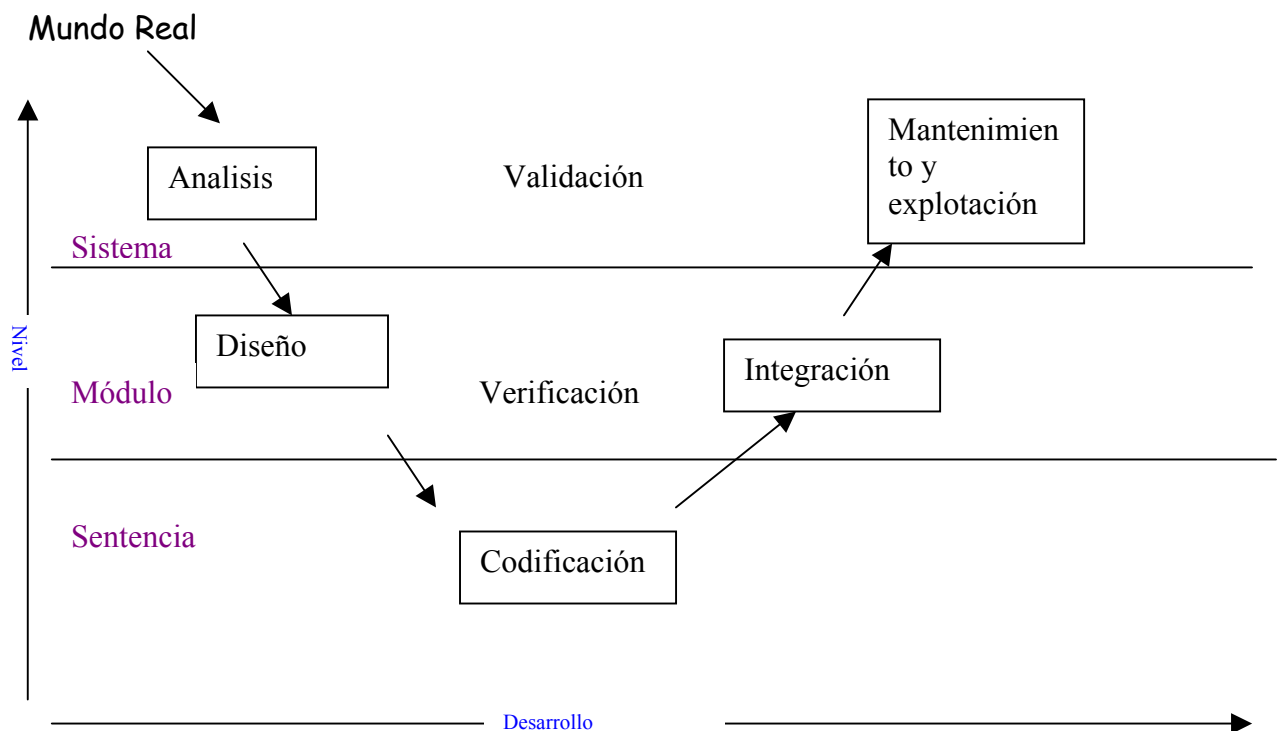
5ª Fase Mantenimiento: Puede ocurrir que durante la explotación del sistema sea necesario realizar cambios para corregir errores que no han sido detectados en las fases anteriores o bien para introducir mejoras. Tendremos que hacer un Documento de cambios ante cualquier modificación.

En todas estas fases la verificación y validación se han de tener en cuenta. La verificación consiste en comprobar que el software que se está desarrollando cumple los requisitos y la validación lo que hace es comprobar que las funciones del software son las que el usuario desea.

El modelo en cascada trata de aislar cada fase de la siguiente de manera que las fases sucesivas puedan ser desarrolladas por grupos de personas distintas facilitándose así la especialización.

MODELO EN V

Se caracteriza por su visión jerarquizada



Ventajas e inconvenientes del modelo de ciclo de vida

Tiene una ventaja y es que es fácil establecer las pautas de seguimiento y control y varios inconvenientes:

- La secuencialidad del modelo
- No prevé revisiones intermedias por parte del usuario por lo que este no ve nada "tangible" hasta fases muy finales
- Para que este modelo sea eficaz hay que partir de que los requisitos una vez definidos no sean "retocados".

MODELOS BASADOS EN PROTOTIPOS

Estos modelos fueron creados para solventar las diferencias percibidas en los modelos clásicos. Permiten a los desarrolladores construir rápidamente versiones tempranas de los sistemas software que pueden evaluar los usuarios.

No resulta económico generar documentación durante las fases iterativas de la construcción del prototipo. Existen varios modelos derivados del uso de prototipos:

- Prototipos rápidos, también llamados maquetas
- Prototipos evolutivos

1.- Prototipos rápidos

Deben poder construirse con facilidad para evaluarlos en una temprana fase del desarrollo y además han de ser baratos.

Otra de las características es que deben ser desarrollados en poco tiempo. También se denominan de usar y tirar. Tienen como finalidad la de adquirir experiencia sin pretender emplearlos como productos.

1.1 Objetivos del prototipo:

- Reducir el riesgo de construir un producto que se aleje de las necesidades del usuario
- Reducir el coste de desarrollo al disminuir las correcciones en etapas avanzadas del mismo.
- Aumentar las posibilidades de éxito del producto.

2.- Prototipos evolutivos

Se construye una implementación parcial del sistema que satisface los requisitos conocidos, la cual es empleada por el usuario para comprender mejor la totalidad de requisitos que desea.

Estos modelos se encaminan a conseguir un sistema flexible que se pueda expandir de forma que sea posible realizar rápidamente un nuevo sistema cuando los requisitos cambian. Están especialmente indicados en situaciones en la que se trabaja con lenguajes de 4ª generación (4GL) y cuando el usuario no sabe lo que quiere.

En este modelo se asume que los requisitos desde el principio van a cambiar continuamente, además lo lógico es comenzar con los aspectos que mejor se comprendan puesto que solo se conocerán unos pocos requisitos y los restantes se tienen que ir descubriendo en sucesivas evoluciones del prototipo; cada versión que se desarrolle será una nueva versión de todo el sistema. Un ejemplo de prototipado evolutivo es el llamado RAD (desarrollo rápido de aplicaciones) también llamado ciclo de vida RAD.

Ventaja e inconvenientes de los modelos basados en prototipos

1. Ventajas

- Los requisitos de los usuarios son más fáciles de determinar y la implantación del sistema será más sencilla debido a que los usuarios conocen lo que esperan.
- Los sistemas se desarrollan más rápidamente
- Este paradigma facilita la comunicación con los usuarios mejorándose dicha comunicación entre el analista y el usuario (cliente).

2. Inconvenientes

- Puede crear falsas expectativas en el usuario ya que puede ver el prototipo como si fuera el producto final
- Puede darse una fuerte intromisión de los usuarios finales en la integración
- Se producen inconsistencias entre el prototipo y el sistema final
- Para todo tipo de prototipado cabe decir que no es un paradigma apto para proyectos grandes y de larga duración ni para aplicaciones pequeñas (menos de un mes), siendo óptimo en aplicaciones y proyectos cuya duración esté fijada entre 3 y 5 meses.

MODELO DE CICLO DE VIDA EN ESPIRAL

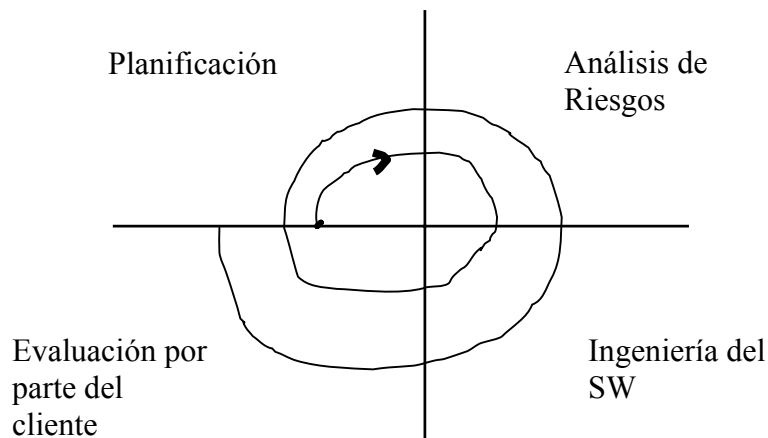
Como alternativa a los modelos de ciclo de vida tradicionales se encuentra este modelo que fue propuesto por Böehm para solventar los principales que presentaban aquellos.

Puede considerarse como un refinamiento del modelo evolutivo general.

Este modelo se caracteriza principalmente porque incorpora un nuevo elemento que es el análisis de riesgos.

Las principales diferencias entre este modelo y los anteriores son:

- En este modelo hay un reconocimiento explícito de las diferentes alternativas para alcanzar los objetivos del proyecto.
- Se centra en la identificación de los riesgos asociados a cada alternativa y en la manera de resolver dichos riesgos.
- Los proyectos se dividen en ciclos (ciclos de la espiral) avanzándose en el desarrollo mediante consensos al final de cada ciclo.
- Se adapta a cualquier tipo de actividad



Cada espiral completa es un ciclo donde se elabora un producto; la unión de todos será el sistema final.

Cada ciclo de la espiral implica una progresión en el desarrollo del producto software, que aplica la misma secuencia de pasos para cada parte del proyecto.

Esta secuencia de pasos se compone de 4 actividades:

- Planificación: Se identifican los objetivos, las alternativas y las restricciones de la fase
- Análisis de Riesgos: Aquí se llevan a cabo el análisis de las alternativas e identificación y resolución de los riesgos.
- Ingeniería del Software: Es donde se desarrolla el producto que corresponde a ese ciclo
- Evaluación por parte del cliente: Valora los resultados que se obtienen de la ingeniería y se planifica el siguiente ciclo.

Hay que destacar dos conceptos:

- La dimensión radial: Indica los costes de desarrollo acumulados
- La dimensión angular: Indica el proceso hecho en cualquier fase

Ventajas e inconvenientes

- Ventajas
 - Amplio rango de opciones a las que puede ajustarse
 - Su orientación al riesgo evita, si no elimina, muchas de las posibles dificultades
 - Concentra su atención en opciones que permiten reutilizar software
 - Se centra en la eliminación de errores y alternativas poco atractivas
 - Permite incorporar objetivos de calidad en el desarrollo software
 - Se adapta bien al diseño y POO
- Inconvenientes:
 - Depende de manera excesiva de la experiencia que se tenga en identificar y evaluar riesgos.
 - Dificultad para adaptar su aplicabilidad al software contratado debido a la poca flexibilidad y libertad de este.

MANTENIMIENTO DEL SOFTWARE

Evolución de las aplicaciones

Según Pressman:

- Mantenimiento Correctivo
- Mantenimiento adaptativo
- Mantenimiento perfectivo

Fase de Mantenimiento: Consiste en repetir o realizar parte de las actividades de las fases anteriores para introducir cambios en una aplicación del software ya entregada al cliente y puesta en explotación.

1. Mantenimiento Correctivo: Tiene como finalidad corregir errores en el producto software que no han sido detectados y eliminados durante el desarrollo inicial del mismo.
2. Mantenimiento Adaptativo: Se produce en aplicaciones cuya explotación dura bastante años, de manera que los elementos básicos HW y SW que constituyen la plataforma o entorno en que se ejecutan evolucionan, modificándose parcialmente la interfaz ofrecida a las aplicaciones que corren sobre ellos.
3. Mantenimiento Perfectivo: Es necesario para obtener versiones mejoradas del producto que permitan mantener o aumentar el éxito del mismo.

Gestión de cambios

Con independencia del objetivo concreto del mantenimiento, las actividades a realizar durante el mismo son básicamente la realización de cambios significativos sobre el software ya realizado. Podemos distinguir 2 enfoques diferentes en cuanto a la gestión de cambios, en función del mayor o menor grado de modificación del producto:

1. Si el cambio a realizar afecta a la mayoría de los componentes de software se puede plantear como un nuevo desarrollo y aplicar un nuevo ciclo de vida utilizando lo ya desarrollado (reutilización).
2. Si el cambio afecta a una parte localizada del producto entonces se puede organizar como una simple modificación de algunos elementos

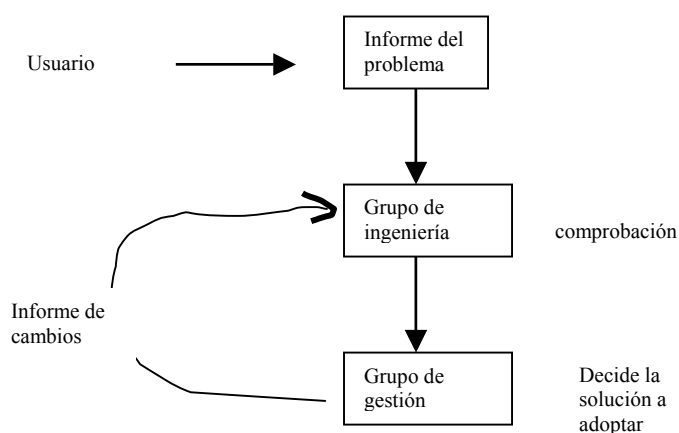
La realización de cambios se va a controlar mediante dos tipos de documentos:

- Informe del problema
- Informe de cambios

El primero hace una descripción sobre el problema que presenta el producto a la hora de su utilización.

El segundo describe la solución dada a un problema y el cambio realizado en el producto software.

Esquemáticamente podríamos verlos como:



Reingeniería

Hay productos software que fueron desarrollados de manera artesanal, es decir, no documentados y de ellos sólo se dispone del código fuente. Para poder mantener este tipo de productos software con escasez de documentación existen dos técnicas:

- La ingeniería inversa: Consiste en tomar el código fuente y a partir de él construir la documentación.
- La reingeniería: Consiste en generar un sistema bien organizado y documentado a partir del sistema inicial deficiente.

GARANTÍA DE CALIDAD DEL SOFTWARE

La calidad de un producto software vienen determinados por el proceso seguido en su desarrollo.

En el modelo de ciclo de vida, las actividades de este tienen como objetivo controlar la calidad del producto son las revisiones y las pruebas.

Factores de calidad

Existe un esquema general de mediciones de la calidad del software propuesto por McCall basado en valoraciones a tres niveles diferentes denominados: *factores*, *criterios* y *métricas*. Los factores de calidad constituyen el nivel superior, y son la valoración propiamente dicha, significativa de la calidad. La valoración no se hace directamente sino en función de ciertos criterios o aspectos de nivel intermedio que influyen sobre los factores de calidad. Las métricas están en el nivel inferior, son mediciones puntuales de determinados atributos o características del producto y son la base para evaluar los criterios.

Entre los factores de calidad encontramos los siguientes:

- Corrección: Es el grado en que un producto software cumple con sus especificaciones. Se puede definir como el porcentaje de requisitos que se cumplen adecuadamente.
- Fiabilidad: Es el grado de ausencia de fallos durante la operación del software; puede estimarse como el número de fallos producidos o tiempo durante el que permanece inutilizable.
- Eficiencia: Relación entre la cantidad de resultados suministrados y los recursos requeridos durante la operación.
- Seguridad: Dificultad para el acceso a los datos u operaciones por parte de personal no autorizado
- Facilidad de uso: Es la inversa del esfuerzo requerido para aprender a usar un producto software y poder utilizarlo adecuadamente.
- Mantenibilidad: Facilidad para corregir el software.
- Flexibilidad: Facilidad para modificar el producto software.
- Facilidad de prueba: Es la inversa del esfuerzo requerido para ensayar un producto software y comprobar su corrección o fiabilidad
- Transportabilidad: Facilidad para adaptar el producto software a una plataforma (Hardware + S.O) diferente a aquella para la que fue desarrollado inicialmente.

- Reusabilidad: Facilidad para emplear parte del producto en otros desarrollos posteriores.
- Interoperatividad: Facilidad del producto Software de trabajar en combinación con otros productos.

Plan de garantía de la calidad

La Calidad del software se consigue mediante una buena organización del desarrollo, lo cual se materializará en un documento llamado plan de garantía de calidad del software el cual contempla los siguientes aspectos:

1. Organización de los equipos de personas y la dirección y seguimiento del desarrollo.
2. El modelo de ciclo de vida a seguir.
3. Documentación requerida.
4. Revisiones y auditorías que se llevaran a cabo durante el desarrollo para garantizar que las actividades y documentos son correctos y aceptables.
5. Organización de las pruebas que se realizarán sobre el producto software a distintos niveles.
6. Organización de las etapas de mantenimiento.

Revisiones

Una revisión consiste en inspeccionar el resultado de una actividad para determinar si es aceptable o no. Se aplicara por lo tanto a la documentación generada en cada fase del desarrollo. Algunas recomendaciones a seguir para formalizar las revisiones son:

- Deben ser realizadas por un grupo de personas, no por una sola
- El grupo que realice la revisión debe ser reducido (3-5 personas)
- No deben ser realizadas por los autores del producto y así poder garantizar la imparcialidad del criterio.
- Se debe revisar el producto, pero no el productor ni el proceso de producción
- Si la revisión tiene que decidir la aceptación o no del producto se debe establecer previamente una lista formal de comprobaciones a realizar
- Debe levantarse acta de la reunión de la revisión. Este documento puede considerarse como el producto de la revisión.

Pruebas

Consiste en hacer funcionar un producto software o una parte de él en condiciones determinadas y comprobar si los resultados son los correctos. Por lo tanto el objetivo de las pruebas será descubrir los errores que pueda contener el software probado.

Las pruebas no permiten garantizar la calidad de un producto.

Gestión de la configuración

Es preciso llevar un control y registro de los cambios con el fin de reducir errores, aumentar la calidad y productividad.

El objetivo de la gestión de la configuración es el de mantener la integridad de los productos que se obtienen a lo largo del desarrollo del sistema de información, garantizando que no se realizan cambios incontrolados y que todos los participantes en el desarrollo del sistema disponen de la versión adecuada de los productos que manejan.

Esta gestión se realiza durante todas las actividades asociadas al desarrollo del sistema.

Desde el punto de vista del usuario y del desarrollador se consideran como elementos componentes de la configuración todos los que intervienen en el desarrollo. Estos elementos serán por tanto:

- Documentos del desarrollo
- Código fuente de los modelos
- Programas, datos y resultados de las pruebas
- Manuales del usuario
- Documentos de mantenimiento
- Prototipos intermedios

¿Qué es configurar software?

La manera en que diversos elementos se combinan para construir un producto software bien organizado.

Para mantener bajo control la configuración del sw nos vamos a apoyar en dos técnicas que son:

- Control de versiones: Consiste en almacenar de forma organizada las sucesivas versiones de cada elemento de la configuración.
- Control de cambios: Consiste en garantizar que las diferentes configuraciones del software se componen de elementos compatibles entre si.

El control de cambios se realiza empleándole concepto de *línea base* que es una configuración particular del sistema. Cada línea base se constituye a partir de otra mediante la inclusión de ciertos cambios.

Las líneas base constituyen configuraciones estables que no pueden ser modificadas. La única forma de modificar una línea base es crear otra nueva introduciendo los cambios apropiados.

Normas y estándares

Las recomendaciones de la ingeniería del software se han introducido mediante normas, algunas éstas han sido recogidas por organizaciones internacionales y establecidas como estándares. Algunas de estas organizaciones son:

- IEEE
- DoD: Departamento de defensa americano
- ESA: Agencia europea del espacio
- ISO: Son las siglas del organismo internacional de normalización. El organismo español que lo integra es AENOR. Entre sus normas de Ingeniería del software de mayor nivel se encuentran las ISO-9001 que establecen los criterios que han de cumplir las empresas que desarrollen software para obtener certificaciones de determinados niveles de garantía de calidad para su producción.
- MÉTRICA 3: Ofrece a los organismos oficiales un instrumento útil para la sistematización de las actividades que dan soporte al ciclo de vida del software.

TEMA 2 ESPECIFICACIÓN DEL SOFTWARE

2.1 MODELADO DE SISTEMAS

Cuando hablamos de modelo nos referimos a un modelo completo y preciso del comportamiento u organización del sistema software.

2.1.1 Definición del modelo conceptual

Es todo aquello que nos permite conseguir una abstracción lógica-matemática del mundo real y que facilita la comprensión del problema a resolver.

El modelo de un sistema software debe establecer las restricciones y propiedades del modelo.

Nota: El modelo debe explicar que debe hacer el sistema y no como lo debe hacer.

2.1.2 Técnicas de modelado

Las dificultades que presenta la obtención de un modelo son:

- Los sistemas a modelar son complejos
- El no disponer de experiencia o referencia anterior

Ante estas dificultades algunas de las técnicas a emplear son:

2.1.2.1 Descomposición. Modelo jerarquizado

Ante un problema complejo esta técnica lo que hace es descomponer el problema en otros más sencillos. De esta manera se establecerá el modelo jerárquico. La descomposición se puede realizar de dos formas:

1. Descomposición Horizontal: Trata de descomponer funcionalmente el sistema, siendo posible descomponer el sistema en otros subsistemas más simples.
2. Descomposición Vertical: Trata de descomponer estructuralmente el sistema o el problema.

2.1.2.2 Aproximaciones sucesivas

Es normal que el sistema software que se quiera modelar sustituya a un proceso de trabajo ya existente que se realizaba de manera manual o con un grado de automatización menor del que se pretenda lograr con el nuevo sistema. En este caso se va a crear un modelo de partida basado en la forma de trabajo anterior, pero tendrá que ser depurado mediante aproximaciones sucesivas hasta alcanzar el sistema final.

Para lograr un buen resultado mediante esta técnica además del analista es fundamental contar con la colaboración de alguien que conozca bien el sistema anterior, capaz de incorporar mejoras del nuevo sistema.

2.1.2.3 Empleo de diversas notaciones

En ocasiones el modelo resulta muy complejo o incluso imposible de realizar empleando una única notación para su elaboración por lo que será conveniente emplear notaciones complementarias que simplifiquen el modelo. Una posible notación es el lenguaje natural.

Existen diversas herramientas de modelado para la ayuda al análisis y diseño de sistemas llamadas herramientas CASE.

2.1.2.4 Consideración de distintos puntos de vista

Para concretar la creación de un modelo es necesario adoptar un determinado punto de vista, ya que obviamente el modelo estará influenciado por el punto de vista adoptado, por lo que este deberá ser el más adecuado.

En ocasiones el modelo no quedará completamente definido con un solo punto de vista por lo que lo más adecuado será realizarlo teniendo en cuenta más de uno. Que muestre todos aspectos del dominio.

2.1.2.5 Realizar un análisis del dominio

Por dominio entenderemos el campo de aplicación en el que se encuadra el software a desarrollar.

Dentro de cada campo existe una cierta manera de realizar las cosas y una terminología que debe ser respetada y que tendrá que ser tomada en cuenta. A esto es a lo que llamamos realizar un Análisis del Dominio de la aplicación.

Para realizar este análisis, es aconsejable estudiar los siguientes aspectos:

1. Normativa que afecte al sistema
2. Otros sistemas semejantes
3. Estudios recientes en el campo de la aplicación
4. Bibliografía clásica y actualizada
5. ...etc...

Este estudio facilitará la creación de un modelo más universal. Como ventajas de este enfoque se pueden citar las siguientes:

1. Facilitar la comunicación entre el analista y el usuario del sistema
2. Creación de elementos realmente significativos del sistema: Cuando se ignora el dominio de una aplicación, la solución que se adopta queda particularizada en exceso.
3. Reutilización posterior del Software desarrollado.

2.2 ANÁLISIS DE LOS REQUISITOS DEL SOFTWARE

La etapa del análisis que se encuadra dentro de la primera fase del ciclo de vida tiene una importancia decisiva en el desarrollo de todas las etapas posteriores.

El análisis de requisitos consiste en caracterizar el problema a resolver. Esta tarea es bastante compleja ya que el primer problema que se le presenta al analista es conseguir un interlocutor válido al que denominaremos cliente.. Este puede ser una persona o bien estar constituido por varias personas expertas en todo o sólo en uno de los campos que se pretenden automatizar con el sistema.

El cliente junto con el analista serán los encargados de elaborar las especificaciones del proyecto software.

2.2.1 Objetivos del análisis

El objetivo global es obtener las especificaciones que debe cumplir el sistema a desarrollar. El medio que se emplea para conseguir dicho objetivo es obtener un modelo válido que recoja todas las necesidades y exigencias del cliente, así como las restricciones que el analista considere debe verificar el sistema. Las especificaciones se obtendrán basándose en el modelo obtenido. Para lograr una especificación correcta el modelo global del sistema deberá tener las siguientes propiedades:

1. Completo y sin omisiones
2. Conciso y sin trivialidades
3. Sin ambigüedades: Existe a veces cierta tendencia a considerar que el análisis de requisitos es un mero trámite. Las consecuencias de esta actitud son:
 - Dificultades en el diseño
 - Retrasos y errores en la implementación
 - Imposibilidad de verificar si el sistema cumple las especificaciones.
4. Sin detalles de diseño o implementación
5. Fácilmente entendible por el cliente
6. Separando los requisitos funcionales y no funcionales: Los requisitos funcionales son los destinados a establecer el modelo de funcionamiento del sistema y serán el fruto fundamental de las discusiones entre analista y cliente. Los requisitos no funcionales están destinados a encuadrar el sistema dentro de un entorno de trabajo. Estos no tienen interés para el cliente y por tanto deben permanecer claramente separados en el modelo del sistema:
 - Capacidades mínimas y máximas
 - Interfases con otros sistemas
 - Aspectos de seguridad
 - Recursos que se necesitan
 - ... etc ...
7. Dividir y jerarquizar el modelo
8. Fijar los criterios de validación de sistema: Un buen método a utilizar para fijar estos criterios es el realizar con carácter preliminar el manual del usuario.

2.2.2 Tareas del análisis

Para realizar un buen análisis de requisitos, conviene efectuar una serie de tareas que son:

1. Estudio del sistema en su contexto: Consiste en conocer el medio en el que se va a desarrollar el sistema
2. Identificación de necesidades: Será el analista el que se encargue de concretar las necesidades que se van a poder cubrir con los medios disponibles y dentro del presupuesto y plazos asignados.
3. Análisis de alternativas. Estudio de viabilidad: Consiste en buscar la alternativa que cubra las necesidades reales detectadas en la tarea anterior y que tenga en cuenta su viabilidad tanto técnica como económica.
4. Establecimiento del modelo del sistema: Para la elaboración del modelo se deberán utilizar todos los medios disponibles, desde procesadores de texto hasta herramientas CASE pasando por las más diversas herramientas gráficas. Todo ello debe contribuir a simplificar la elaboración del modelo y a facilitar la comunicación entre analista, cliente y diseñador.
5. Elaboración del documento de elaboración de requisitos: El resultado final del análisis va a ser este documento, el cual debe recoger todas las conclusiones obtenidas en la fase de análisis. Este documento es el que utilizará el diseñador como punto de partida de su trabajo.
6. Revisión continuada del análisis: Si se prescinde de esta tarea se corre el riesgo de realizar un sistema del que no se tenga ninguna especificación concreta y en consecuencia tampoco ningún medio de validar si es o no correcto el sistema finalmente desarrollado.

2.3 NOTACIONES PARA LA ESPECIFICACIÓN

La especificación es una descripción del modelo que se pretende desarrollar.

La notación o notaciones empleadas deberán ser fáciles de entender por el cliente y por todos aquellos que participen en el análisis y desarrollo del sistema.

El utilizar una notación u otra dependerá de la complejidad o tipo de sistema a desarrollar.

2.3.1 Lenguaje natural

Esta notación es suficiente para aquellos sistemas cuya complejidad es baja. Si la complejidad de un sistema es alta o mediana resulta imposible el empleo de esta notación. Los principales errores que presenta son:

- Imprecisiones
- Incorrecciones
- Repeticiones

El lenguaje natural será siempre la notación a emplear siempre que sea necesario aclarar cualquier aspecto concreto del sistema que no sean capaces de reflejar el resto de las notaciones.

Siempre que se utilice el lenguaje natural es importante organizar y estructurar los requisitos. La mejor manera de lograrlo, es concebir cada uno de los requisitos como una cláusula de contrato entre el analista y el cliente.

El *lenguaje natural estructurado* es una notación más formal que el lenguaje natural. Se trata de establecer ciertas reglas para la construcción de las frases en las que se especifica una acción de tipo secuencia, condición o iteración.

2.3.2 Descripciones funcionales. Pseudocódigo

Los requisitos funcionales se consideran dentro de la especificación de requisitos como una parte esencial. Entenderemos por pseudocódigo como una notación basada en un lenguaje de programación estructurado. El pseudocódigo no tiene una sintaxis estricta y se pueden introducir descripciones en lenguaje natural. Las estructuras básicas que se emplean en el pseudocódigo son:

1. Selección:

SI <condición> ENTONCES
 <pseudocódigo>
SI-NO
 <pseudocódigo>
FIN-SI

2. Selección por casos:

CASO <especificación del elemento selector>
SI-ES <descripción del caso 1> HACER <pseudocódigo>
SI-ES <descripción del caso 2> HACER <pseudocódigo>
.....
SI-ES <descripción del caso n> HACER <pseudocódigo>
OTROS <pseudocódigo>
FIN-CASO

3. Iteración con precondición:

MIENTRAS <pseudocódigo de la descripción> HACER
<pseudocódigo>
FIN-MIENTRAS

4. Repetir con postcondición:

REPETIR
<pseudocódigo>
HASTA <pseudocódigo de la condición>

5. Número de iteraciones conocidas:

PARA-CADA <Especificación del elemento índice> HACER
<pseudocódigo>
FIN-PARA

2.3.3 Descripción de datos

Constituye otro aspecto importante de una especificación software. Consiste en detallar la estructura interna de los datos que maneja el sistema. Solamente se describirán aquellos datos que resulten relevantes para entender que hace el sistema.

La notación adoptada en la metodología de análisis estructurado es lo que se denomina como diccionario de datos.

Existen diversos formatos posibles de diccionario de datos, pero todos aconsejan que al menos se pueda describir la siguiente información para cada dato

1. Nombre(s): identificador del dato.
2. Utilidad: Se indicarán todos los procesos, descripciones funcionales, almacenes de datos, etc en los que se utiliza el dato.
3. Estructura: Se indicarán los elementos de los que está constituido el dato utilizando la siguiente notación:
 - A+B Secuencia o concatenación de los elementos A y B
 - [A|B] Selección entre los distintos elementos A o bien B
 - {A}^N Repetición N veces del elemento A
 - (A) Opcionalmente se podrá incluir el elemento A
 - /Descripción/ Descripción en lenguaje natural como comentario.
 - = Separador entre el elemento de un nombre y su descripción

Ejemplo

Nombre: Datos Tarjeta

Utilidad: Proceso: Comprobar Tarjeta como entrada

Proceso: Comprobar Tarjeta como salida

Almacén de datos: Información de acceso

Entidad externa: Lector de tarjetas

Estructura: Nombre +

1 Apellido +

Apellido2 +

Nivel Acceso +

Clave +

(código empresa)

Nombre= /Ristra con 10 caracteres máximo/

.....

Nivel de acceso= [0|1|2]

Código empresa =[101|102|.....|199]

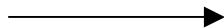
2.3.4 DIAGRAMA DE FLUJO DE DATOS (DFD)

Esta notación está asociada a la metodología de análisis estructurado. Desde el punto de vista de dicho análisis un sistema software se modela mediante el flujo de datos que entra al sistema, las transformaciones que se realizan con los datos de entrada y el flujo que se produce como resultado de la transformación. Los componentes del DFD son los siguientes:

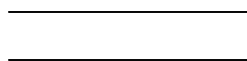
- Proceso o burbuja: Transforma la información de entrada después de pasar por el en información de salida.



- Flujos de datos: Van a acompañados de texto. Describen el movimiento de la información de una parte del sistema a otra.



- Almacenes: Se emplean para modelar una colección de datos que están en reposo dentro del sistema, como por ejemplo los archivos.



- Entidades externas: Son aquellas que producen consumen información del sistema. Residen fuera del sistema a ser modelado. Hay 3 puntos a tener en cuenta:



- i. Son externas al sistema que queremos modelar
- ii. Ni el analista ni el programador pueden cambiar los contenidos ni la forma de trabajo de una entidad externa
- iii. Las relaciones que existen entre las entidades no se muestran en el DFD.

Los DFD's se usan de forma jerarquizada por niveles. El primer nivel lo constituye el diagrama de contexto también llamado de nivel 0. Este representa el software completo como una sola burbuja. Los DFD's sirven principalmente para establecer un módulo conceptual del sistema que facilite la estructuración de su especificación. Este modelo es fundamentalmente estático y refleja los procesos necesarios para su desarrollo y las interrelaciones que existen entre ellos.

2.3.5 DIAGRAMA DE MODELO DE DATOS. MODELO ENTIDAD RELACION (DE/R)

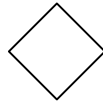
Esta es la notación principal para la modelización de los datos. Permite definir todos los datos que manejará el sistema junto con las relaciones que se desean existan entre ellos.

Este modelo es el punto de partida necesario para comenzar cualquier diseño. Los componentes de que consta son:

- Entidades:



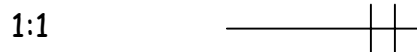
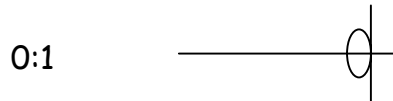
- Relación: Se trata de la conexión entre entidades



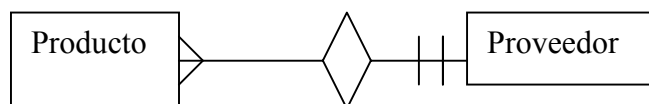
- Flecha: Representa el sentido de la relación



- Cardinalidad: Indica entre que valores mínimos y asimos se mueve la relación entre entidades.



Ejemplo:



1 producto es suministrado por un proveedor

1 proveedor puede suministrar varios productos.

2.3.6 DIAGRAMA DE TRANSICIÓN DE ESTADOS (DTE)

Representa el comportamiento de un sistema que muestra los estados así como los sucesos que hacen que el estado del sistema cambie.

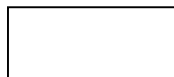
Todos los estados y sucesos configuran la dinámica del sistema que se produce mientras se está ejecutando.

Esta notación se utiliza para describir el comportamiento dinámico del sistema a partir de los estados más importantes. Se complementan con los DFD's. Los componentes son:

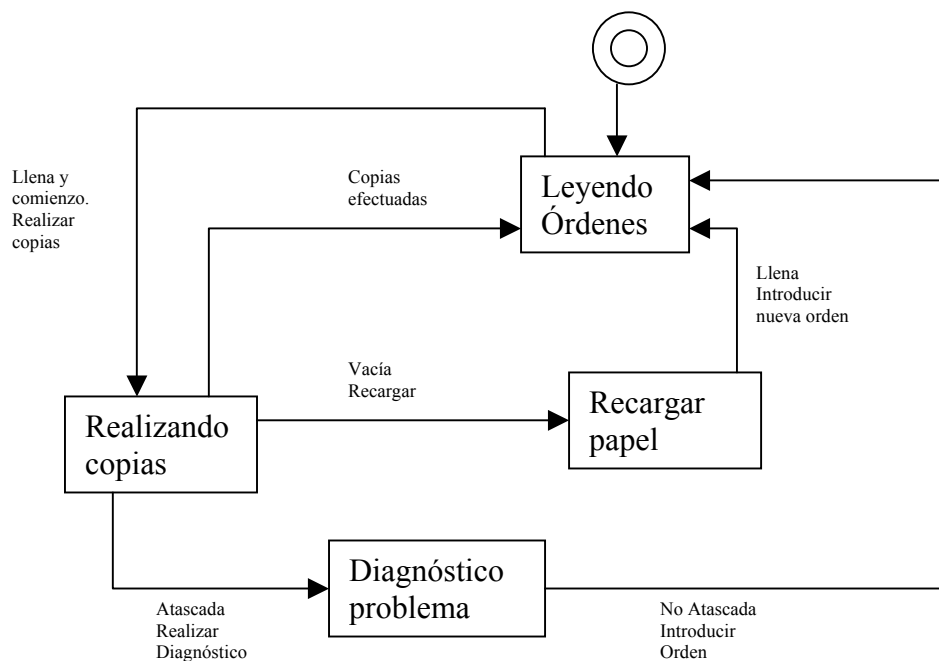
- Estado inicial/final:



- Estados intermedios:



- Evento o suceso: Provocan el cambio de estado. Se representan mediante una flecha dirigida que va del estado antiguo al nuevo.



DTE funcionamiento

2.4 DOCUMENTO ESPECIFICACIÓN REQUISITOS (SRD)

Es el que se va a encargar de recoger todo el fruto del trabajo realizado durante la etapa de análisis del sistema de manera integral en un único documento.

Este documento es imprescindible y fundamental y es el punto de partida de desarrollo de cualquier sistema software. Debe ser redactado de una manera sencilla y tiene que ser fácil de modificar, puesto que será revisado a lo largo del desarrollo del software con frecuencia. Debe facilitar la labor de verificación del cumplimiento de las especificaciones. Según la agencia espacial europea (ESA) consta del siguiente contenido:

1. INTRODUCCIÓN

- 1.1 Objetivo:
- 1.2 *Ámbito:* Aquí se identificará y dará nombre al producto(s) resultantes del proyecto.
- 1.3 Definiciones, siglas y abreviaturas
- 1.4 Referencias
- 1.5 Panorámica del documento

2. DESCRIPCIÓN GENERAL

- 2.1 Relación con otros proyectos
- 2.2 Relación con proyectos anteriores o posteriores
- 2.3 Objetivo y funciones
- 2.4 Consideraciones de entorno
- 2.5 Relaciones con otros sistemas
- 2.6 Restricciones generales
- 2.7 Descripción del modelo

3. REQUISITOS ESPECIFICOS

- 3.1 Requisitos funcionales
- 3.2 Requisitos de Capacidad
- 3.3 Requisitos de interfase
- 3.4 Requisitos de operación
- 3.5 Requisitos de recursos
- 3.6 Requisitos de verificación
- 3.7 Requisitos de pruebas de aceptación
- 3.8 Requisitos de documentación
- 3.9 Requisitos de seguridad
- 3.10 Requisitos de transportabilidad

- 3.11 Requisitos de calidad
 - 3.12 Requisitos de fiabilidad
 - 3.13 Requisitos de Mantenibilidad
 - 3.14 Requisitos de salvaguarda
4. APENDICES

EJERCICIO. Hacer DFD

Una sociedad especializada en la distribución de productos de equipamiento eléctrico dispone de un catálogo de 30 productos.

La venta se realiza por una red de representantes que trabajan para la sociedad. La comercialización de los productos solo abarca ciertas áreas geográficas del país.

Los representantes están repartidos por sectores geográficos. Un cierto número de ellos están especializados en uno o varios ramos profesionales. En la actualidad se manejan 20 ramos

Si tras la entrevista entre cliente y representante se ha cerrado una venta el cliente firma una nota de pedido. El pedido se transmite a la sociedad por el representante.

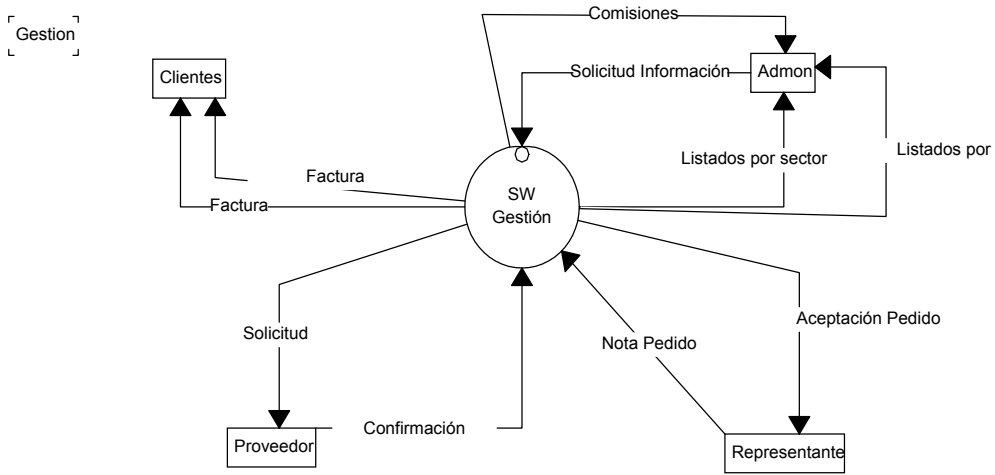
En todos los casos cada venta es firmada por el representante responsable de la cuenta de dicho cliente. En casos particulares una venta puede involucrar a 2 representantes: el responsable de la venta y otro representante. En este caso la comisión se repartirá en partes iguales entre dichos representantes.

Cada mes los representantes perciben el montante de las comisiones por las ventas hechas y aceptadas por la sociedad. Por cada venta aceptada por la sociedad se confecciona una factura por los artículos disponibles en existencias. En caso de artículos sin existencias se elaborará una factura complementaria cuando se reciba dicho material.

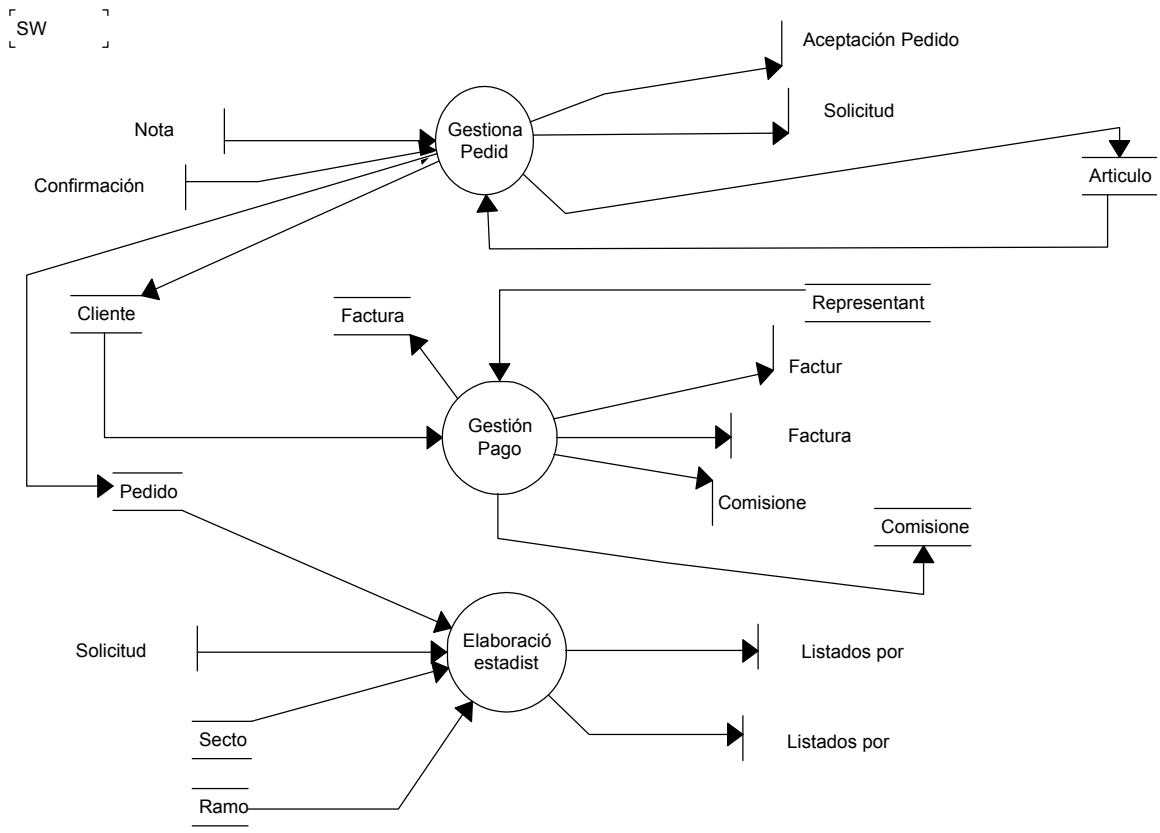
Los pedidos aceptados no son objeto de ningún tratamiento.

Cada mes se confeccionan unos listados estadísticos a fin de seguir la evolución de las ventas por ramos y sector

DFD Nivel 0



DFD Nivel 1



TEMA3 FUNDAMENTOS DEL DISEÑO DEL SW

3.1 INTRODUCCIÓN

En este tema se hará hincapié en especificar o determinar *cómo* se debe resolver el proyecto mientras que en los anteriores se trataba se *qué* debía hacer el sistema.

El diseño de un sistema sw es la descripción del sistema a desarrollar. Se trata pues de definir y formalizar la estructura del sistema con el suficiente detalle como para permitir su realización física. El punto de partida para abordar esta fase es el documento de especificación de requisitos (SRD).

La realización del diseño de un sistema será un proceso creativo; también será importante la experiencia previa. Siempre que sea posible se intentará reutilizar el mayor nº de módulos o elementos ya desarrollados.. El método más eficaz para adquirir experiencia en el diseño consiste en participar en alguno de ellos y en aprender de otros diseñadores. También se aconseja el estudio de diseños ya realizados y el análisis de las razones por las cuales se ha adoptado una u otra solución.

En los años 60 el objetivo del diseño era conseguir la mayor eficiencia posible. Actualmente el objetivo fundamental es conseguir que el software sea fácil de mantener y que se reutilizable.

Durante la etapa del diseño se tiene que pasar gradualmente del "qué" al "cómo", estableciendo la organización y estructura física del software .

A lo largo del proceso de diseño se realizan un conjunto de actividades que tienen como objetivo sistematizar cada uno de los aspectos o elementos de la estructura del sistema. Las actividades habituales en el diseño son:

1. Diseño arquitectónico: Es la primera actividad a realizar y en ella se debe abordar los aspectos estructurales y *la organización del sistema* y su posible división en subsistemas o módulos. Se tienen que establecer las relaciones entre los subsistemas creados y definir las interfaces entre ellos. Esta actividad proporcionara una visión global dl sistema.

2. Diseño detallado: En esta fase se aborda *la organización de los módulos*. Se trata de determinar cual es la estructura más adecuada para cada uno de los módulos en los que quedó subdividido el sistema global. Cuando se trata de diseñar sistemas no muy complejos el diseño detallado no existe.
3. Diseño procedimental: Aquí se abordan *la organización de las operaciones o servicios* que ofrecerán cada uno de los módulos. En esta etapa se diseñan los algoritmos que se emplean en el desarrollo de los módulos de implementación; en esta actividad se detalla en pseudocódigo o PDL (lenguaje de descripción de programas), sólo los aspectos más relevantes de cada algoritmo.
4. Diseño de datos: Se aborda la organización de la base de datos del sistema. Se puede realizar en paralelo con el diseño detallado y procedimental. El punto de partida para esta actividad es el diccionario de datos y los diagramas de E/R. El diseño de datos es importante ya que permite conseguir el objetivo de que el sistema sea reutilizable y fácil de mantener.
5. Diseño de la interfaz de usuario: En cualquier operación resulta muy importante el esfuerzo de diseño destinado a conseguir un diálogo más ergonómico entre el usuario y el ordenador. Esta actividad es la encargada de la organización de la interfaz de usuario y su importancia ha proporcionado el desarrollo de técnicas y herramientas específicas que facilitan el diseño.

El resultado de todas las actividades de diseño deben dar lugar a una especificación lo más formal posible de la estructura global del sistema. Esto constituye el producto del diseño y se recogerá en el documento de diseño del software.

3.2 CONCEPTOS DE BASE

Los más importantes a tener en cuenta en cualquier diseño son:

- Abstracción: Cuando se diseña un nuevo sistema software es importante identificar los elementos más significativos de los que consta y además abstraer la utilidad específica de cada uno. Este esfuerzo dará como resultado que el elemento diseñado pueda ser sustituido por otros con mejores prestaciones y también permitirá la reutilización. Son dos de los principales objetivos del diseño:
 - Conseguir elementos fácilmente mantenibles
 - Reutilización

En el diseño de elementos software se pueden utilizar tres formas de abstracción:

- i. Abstracción Funcional: Se utiliza para crear expresiones parametrizadas mediante el empleo de funciones o procedimientos. Para diseñar una abstracción funcional es necesario fijar los parámetros que se le debe pasar así como el resultado que devolverá.
 - ii. Tipos Abstractos: Sirven para crear nuevos tipos de datos que se necesitarán para abordar el diseño del sistema. Además, junto al nuevo dato, se deben diseñar todas las operaciones asociadas al tipo.
 - iii. Máquina abstracta: Pretende establecer un nivel de abstracción superior a los dos anteriores y en él se define de manera formal el comportamiento de una máquina.
- Modularidad: Los proyectos normalmente requieren el trabajo simultáneo y coordinado de varias personas. Una forma clásica de conseguir esto es mediante el empleo de un diseño modular. Uno de los primeros pasos que se deben dar al abordar un diseño es dividir el sistema en sus correspondientes módulos. Esta división permite encargar a personas diferentes el desarrollo de cada uno de los módulos de manera que puedan hacer el trabajo simultáneamente.

Las ventajas de usar un diseño modular son:

- La Claridad: Es más fácil entender y manejar cada una de las partes de un sistema que tratar de entenderlo como un todo compacto.
 - Reducción de costes: Resulta más barato desarrollar, depurar, documentar probar y mantener un sistema modular que otro que no lo sea.
 - Reutilización: Si los módulos se diseñan teniendo en cuenta otras aplicaciones resultará inmediata su reutilización.
- Refinamiento: En un diseño siempre se parte inicialmente de una idea poco concreta que se irá refinando en sucesivas aproximaciones hasta perfilar el más mínimo detalle. Las especificaciones recogidas en el documento SRD siempre finalizan siendo expresadas en un lenguaje de programación de alto nivel. La forma natural de acercar el lenguaje natural y el de programación es utilizando el refinamiento.

El objetivo global de un nuevo sistema software expresado en su especificación se debe refinar en sucesivos pasos hasta que todo quede expresado en el lenguaje de programación.

- Estructuras de datos: La organización de la información es una parte importante del diseño de un sistema software. De esto son buenas pruebas las metodologías de diseño orientadas a los datos (Warnier, Jackson). En estas metodologías, las estructuras de datos son el punto de partida del que se arranca para realizar el diseño. Las estructuras fundamentales dentro del diseño son:
- Registros
 - Conjuntos
 - Árboles
 - Listas, colas y pilas
 - Tablas
 - Ficheros, etc

Será labor del diseñador buscar a partir de dichas estructuras la combinación mejor para lograr aquella estructura(s) que den respuesta a las necesidades del sistema.

- Ocultación: Al programador "usuario" de un módulo desarrollado por otro programador del equipo puede quedarle oculta la organización de los datos internos y el detalle de los algoritmos que emplea.

Cuando se diseña una estructura de cada módulo de un sistema se debe hacer de manera que queden ocultos todos los detalles que resulten irrelevantes para su utilización. Lo que se pretende es ocultar al "usuario" todo lo que pueda ser susceptible de cambio en el futuro y además que sea irrelevante para el uso. Por lo que se mostrará en la interfase sólo aquello que no varíe ante cualquier cambio. Las ventajas de aplicar este concepto son:

- Depuración:
- Mantenimiento

- Genericidad: Consiste en diseñar un elemento genérico con aquellas características comunes a todos los elementos agrupados. Este concepto es de gran utilidad en el diseño y da lugar a soluciones simples y fáciles de mantener; pero la implementación de los elementos genéricos obtenidos como resultado del diseño pueden resultar bastante complejos.
- Herencia: Cuando hay elementos con características comunes se establece una jerarquía entre dichos elementos del sistema. Partiendo de un elemento padre que posee una estructura y unas operaciones básicas, desarrollamos unos elementos hijos que heredan del padre su estructura y algunas operaciones básicas. Para ampliarlos o simplemente adaptarlos a sus necesidades.

El mecanismo de herencia permite reutilizar gran cantidad de software ya desarrollado. Los elementos hijos tendrán sus propias estructuras y operaciones.

Este concepto está muy ligado a las metodologías de análisis y diseño orientado a objetos.

Existen dos tipos de Herencia:

- Simple: Cuando un hijo hereda de un único padre
- Múltiple: Cuando hereda de mas de un padre.

3.2.8 Polimorfismo

El concepto de polimorfismo engloba distintas posibilidades utilizadas habitualmente para conseguir que un mismo elemento software adquiriera varias formas simultáneamente:

- Anulación: Tiene que ver con el concepto de herencia. Cuando un hijo HEREDA una estructura o una operación del padre anula la más general.
- Diferido: En este caso se plantea la necesidad de operación para el elemento padre pero su concreción se deja diferida para cada uno de los elementos "hijos" concrete su forma específica.
- Sobrecarga: En este caso quienes adquieren múltiples formas son los operadores, funciones o procedimientos. Se puede usar con operadores (ejemplo del '+' para sumar reales o concatenar) y con funciones o procedimientos.

Todas estas posibilidades del polimorfismo redundan en una mayor facilidad para realizar software reutilizable y mantenible.

Al igual que la herencia el concepto de polimorfismo está ligado a las metodologías orientadas a objetos.

3.2.9 Concurrencia

En los sistemas de tiempo real existe la necesidad de aprovechar toda la capacidad de proceso del ordenador para poder atender a todos los eventos que se producen y en el mismo instante que lo hacen.

A la hora de diseñar un sistema con restricciones de tiempo se ha de tener en cuenta los siguientes conceptos:

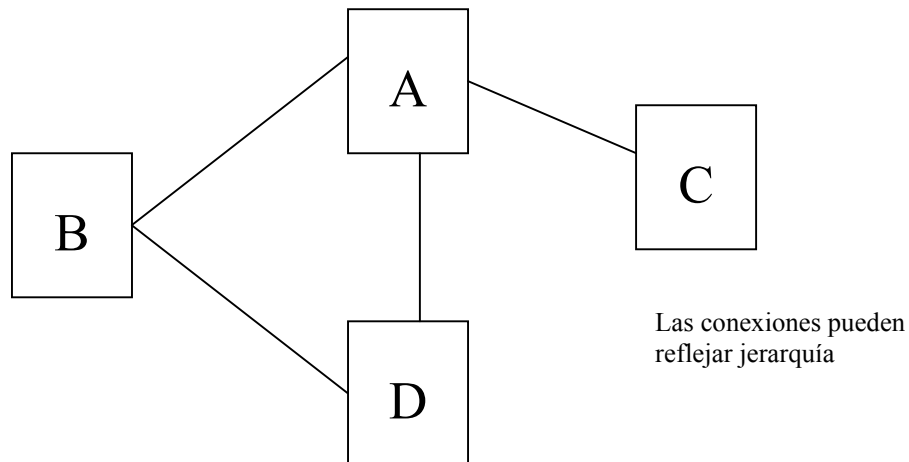
- Tareas concurrentes: Determinar que tareas se deben ejecutar en paralelo para cumplir con las restricciones impuestas. Se deberá prestar especial atención a aquellas tareas con tiempo de respuesta más críticos y aquellas otras que se ejecutaran con más frecuencia.
- Sincronización entre tareas: Determinar los puntos de sincronización entre las distintas tareas. Normalmente las tareas nunca funcionan cada una por su cuenta. Cuando una tarea T1 se encarga de obtener un resultado que debe utilizar otra T2 ambas se deben sincronizar. Para esta sincronización se pueden usar p.ej semáforos.
- Comunicación entre tareas: Determinar si la cooperación se basa en el empleo de datos compartidos o mediante el paso de mensajes entre tareas. En cualquier caso el diseñador debe concretar la estructura de los datos compartidos o de los mensajes. También se debe concretar que tareas son productoras y cuales consumidoras de datos. En el caso de usar datos compartidos se tienen que evitar que los datos puedan ser modificados en el momento de la consulta, por lo que en este caso será necesario usar, semáforos, monitores, etc para garantizar la exclusión mutua entre las distintas tareas.
- Interbloqueos (DeadLock): Se produce cuando una o varias tareas permanecen esperando por tiempo indefinido la finalización de la otra.

3.3 NOTACIONES PARA EL DISEÑO

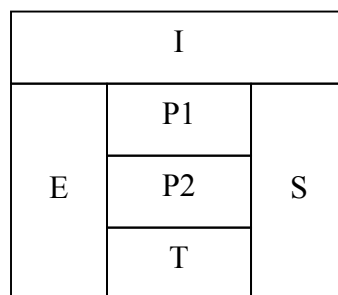
El objetivo fundamental de cualquier notación es resultar precisa, clara y sencilla de interpretar en el aspecto concreto que se describe. Se trata sobre todo de evitar ambigüedades e interpretaciones erróneas del diseño. En este sentido lo más adecuado sería emplear notaciones formales de tipo cuasi matemático. Sin embargo, sólo una parte muy pequeña del resultado de un diseño se suele describir utilizando únicamente notaciones formales. Según el aspecto del sistema a describir clasificaremos las notaciones en los siguientes grupos:

3.3.1 Notaciones estructurales

Sirven para cubrir un primer nivel del diseño arquitectónico y con ellos se trata de desglosar y estructurar el sistema en sus partes fundamentales. Una notación habitual para desglosar el sistema en sus partes son los *diagramas de bloques*



Otra notación informal para estructurar un sistema es el de las “CAJAS ADOSADAS” para delimitar los bloques. La conexión entre los bloques se pone de manifiesto cuando entre dos cajas existe una frontera común



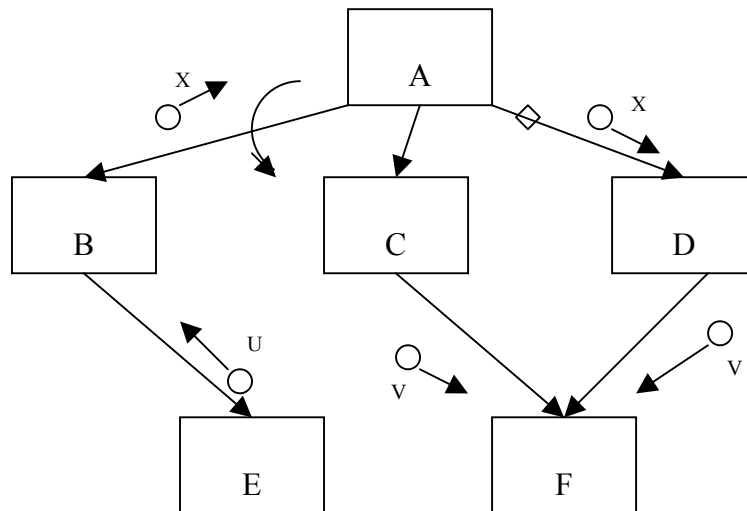
3.3.1.1 Diagramas de estructura

Fueron propuestos para describir la estructura de los sistemas software como una jerarquía de subprogramas o módulos. Los símbolos empleados son:

- Rectángulos: Módulo o subprograma cuyo nombre se indica en el interior
- Líneas: Une a dos rectángulos e indica que el módulo superior llama al inferior. Puede acabar en una flecha.

- Rombo: Se sitúa sobre una línea e indica que esa llama o utilización es opcional.
- Arco: Se sitúa sobre una línea e indica que esa llamada se efectúa de manera repetitiva.
- Círculo con flecha: Se sitúa en paralelo a la línea y representa el envío de datos de un módulo a otro. Si el círculo está relleno son datos de control.

Cuando el diseño realizado es correcto obtenemos un diagrama en forma de árbol invertido mostrando la jerarquización de los módulos. El diagrama de estructura refleja una organización estática de los módulos.



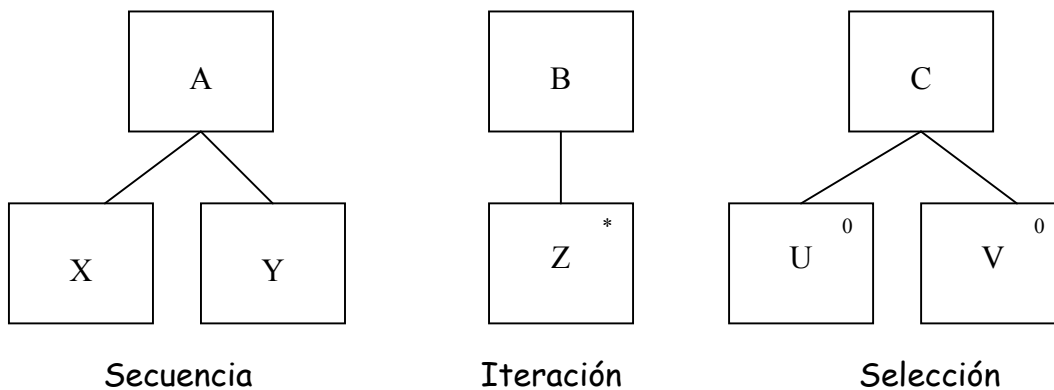
3.3.1.3. Diagramas de Jackson

Fue diseñada por Jackson para diseñar sistemas software a partir de las estructuras de sus datos de entrada y salida.

El proceso de diseño es bastante sistemático y se lleva a cabo en tres pasos:

1. Especificación de la estructuras de los datos de entrada y salida
2. Obtener una estructura del programa capaz de transformar las estructura de los datos de entrada en las de salida. Este paso implica una conversión de las estructuras de datos en estructuras del programa que las manejan, teniendo en cuenta las equivalencias clásicas entre ambas:
 - TUPLA-SECUENCIA: Colección de elementos fijos diferentes combinados en un orden fijo.
 - UNION-SELECCIÓN: Selección de un elemento entre varios posibles de tipos diferentes
 - FORMACIÓN-ITERACION: Colección de elementos del mismo tipo
3. Expansión de la estructura del programa para lograr el diseño detallado del sistema. Para realizar este paso normalmente se utiliza pseudocódigo.

La metodología de Jackson esta englobada dentro de las de *diseño dirigido por los datos*, que has sido utilizado fundamentalmente para diseñar sistemas relativamente pequeños de procesamiento de datos.



3.3.2 Notaciones estáticas

Estas notaciones sirven para describir características estáticas del sistema tales como la organización de la información, sin tener en cuenta su evolución durante el funcionamiento del sistema.

Por tanto como resultado del diseño se tendrá una organización de la información con un nivel de detalle mucho mayor que en el SRD. En cualquier caso las notaciones que se pueden emplear para describir el resultado de este trabajo son las mismas que se emplean para realizar la especificación.

3.3.2.1 Diccionario de datos

Con esta notación se detalla la estructura interna de los datos que maneja el sistema.

Para el diseño se partirá del diccionario de datos incluido en el documento SRD y mediante los refinamientos necesarios se ampliará y completará hasta alcanzar el nivel de detalle necesario para iniciar la codificación.

3.3.2.2 Diagramas entidad-relación

Esta notación permite definir el modelo de datos, las relaciones entre los datos y en general la organización de la información. Para la fase de diseño se tomará como punto de partida el diagrama propuesto en el SRD que se complementará y ampliará

3.3.3 Notaciones dinámicas

Estas notaciones permiten describir el comportamiento del sistema durante su funcionamiento. Al diseñar la dinámica del sistema se detallará su comportamiento externo y se añadirá la descripción de un comportamiento interno capaz de garantizar que se cumplen todos los requisitos especificados en el documento SRD. Así las notaciones que se emplean son las mismas que la utilizadas en la especificación.

3.3.3.1 Diagramas de flujo e datos

Los DFD's serán mucho más exhaustivos que los de la especificación, debido a la necesidad de describir como se hacen internamente las cosas y no sólo que cosas debe hacer el sistema.

3.3.3.2 Diagramas de transición de estados

En el diseño del sistema pueden aparecer nuevos diagramas de estado que reflejen las transiciones entre estados internos. Sin embargo, es preferible no modificar o ampliar los diagramas recogidos en el documento SRD encargados de reflejar el funcionamiento externo del sistema.

3.3.3.3 Lenguaje de descripción de programas (PDL)

Se utiliza tanto para realizar la especificación funcional del sistema como para elaborar el diseño. La diferencia entre ambas situaciones la marca el nivel de detalle al que se desciende en la descripción funcional.

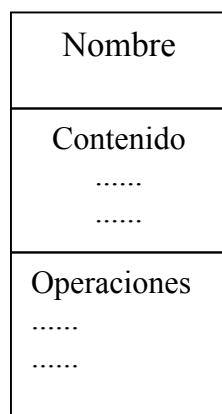
Cuando se quiere descender al nivel de detalle que se requiere en la fase de diseño, la notación PDL se amplía con ciertas estructuras de algún lenguaje de alto nivel. Uno de los más apropiados es ADA.

3.3.4 Notaciones híbridas

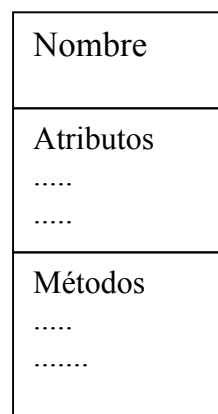
Tratan de cubrir simultáneamente aspectos estructurales estáticos y dinámicos.

3.3.4.1 Diagramas de abstracciones

Fue propuesta originalmente por Liskov para describir la estructura de un sistema Software compuesto por elementos abstractos. En la propuesta original se contemplan dos tipos de abstracciones :las funciones y los TDA's. A ellos se han añadido con símbolo propio los datos encapsulados.



Abstracción



Objeto

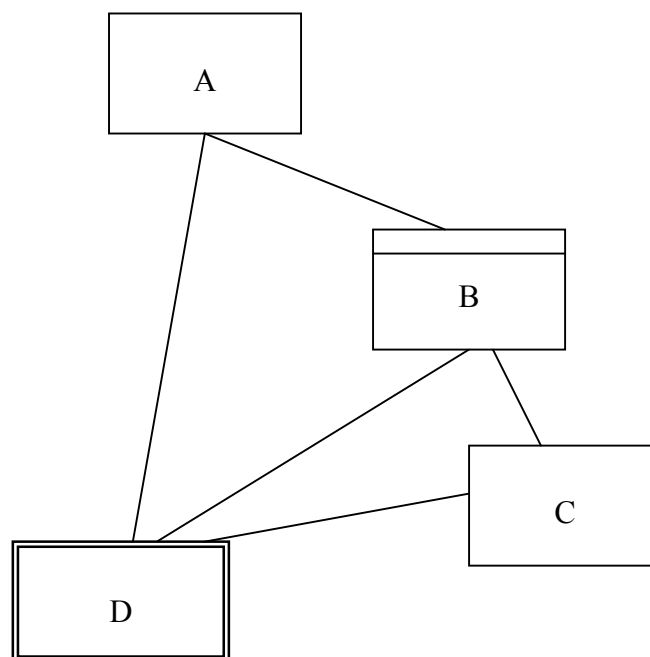
En una abstracción se distinguen tres partes:

- Nombre: Es el identificador de la abstracción.
- Contenido: Es el elemento estático de la abstracción y en él se define la organización de los datos que constituyen la abstracción
- Operaciones: Es el elemento dinámico de la abstracción y en él se agrupan todas las operaciones definidas para manejar el CONTENIDO.

Inicialmente la única forma de abstracción disponible en los lenguajes era la definición de subprogramas (funciones o procedimientos). Un subprograma constituye una operación abstracta que denominaremos *abstracción funcional*. Esta forma de abstracción no tiene contenido y está constituida por una única operación.

En un diseño bien organizado se puede agrupar en una misma entidad la estructura del tipo de datos con las correspondientes operaciones necesarias para su manejo. Esta forma de abstracción se denomina *tipo abstracto de datos* y tiene CONTENIDO y OPERACIONES.

Cuando se necesita una variable de un determinado tipo abstracto, su declaración se puede encapsular dentro de la misma abstracción. Así, todas las operaciones de la abstracción se referirán siempre a esa variable si se necesita de indicarlo de forma explícita. Esta forma de abstracción se denomina *Dato encapsulado*, tiene CONTENIDO Y OPERACIONES, al igual que un TDA, pero no permite declarar otras variables de su mismo tipo.



En este caso los módulos serían abstracciones en sus distintas formas posibles, con relación jerárquica e implica que la abstracción superior utiliza a la inferior. Así la abstracción funcional A utiliza el dato encapsulado D y el tipo abstracto B. Al dato encapsulado D también lo utilizan B y la abstracción funcional C que a su vez es utilizada por B.

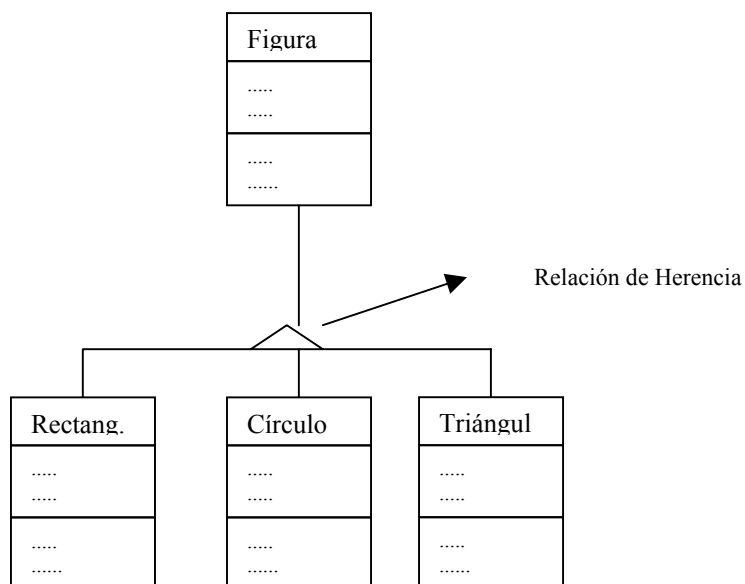
3.3.4.2 Diagramas de objetos

Las abstracciones se pueden considerar una propuesta de los expertos en programación, mientras que los objetos son una propuesta de los expertos en inteligencia artificial.

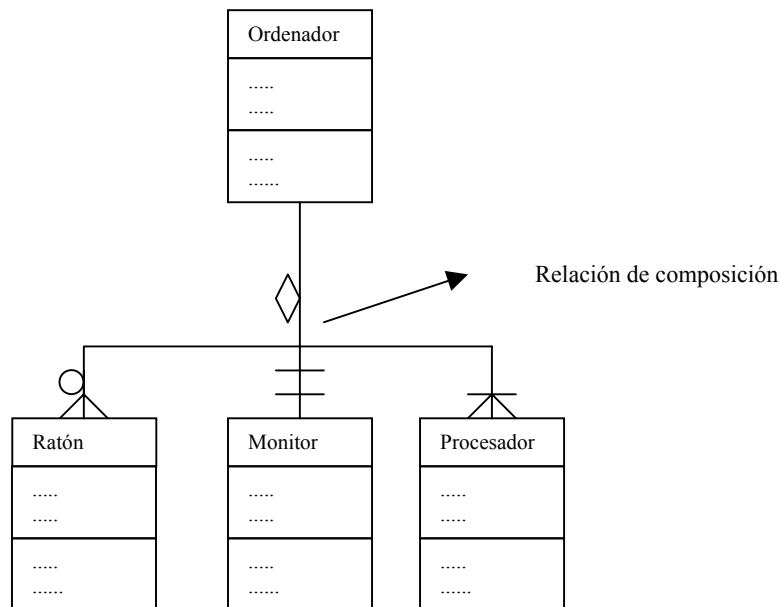
La estructura de un objeto es exactamente igual a la de una abstracción.

Las diferencias fundamentales entre ambos conceptos son las siguientes:

1. No existe nada equivalente a los datos encapsulados ni a las abstracciones funcionales cuando se utilizan objetos en forma estricta.
 2. Solo entre objetos se contempla una relación de herencia. Se pueden establecer entre los objetos dos tipos de relaciones especiales:
- Clasificación, especialización o HERENCIA: Esta relación permite diseñar un sistema aplicando el concepto de herencia.



- **Composición:** La relación de composición permite describir un objeto mediante los elementos que lo forman. En esta relación sólo hay que indicar la Cardinalidad en un sentido.



3.4 DOCUMENTOS DE DISEÑO

El resultado principal de la labor realizada en la etapa de diseño se recoge en un documento que se utilizará como elemento de partida para las sucesivas etapas del proyecto y que denominaremos *Documento de Diseño de Software* (SDD).

Las normas de la ESA establecen el empleo de un *Documento de Diseño Arquitectónico* (ADD) para describir el sistema en su conjunto y otro *Documento de diseño detallado* (DDD) para describir por separado cada uno de los componentes del sistema.

3.4.1 Documentos ADD

1. **INTRODUCCIÓN:** Esta sección debe dar una visión general de todo el documento ADD:
 - Objetivo
 - Ámbito
 - Definiciones, siglas y abreviaturas
 - Referencias

2. PANORÁMICA DEL SISTEMA: Debe dar una visión general de los requisitos funcionales (y de otro tipo) del sistema que ha de ser diseñado, haciendo referencia al documento SRD.

3. CONTEXTO DEL SISTEMA: En esta sección se indicará se este sistema posee conexiones con otros y si debe funcionar de una forma integrada con ellos. En cada uno de sus apartados se definirá la correspondiente interfase que se debe utilizar con cada uno de los sistemas.
4. DISEÑO DEL SISTEMA: Se describe el nivel superior del diseño, en que se considera al sistema en su conjunto y se hace una primera estructuración en componentes.
 - 4.1 Metodología de diseño de alto nivel
 - 4.2 Descomposición del sistema

5. DISEÑO DE LOS COMPONENTES: Para cada uno de los componentes descritos en 4.2:
 - 5.n. Identificador del componente (nombre)
 - 5.n.1 Tipo: Se describe la clase de componente
 - 5.n.2 Objetivo: La necesidad de que exista el componente. Para ello se puede hacer referencia a un requisito concreto que se trata de cubrir. Si éste no forma parte del documento SRD se tendrá que detallar en este momento.
 - 5.n.3 Función: ¿Qué hace el componente?
 - 5.n.4 Subordinados
 - 5.n.5 Dependencias: Se enumeran los componentes que usan a este
 - 5.n.6 Interfases: Se describe cómo otros componentes interactúan con este.
 - 5.n.7 Recursos: Se describen los elementos externos al diseño que son usados por este componente : impresoras, particiones,..
 - 5.n.8 Referencias
 - 5.n.9 Proceso: Se describen los algoritmos o reglas que utiliza el componente.
 - 5.n.10 Datos: Se describen los datos internos del componente incluyendo el método de representación, valores iniciales, etc. Se puede realizar mediante un diccionario de datos.

6. VIABILIDAD Y RECURSOS ESTIMADOS: Se analiza la viabilidad de la realización del sistema y se concretan los recursos que se necesitan para llevarlo a cabo.
7. MATRIZ REQUISITOS/COMPONENTES: Se muestra una matriz poniendo en las filas todos los requisitos y en las columnas todos los componentes. Para cada requisito se marcará el componente(s) encargados de que se cumpla.

3.4.2 Documento DDD

Ira creciendo con el desarrollo del proyecto. La diferencia con el ADD es el nivel de detalle, en este existen un mayor número de componentes y para cada uno de ellos se baja incluso hasta el nivel de codificación. Los listados fuente se recogen dentro del documento como un apéndice.

Parte 1 DESCRIPCIÓN GENERAL

1. INTRODUCCIÓN
 - 1.1 Objetivo
 - 1.2 Ámbito
 - 1.3 Definiciones, siglas y abreviaturas
 - 1.4 Referencias
 - 1.5 Panorámica

2. NORMAS, CONVENIOS Y PROCEDIMIENTOS
 - 2.1 Normas de diseño de bajo nivel
 - 2.2 Normas y convenios de documentación
 - 2.3 Convenios de Nombres
 - 2.4 Normas de programación
 - 2.5 Herramientas de desarrollo de software

Parte 2 ESPECIFICACIONES DE DISEÑO DETALLADO

- n. Identificador del componente
 - n.1 Identificador
 - n.2 Tipo
 - n.3 Objetivo
 - n.4 Función
 - n.5 Subordinados
 - n.6 Dependencias
 - n.7 Interfases
 - n.8 Recursos
 - n.9 Referencias
 - n.10 Proceso
 - n.11 Datos

APÉNDICE A. LISTADOS FUENTE

APÉNDICE B. MATRIZ REQUISITOS/COMPONENTES

TEMA 4. TÉCNICAS GENERALES DE DISEÑO DE SOFTWARE

El diseño de software es una actividad que requiere tener experiencia previa, la cual solo se adquiere a través del estudio de las técnicas de diseño.

Basadas en una o varias técnicas de diseño se han llevado a cabo metodologías completas de diseño y herramientas CASE para diseño asistido por ordenador. Todas las técnicas tratan de conseguir dos objetivos:

- Descomposición modular del sistema de manera que vamos a obtener una división del sistema en partes o módulos.
- Decisión sobre los aspectos de implementación o representación de datos, es decir, el diseñador decidirá sobre los algoritmos y/o estructuras de datos que se deberán utilizar para llevar a cabo el sistema.

Las técnicas de diseño están agrupadas en:

- Diseño funcional descendente.
- Diseño orientado a objetos.
- Diseño de datos.

4.1 DESCOMPOSICIÓN MODULAR

Todas las técnicas de diseño están de acuerdo en la necesidad de realizar una descomposición modular del sistema y para conseguirlo es necesario especificar los siguientes aspectos:

- Identificar los módulos.
- Describir cada módulo.
- Describir las relaciones entre dichos módulos.

La diferencia fundamental entre las diferentes técnicas de diseño es lo que se entiende por módulo en cada una de ellas y los criterios que se deben emplear para su identificación.

Un módulo es un fragmento de un sistema sw que se puede elaborar con relativa independencia de los demás, por lo que la idea de todo esto es poder elaborar módulos diferentes de manera que puedan ser realizados por personas diferentes trabajando ellas en paralelo.

Los tipos posibles de módulos que existen son innumerables, algunos de ellos pueden ser:

- Código fuente: Éste contiene texto fuente escrito en un lenguaje de programación determinado (el tipo de módulo que se utiliza con más frecuencia). Su formato y organización dependen de la técnica y el lenguaje empleado.
- Tabla de datos: Se emplea para tabular los datos de inicialización, experimentales, o simplemente de difícil obtención, esta tabla puede ser un módulo específico para cada forma de depósito, de manera que si se modifica el depósito sólo será necesario cambiar este modulo.
- Configuración: Un sistema se puede concebir para trabajar en entornos diferentes por lo que nos interesa agrupar en un mismo modulo toda aquella información que permita configurar el entorno concreto de trabajo.
- Otros: En general un modulo puede servir para agrupar ciertos elementos del sistema relacionados entre sí y que se puedan tratar de forma separada del resto.

El formato concreto de cada tipo de modulo dependerá de la técnica, metodología o herramienta utilizada.

Casi siempre el objeto fundamental de cualquier diseño es el de conseguir un sistema mantenible y solo en ciertos casos se sacrificará dicho objetivo para lograr una mayor velocidad de proceso o un menor tamaño de código. Una descomposición modular debe presentar ciertas cualidades mínimas para que se pueda considerar válida, estas cualidades van a ser:

4.1.1 Independencia Funcional

En la matriz requisitos/componentes del final de los documentos ADD y DDD es necesario indicar que componentes (módulos) se encargarán de realizar cada uno de los requisitos (funciones) indicados en el documento SRD, si el análisis está bien realizado y las funciones son independientes estos módulos tendrán independencia funcional entre ellos.

A continuación y mediante sucesivos pasos de refinamiento del diseño se agrupan funciones afines en un mismo módulo o por el contrario se subdividirán ciertos módulos en otros más sencillos (para llevar a cabo esa tarea hay que tener en cuenta conceptos tales como la abstracción, ocultación, etc...).

Para que un módulo posea independencia funcional debe realizar una función concreta o conjunto de funciones afines sin apenas ninguna relación con el resto de los módulos del sistema.

Al descomponer un sistema en módulos es necesario que existan ciertas relaciones entre ellos, lo que se pretende es reducir las relaciones entre éstos al mínimo, ya que una mayor independencia implica una mayor facilidad de mantenimiento o sustitución de un módulo por otro y aumenta la reutilización del módulo. Para medir la independencia funcional se emplean dos criterios.

1.- Acoplamiento: El grado de acoplamiento entre módulos es una medida de la interrelación que existe entre ellos.

Tipo de cohesión y complejidad de la interface.

Para medir el grado de acoplamiento entre módulos se utiliza la siguiente escala:

Acoplamiento por contenido.

Acoplamiento común.

Acoplamiento externo.

Acoplamiento de control.

Acoplamiento por etiqueta.

Acoplamiento de datos.

Sin acoplamiento directo.

El objetivo que se persigue es que durante el diseño el acoplamiento sea mínimo por lo que habrá que buscar descomposiciones con acoplamiento débil o como mucho moderado.

Acoplamiento por contenido: Tiene lugar cuando desde un módulo se pueden cambiar los datos locales e incluso el código de otro módulo.

Este tipo de acoplamiento fuerte sólo se puede lograr empleando un lenguaje ensamblador o de muy bajo nivel, resulta imposible el mantenimiento así como el entendimiento y depuración de un sistema con este tipo de acoplamiento.

Acoplamiento común: En este se emplea una zona común de datos a la que tienen acceso varios o todos los módulos del sistema. El empleo de este acoplamiento exige que todos los módulos estén de acuerdo en la estructura de la zona común y cualquier cambio adoptado por alguno de ellos afectará al resto, al igual que el anterior la depuración y mantenimiento resulta muy difícil.

Acoplamiento externo: En éste la zona común esta constituida por algún dispositivo externo, al que están ligados todos los módulos, aunque éste es inevitable siempre se deben buscar descomposiciones en que los dispositivos externos acepten al mínimo número de módulos.

Acoplamiento de control o moderado: En este tipo de acoplamiento una señal o dato de control es lo que se pasa de un modulo a otro, con este tipo de acoplamiento dentro de un mismo modulo se pueden tratar distintas situaciones, en general este acoplamiento se utiliza para solicitar o anular servicios adicionales a un determinado módulo.

El acoplamiento débil sólo se produce mediante el intercambio de aquellos datos que un módulo necesita de otro, si el intercambio se realiza estrictamente con los datos que se necesitan estaremos ante un acoplamiento de datos (es el mejor acoplamiento posible) y si en el intercambio se suministra una referencia que facilita el acceso no solo a los datos sino también a la estructura completa de la que forman parte (árbol, pila, vector) estaremos ante un acoplamiento por etiqueta. Ambos tipos de acoplamiento son los mas deseables en una descomposición modular.

[Máxima cohesión, mínimo acoplamiento]

2- Cohesión: Éste es complementario al anterior ya que además de buscar un acoplamiento débil entre los módulos es necesario conseguir que el contenido de cada módulo sea coherente.

La escala para medir la cohesión de un módulo es:

Cohesión abstraccional

Cohesión funcional

Cohesión secuencial

Cohesión de comunicación

Cohesión temporal

Cohesión lógica

Cohesión coincidental

Cohesión coincidental: Esta se produce cuando cualquier relación entre los elementos del módulo es una pura coincidencia, es decir, no guardan ninguna relación entre ellos (caso peor).

Cohesión lógica: Se produce cuando se agrupan en un mismo módulo elementos que realizan funciones similares, esta cohesión es la que se da en los módulos entrada/ salida o cuando se diseña un módulo para el manejo de los mensajes de error.

Cohesión temporal: Es el resultado de agrupar en un mismo módulo aquellos elementos que se ejecutan en el mismo momento, esta es la situación que se produce en la fase de inicialización o finalización del sistema.

La cohesión baja deba evitarse siempre.

Cohesión de comunicación: Es aquella que se produce cuando todos los elementos de un módulo operan con el mismo conjunto de datos de entrada o producen el mismo conjunto de datos de salida.

Cohesión secuencial: Se produce cuando todos los elementos de un módulo trabajan de manera secuencial (la salida de un elemento es la entrada del siguiente de manera sucesiva).

Con una cohesión media se puede llegar a reducir el número de módulos.

Cohesión funcional: En esta cada módulo se encarga de realizar una función concreta y específica, esta técnica por lo tanto tiene por objeto que todos los módulos tengan una cohesión funcional (técnica del diseño funcional descendente)

Cohesión abstraccional: Con la técnica de diseño basado en abstracciones un módulo con cohesión funcional será una abstracción funcional.

Esta cohesión se logra cuando se diseña un módulo como TDA con la técnica especificada anteriormente o como una clase de objetos con la técnica orientada a objetos.

Una cohesión alta es lo que se debe perseguir siempre en cualquier descomposición modular de manera que se facilitará el mantenimiento y reutilización de los módulos.

Finalmente la descomposición modular con una mayor independencia funcional se logrará con un acoplamiento débil entre sus módulos y una cohesión alta dentro de cada uno de ellos.

4.1.2 Compresibilidad

La dinámica del proceso de diseño e implantación de un sistema hace que los cambios sean frecuentes, para facilitar estos cambios es necesario que cada módulo sea comprensible de forma aislada.

El primer factor que facilita la comprensión de un módulo es su independencia funcional.

Además de la cohesión alta y acoplamiento débil es necesario cuidar los siguientes factores:

1. Identificación. Una elección adecuada del nombre del módulo así como el nombre de cada uno de sus elementos facilita mucho su comprensión. Estos nombres deberán identificar la entidad que representan.

2. Documentación. La labor de documentación de cada módulo y del sistema en general deberá seguir para facilitar la comprensión. Se deberá establecer normas y convenios de documentación en cada diseño, las cuales formaran parte del documento de diseño detallado (DDD).

3- Simplicidad. Siempre las soluciones sencillas serán las mejores, un esfuerzo del diseñador debe estar dedicado a simplificar al máximo las soluciones propuestas.

4.1.3 Adaptabilidad

La independencia funcional así como la comprensibilidad son 2 cualidades esenciales que ha de tener cualquier diseño para hacer posible su adaptabilidad.

Los factores a tener en cuenta para facilitar la adaptabilidad son:

- Previsión: resulta complicado prever que evolución futura tendrá el sistema.
- Accesibilidad: antes de abordar la nueva adaptación de un sistema, es necesario conocerlo bien, previa a la adaptación es imprescindible estudiar la estructura del sistema así como todos aquellos detalles a los que afecta la adaptación. Este trabajo sólo es posible si resulta sencilla la accesibilidad a los documentos de especificación, diseño e implementación, por lo tanto se requerirá una organización minuciosa que será posible si se utilizan herramientas CASE.
- Consistencia: Cuando se llevan a cabo adaptaciones sucesivas del sistema es importante mantener la consistencia entre todos los documentos de especificación, diseño, implementación para cada nueva adaptación. Existen herramientas que permiten el control de versiones y configuración de manera automática para mantener así la consistencia de cada adaptación.

4.2 TÉCNICAS DE DISEÑO FUNCIONAL DESCENDENTE

Aquí se incluyen todas aquellas técnicas en que la descomposición del sistema se hace desde un punto de vista funcional, es decir, atendiendo fundamentalmente a las funciones que ha de realizar el sistema, desde el punto de vista de la codificación cada módulo corresponde a un subprograma, por esta razón estas técnicas conducen a estructuras modulares.

4.2.1 Desarrollo por refinamiento progresivo

Esta técnica corresponde a la aplicación en la fase de diseño de la metodología denominada programación estructurada y que conduce a la construcción de programas mediante refinamientos sucesivos. La programación estructurada recomienda emplear en la construcción de programas solo estructuras de control claras y sencillas con un único punto inicial y un único punto final de ejecución.

En particular son: Secuencia, Selección, Iteración.

La construcción de programas basada en el concepto de refinamiento consiste en plantear inicialmente el programa como una operación global y única e ir descomponiéndola en función de otras operaciones más sencillas de manera que cada paso de descomposición consistirá en refinar la operación.

La aplicación de esta técnica en la fase de diseño consiste en realizar sólo los primeros niveles de refinamiento.

Ej. De diseño por refinamiento.

Obtener raíces→

Leer los coeficientes

Resolver ecuación→

Calcular discriminante

Calcular raíces→

Si el discriminante es negativo entonces

Calcular raíces complejas

Sino

Calcular raíces reales

Finsi.

Imprimir raíces.

4.2.2 Programación estructurada de Jackson

Esta técnica sigue las ideas de la programación estructurada, en cuanto a las estructuras recomendadas (secuencia, selección, iteración) y el método de refinamientos sucesivos para construir la estructura del programa en forma descendente.

La aportación fundamental de la Programación Estructurada de Jackson frente a la metodología general de programación estructurada esta en las recomendaciones para ir construyendo la estructura del programa.

Esta idea es apropiada para las proceso de datos y en la actualidad han sido englobadas en los sistemas de información.

La técnica original de JSP se basa en los siguientes pasos:

- Analizar el entorno del problema y describir las estructuras de datos a procesar.
- Construir la estructura del programa basada en las estructuras de datos.
- Definir las tareas a realizar en términos de las operaciones elementales disponibles.

4.2.3 Diseño Estructurado

Esta técnica es el complemento del análisis estructurado, ambas técnicas emplean los diagramas de flujos de datos.

La tarea de diseño consiste en pasar los DFD's a los diagramas de estructura, la dificultad radica en que hay que establecer una jerarquía o estructura de control entre los diferentes módulos.

Para establecer dicha jerarquía de control entre las diversas operaciones descritas en los DFD's la técnica de diseño estructurada recomienda hacer el análisis de flujo de datos global, es decir, realizar análisis denominados de flujo de transformación y de flujo de transacción.

Análisis de flujo de transformación. Consiste en identificar un flujo global de información desde los elementos de entrada al sistema hasta los de salida. Los procesos se dividen en 3 regiones denominadas.

- Flujo de entrada.
- Centro de transformación.
- Flujo de salida.

Análisis de flujo de transacción. Éste se aplicará cuando el flujo de datos se pueda descomponer en varias líneas separadas, cada una de las cuales corresponde a una función o transacción distinta de manera que solo una de estas líneas se activa para cada entrada de datos de tipo diferente, dicho análisis consiste en identificar el denominado centro de transacción a partir del cual se ramifican las líneas de flujo.

4.3 TÉCNICAS DE DISEÑO BASADAS EN ABSTRACCIONES

Dichas técnicas surgen cuando se identifican con precisión los conceptos de abstracción de datos y ocultación. La idea consiste en que los módulos se correspondan o bien con funciones o bien con Tipos de datos abstractos, las estructuras modulares resultantes pueden implementarse mediante lenguajes de programación estructurados (Pascal, C, Ada..) y lenguajes de programación orientados a objetos.

4.3.1 Descomposición modular basada en abstracciones

Esta técnica consiste en ampliar el lenguaje existente con nuevas operaciones y tipos de datos definidos por el usuario. (técnica de programación) considerada como diseño consistirá en dedicar módulos separados a la realización de cada tipo de datos abstractos y cada función importante, dicha técnica puede aplicarse en forma ascendente y descendente, si se aplica de forma descendente se puede considerar como una ampliación de la técnica de refinamiento sucesivo, si se aplica de forma ascendente se trata de ir ampliando las primitivas existentes en el lenguaje de programación y las librerías asociadas con nuevas operaciones y tipos de mayor nivel.

4.3.2 Método de Abott

En este método se sugiere una forma metódica de conseguir descripciones más formales que empleando notaciones precisas y no sólo lenguaje natural. La idea es identificar en el texto de la descripción aquellas palabras o términos que puedan corresponder a elementos significativos del diseño:

Tipos de datos, atributos y operaciones.

Los tipos de datos aparecen como sustantivos genéricos, los atributos como sustantivos, las operaciones como verbos.

Algunos adjetivos pueden sugerir valores de atributos.

4.4 TÉCNICAS DE DISEÑO ORIENTADA A OBJETOS

Los objetos añadirán algunas características adicionales como son la herencia y el polimorfismo.

La idea global de estas técnicas es que en la descomposición modular del sistema, cada módulo que obtenga la descripción de una clase de objetos o de varias clases relacionadas entre sí.

4.4.1 Diseño orientado a objetos

El diseño O.O se solapa en parte con el análisis del sistema si en él se emplean objetos para describir el modelo de objetos a desarrollar.

La técnica general de diseño se basa en los siguientes pasos:

- 1 Estudiar y comprender el problema a resolver (éste debe haberse realizado en la fase de análisis de requisitos).
- 2 Desarrollar en líneas generales una posible solución, convendrá considerar varias alternativas y elegir la más apropiada. La solución elegida deberá expresarse con suficiente detalle.

3 Formalizar dicha estrategia en términos de clases y objetos y sus relaciones.

Esto puede hacerse mediante las siguientes etapas:

- Identificar los objetos, clases, sus atributos y componentes.
- Identificar las operaciones entre los objetos y asociarlas a la clase u objeto adecuado.
- Aplicar la herencia donde sea conveniente.
- Describir cada operación en función de las otras (éstas se describirán de manera informal o en pseudocódigo).
- Establecer la estructura modular del sistema asignado, clases y objetos a módulos.

Si tras estos pasos el diseño del sistema no está suficientemente refinado, éstos se volverán a repetir hasta conseguir un diseño aceptable.

4.5 TÉCNICAS DE DISEÑO DE DATOS.

La mayoría de las aplicaciones informáticas requieren almacenar información de manera permanente, la manera típica de hacerlo es apoyando esa aplicación en una base de datos subyacente, siendo ésta donde se almacena la información permanente de una aplicación.

Una forma de organizar la base de datos es en 3 niveles, que son:

El nivel externo. Corresponde a la visión desde el punto de vista del usuario.

El nivel conceptual. Establece una organización lógica de los datos con independencia del sentido físico que tengan éstos en el campo de la aplicación.

Esta organización se resume en un diagrama de modelo de datos, bien modelo E/R o bien del tipo de un diagrama de modelo de objetos.

El nivel físico organiza los datos según los esquemas admisibles en el sistema de gestión de base de datos y/o lenguaje de programación elegido, si se utiliza una BD relacional los esquemas físicos serán las tablas.

El nivel interno.

Nivel externo: Esquemas de usuario

Nivel conceptual: Esquemas lógicos.

Nivel físico: Esquemas físicos.

4.6 DISEÑO DE BD RELACIONALES. BDR's

Partiendo del modelo E/R o del modelo de objetos será posible proporcionar reglas prácticas para obtener los esquemas de las tablas de una BD relacional que reflejen la visión lógica de los datos y que sean aceptablemente eficientes en el modelo relacional, los aspectos de eficiencia se contemplan desde 2 puntos de vista:

Por una parte se establecen las llamadas formas normales que tienden a evitar redundancias en los datos almacenados, por otra parte esta el empleo de índices para mejorar la velocidad de acceso a los datos.

4.6.1 Formas Normales FN's

Las FN's de Codell definen criterios para establecer esquemas de tablas que sean claros y no redundantes, estos criterios se enumeran de < a > nivel de restricción, dando lugar a las FN's siguientes:

1FN, 2FN, 3FN, FNBC, 4FN, 5Fn.

1FN. Se dice que una tabla se encuentra en 1FN si la información asociada a cada una de las columnas es un valor único y no una colección de valores en número variable.

2FN. Una tabla se encuentra en 2FN si esta en 1FN y además hay una clave primaria que puede estar constituida por una columna o combinación de varias que distingue cada casilla que no sea de la clave primaria, depende de toda la clave primaria.

3FN. Una tabla esta en 3FN si esta en 2FN y además el valor de cada columna que no es clave primaria depende directamente de la clave primaria, es decir, no hay independencias entre columnas que no son clave primaria.

4.6.2 Diseño de las entidades

En el modelo relacional cada entidad del modelo E/R traduce en una tabla por cada clase de entidad con una fila por cada elemento de la clase y una columna por cada atributo de la entidad.

Si una entidad esta relacionadas con otras y si quiere tener una referencia rápida entre las entidades relacionadas se puede incluir una columna que contendrá un código o número de referencia que identifique cada elemento

de datos (cada fila) en el número código de referencia puede servir como clave primaria.

4.6.3 Tratamiento de las relaciones de asociación

En el modelo de objetos se distinguen 2 tipos de relaciones que son:

- La composición o agregación.
- La herencia o especialización.

Las demás relaciones de asociación se denominarán relaciones de asociación. En el modelo E/R todas las relaciones se consideran relaciones de asociación.

Para almacenar en tablas la información de relaciones de asociación habrá que mirar la cardinalidad de la relación.

La técnica general. Consiste en traducir la relación a una tabla que contendrá referencias a las tablas de las entidades relacionadas así como los atributos de la relación (si los hay) esto será válido para relaciones con cualquier cardinalidad incluyendo las relaciones N:M.

La referencia a las entidades relacionadas se hará mediante la clave primaria de cada una.

Si la cardinalidad es 1:N será posible incluir los datos de la relación en la misma tabla de una de las entidades relacionadas.

Si la relación es 1:1 se puede fundir las tablas de las 2 entidades en 1 sola.

4.6.4 Tratamiento de las relaciones de composición

Éstas se tratan de la misma manera que las relaciones de asociación:

En las relaciones de composición la cardinalidad del lado del objeto compuesto casi siempre será 1.

4.6.5 Tratamiento de la herencia

Cuando una clase de objetos tiene varias subclases se pueden adoptar 3 formas de almacenar en tablas la información de estas entidades.

1ª FORMA: Se usará una tabla para la superclase con los atributos comunes más una tabla por cada subclase con sus atributos específicos.

2ª FORMA: Se repiten los atributos comunes en las tablas de cada subclase, en este caso se utilizará una tabla por cada subclase desapareciendo la tabla de la superclase.

3ª FORMA: Se prescinde de las tablas de la superclase con todos los atributos de cada una de las subclases.

4.6.6 Diseño de índices

Los índices permiten acceder de manera rápida a un dato concreto reduciendo de esta manera el tiempo de acceso, aunque a cambio aumenta el espacio necesario de almacenamiento y también aumenta el tiempo que se necesita para almacenar cada nuevo dato y para modificar el valor de un atributo indexado. Para acceder a datos a través de sus relaciones con otros será conveniente mantener índices sobre las claves primarias y columnas de referencia de las entidades relacionadas.

4.7 Diseño de bases de datos de objetos

En las BD orientadas a objetos no existe aún un conjunto estable de estructuras de información fundamentales. En las BD relacionales solamente se trabaja con la estructura tabla, mientras que en la BD de objetos existen una gran variedad de estructuras disponibles pueden adoptarse 2 enfoques en el diseño físico con estas bases de datos.

El primer enfoque resultará apropiado cuando la BD de objetos permita usar una gran variedad de estructuras.

El diseño de gestión de BD aportará como complemento de la persistencia de los datos.

El segundo enfoque se aplicará cuando no exista gran variedad de estructura de datos y la BD de objetos resultará análoga a una BD relacional el sistema de gestión de BD aporta la existencia implícita de identificadores de objetos.

TEMA 5: MODIFICACIÓN Y PRUEBAS

5.1 CODIFICACIÓN DEL DISEÑO.

La fase de codificación constituye el núcleo central de cualquiera de los módulos de desarrollo del sw:

Ciclo de vida, uso de prototipos, modelado en espiral, etc..

Las etapas previas de análisis y diseño tienen como misión fundamental la organización y traducción de los requisitos del cliente en módulos de programa que pueden ser codificados de forma independiente y sencilla. La fase de codificación es importante dentro del desarrollo del sw ya que en ella es donde se elabora el producto fundamental de todo desarrollo:

Programa fuente.

Un elemento esencial dentro de la codificación será el lenguaje de programación que se emplea.

Antes de iniciar la fase de codificación será necesario establecer por escrito cual será la metodología de programación que se empleará por todos los miembros del equipo de trabajo. Esta tiene tanta importancia como el lenguaje de programación elegido, en esta fase pueden intervenir un gran número de programadores durante mucho tiempo por lo que para garantizar la homogeneidad en la codificación se deben establecer normas y estilos de codificación cuyo empleo facilita bastante el mantenimiento posterior y la reusabilidad del sw reutilizado.

La codificación se ha de estructurar para facilitar su depuración y las modificaciones derivadas de las pruebas por lo que resultará más sencillo y barato localizar las causas de una disfunción y la posterior modificación del fragmento de programa afectado.

5.2 LENGUAJE DE PROGRAMACIÓN

Éstos son el medio fundamental del que disponemos para realizar la codificación, con los lenguajes actuales de programación resulta más sencillo obtener un sw de calidad fácil de mantener y con posibilidades reales de reutilización.

5.3 DESARROLLO HISTORICO

La utilidad de la mayoría de los Lenguajes de Programación ha sido experimental o de investigación siendo solo un número pequeño de ellos los que han sido utilizados en el desarrollo del sw a escala industrial.

Dentro de la evolución se suelen distinguir 4 generaciones de lenguajes que se solapan en el tiempo y éstas son:

1ª Generación: Los primeros computadores eran máquinas de válvulas donde los programas que se podían ejecutar eran muy pequeños, donde las instrucciones de programa que se introducían estaban constituidas por códigos binarios. Mas tarde aumentarán el número de computadores y con ello la aparición de los lenguajes ensambladores que consistían en asociar a cada instrucción del computador un nemotécnico que recuerde cual es su función.

Los ensambladores se consideran lenguajes de 1ª Generación con un nivel de abstracción muy bajo. Este tipo de programación resulta compleja, da lugar a errores difíciles de detectar, exige que el programador conozca muy bien la arquitectura del ordenador y sobre todo hay que adaptar la solución a las particularidades de cada ordenador concreto. Actualmente solamente se programan en ensamblador pequeños trozos de programa que se podrán insertar en un Lenguaje de Alto Nivel.

2ª Generación: El aumento de la capacidad de memoria y disco hizo que se pudieran abordar programas más grandes y complejos. A finales de los 50 empezaron a desarrollar los primeros lenguajes de alto nivel cuyo carácter más importante era que éstos no dependían de la estructura del ordenador concreto y por primera vez se programaba en alto nivel de manera simbólica.

También se crea la herramienta para el desarrollo del sw, se pasa a trabajar con variables simbólicas de distintos tamaños y estructuras. Se dispone de las primeras estructuras de control para los bucles genéricos o selección de varios caminos de ejecución, etc...

Ej. Cobol, Fortran,...

3ª Generación: A finales de los 60 y principios de los 70 se consolidan las bases teóricas y prácticas de la programación estructurada apareciendo la necesidad de una nueva generación de lenguajes fuertemente tipados que faciliten la estructuración de códigos y datos con redundancia entre la

declaración y el uso de cada tipo de datos, con esto se facilita la verificación en compilación de las posibles inconsistencias de un programa.
Ej. Pascal, Modula, C, Ada.

Dentro de esta misma generación y de forma paralela se han desarrollado lenguajes asociados a otros, paradigma de la programación (OO, funcional) los lenguajes que soportan estos paradigmas son: Lisp, Prolog, C++, Smalltalk.

4ª Generación: Estos lenguajes tratan de ofrecer el mayor nivel de abstracción, normalmente estos lenguajes no son de propósito general considerándose como herramientas específicas, se pretende lograr que cualquiera que con pocos conocimientos de programación puedan utilizar estos lenguajes para realizar aplicaciones sencillas, se pueden utilizar para la realización de prototipos que serán mejorados con lenguajes de 3ª Generación, una posible agrupación según su aplicación sería:

- Base de Datos, estos lenguajes permiten acceder y manipular la información almacenada mediante un conjunto de ordenes de petición. Estos lenguajes tienen como ventaja la aportación de gran versatilidad a la Base de Datos y permiten que el propio usuario diseñe sus listados, informes, etc..
- Generadores de programas, Con éstos se pueden construir elementos abstractos funcionales. Con la utilización de éstos siempre se consigue ahorrar en el tiempo de desarrollo, la mayoría de los generadores de programas sirven para realizar aplicaciones de gestión.
- Cálculo, Existe una amplia gama de herramientas de cálculo destinadas a simplificar tareas científicas o técnicas, algunas de estas herramientas han llegado a convertirse en auténticos Lenguajes de 4ª Generación. Ej: Hojas de cálculo, herramientas de simulación,...
- Otros, Aquí podemos englobar todo tipo de herramientas que permitan una programación de cierta complejidad y para ello utilizan un lenguaje.

5.4 PRESTACIONES DE LOS LENGUAJES

Es importante conocer las prestaciones que ofrecen los lenguajes ya que esto facilitará la tarea de selección en cuanto al más adecuado para una determinada aplicación, estas prestaciones se agrupan en 4 grandes apartados:

Estructuras de Control: Son conjuntos de sentencias o instrucciones que se encuadran dentro de la parte ejecutiva de un programa, también hay que tener en cuenta algunas estructuras específicas para el manejo de excepciones y para la programación concurrente, además de las de la programación estructurada.

Programación estructurada: Cualquier lenguaje imperativo dispone de sentencias que facilitan la programación estructurada, para ello siempre existen sentencias para programar:

La secuencia

La selección (selección general y selección por casos)

La Iteración (repetición, bucle con contador, bucle indefinido)

Todos los lenguajes permiten definir subprogramas mediante el empleo de procedimientos o funciones los cuales se pueden definir de manera recursiva cuya ventaja es la sencillez y claridad con que se representa en la resolución de determinados problemas.

Manejo de excepciones: Éstos son errores o sucesos inusuales que tienen lugar durante la ejecución de un programa. Los errores pueden tener distintos orígenes:

Errores humanos.

Fallos Hw.

Errores Sw

Datos de entrada vacíos.

Valores fuera de rango.

Si las situaciones anteriores no han sido previstas el programa puede abortar.

Es necesario algún mecanismo apropiado para desviar de manera automática la ejecución hacia un fragmento de código apropiado en caso de que ocurra alguna situación anormal. Dicha transferencia de control se contempla en lenguajes que incorporan el manejo de excepciones.

Concurrencia: Algunos de los aspectos fundamentales que se han de tener en cuenta para el diseño y programación son , las tareas que se deben ejecutar concurrentemente, sincronización de tareas, comunicación entre tareas e interbloqueos.

Todos los lenguajes concurrentes permiten declarar distintas tareas y definir la forma en que se ejecutarán concurrentemente, existiendo distintas formas de abordar esto:

- **Las corrutinas**: Estas tienen una estructura semejante a los subprogramas pero entre ellas se puede transferir el control de ejecución en cualquier momento. La concurrencia se limita a un avance de la ejecución de todas las corrutinas pero secuenciando ese avance a través de un acuerdo entre ellas. Esta propuesta es la que utiliza Modula-2.
- **For join**: Una tarea puede arrancar la ejecución concurrente de otras tareas a través de la orden for, finalizándose la concurrencia con la orden join invocando por la misma tarea que ejecutó el For, esta propuesta es la que emplea Unix.
- **Co-begin-co-end**. Todas las tareas que se deben ejecutar concurrentemente se declaran dentro de una construcción co-begin-co-end. La finalización de la concurrencia exige que las tareas hayan finalizado.
- **Procesos**: Cada tarea se declara como un proceso, los procesos declarados se ejecutarán concurrentemente desde el comienzo del programa y no es posible iniciar ninguno nuevo. Esta propuesta es utilizada por Pascal concurrente.

Para lograr la sincronización y cooperación entre tareas los lenguajes concurrentes disponen de unas estructuras donde la mayoría de ellas son intercambiables, en cuanto a que permiten resolver los mismos problemas con una mayor o menor dificultad.

Las estructuras son mayor aceptación se clasifican en 2 grandes grupos:

Las variables compartidas.

Los semáforos.

Los monitores.

Regiones críticas condicionadas

Paso de mensajes

"Communicating sequential processes "

"Rendezvous" de Ada

Llamadas a procedimientos remotos.

El primer grupo necesita que todas las tareas se ejecuten en un mismo ordenador o en distintos pero utilizando una memoria compartida, de manera que todas las tareas puedan acceder a las variables compartidas que emplean para comunicarse.

El segundo grupo se puede utilizar para la sincronización y cooperación entre tareas que se ejecutan en ordenadores que no tienen memoria común (procesos distribuidos) y que están conectados por una red de comunicaciones, que les permite comunicarse a través de mensajes.

Para mostrar los problemas de sincronización y cooperación entre tareas se emplean semáforos, los cuales son estructuras de bajo nivel que se utilizan para programar la concurrencia entre tareas y la cooperación entre éstas.

5.4.2 Estructuras de datos

Nos referimos a las distintas formas que emplean los lenguajes para estructurar los datos que manejan.

- Datos simples: Dato de tipo entero, de tipo real (con precisión simple o doble precisión), datos de tipo carácter, datos de tipo string, enumerados, subrango,...
- Datos compuestos: Se definen como combinaciones de tipos de datos simples y compuestos ya definidos, entre ellos tenemos los vectores, las matrices, los registros, el conjunto, datos dinámicos, etc..
- Constantes: En los lenguajes modernos se pueden declarar constantes simbólicas con nombre, en algunos lenguajes las constantes han de declararse antes que los tipos. Las constantes mantienen siempre el dato.

Comprobación de tipos: Las operaciones que se pueden realizar con los datos de un programa dependerán del nivel de comprobación de tipos que corresponda al lenguaje utilizado, al menos se pueden distinguir los siguientes niveles:

- Nivel 0 (sin tipos). A este nivel pertenecen los lenguajes en los que no se pueden declarar nuevos tipos de datos y los que se utilizan deben pertenecer a sus tipo predefinidos. No es necesario que el compilador realice ninguna comprobación de tipos ya que es responsabilidad del programador.
- Nivel 1 (tipo automático). En este caso es el compilador el encargado de decidir cual es el tipo más adecuado para cada dato que utiliza el programador siendo este también el que se encarga de convertir al tipo adecuado los operandos de una expresión cuando estos son incompatibles entre sí.
- Nivel 2 (tipado débil). También se realiza una conversión automática de tipos pero solo entre datos que poseen ciertas similitudes, no es posible la conversión automática de un valor lógico a numérico o viceversa, pero si son factibles las conversiones entre enteros y reales, las conversiones siempre se hacen hacia el tipo de mayor rango de precisión.
- Nivel 3 (tipo semi-rígido). El lenguaje más representativo de dicho nivel es Pascal, todos los datos que se quieren usar deberán ser declarados previamente con sus respectivos tipos. No es posible realizar operaciones entre tipos de datos incompatibles. En este tipo existen algunas vías de escape que permiten evitar que los procedimientos y funciones a la hora de comprobar el número de argumentos y el tipo de éstos coincida dicha declaración con su utilización, siendo la vía de escape más utilizada la compilación separada.
- Nivel 4 (tipado fuerte). En éste no existe ningún escape y el programador esta obligado a hacer explícita cualquier conversión de tipo que necesite realizar. Las comprobaciones de tipo se realizarán en compilación, carga y ejecución.

Abstracciones de objetos.

Desde el punto de vista del diseño se distinguen 3 formas de abstracción:

- Abstracciones funcionales.
- Tipos abstractos.
- Máquinas abstractas.

Para la codificación de un diseño basado en las 2 primeras formas, los lenguajes de propósito general disponen de estructuras que simplifican bastante esta labor. En cambio para la programación de una máquina abstracta se requieren herramientas mucho más específicas.

- Abstracciones funcionales: Según el lenguaje el nombre que se le da a la abstracción funcional puede cambiar, puede llamarse subprogramas, subrutinas, procedimientos... En todos ellos se tienen que definir la interfaz de la abstracción, se tiene que codificar la operación que realiza y finalmente se dispone de un mecanismo para la abstracción de la función. En la mayoría de los lenguajes permanece oculta la codificación de la operación para quien hace uso de éste.
- Tipos abstractos de datos: Para la codificación del tipo abstracto de datos se deben agrupar en una única entidad el contenido o atributos de la abstracción y las operaciones definidas para el manejo de los atributos. Debe existir además un mecanismo de ocultación que impida el acceso al contenido por una vía distinta a las que ofrece las operaciones definidas.
- Objetos: Existe un gran paralelismo entre abstracciones y objetos pero desde el punto de vista de la programación las diferencias son más importantes, los conceptos de polimorfismo y herencia aparecen ligados a los sujetos y para su codificación es necesario que los lenguajes de programación dispongan de unas construcciones distintas, solamente con los lenguajes orientados a objetos resulta factible la codificación de diseños orientados a objetos con herencia simple, múltiple o polimorfismo (C++, Smalltalk, Java)

Modularidad

Este concepto esta ligado a la división del trabajo y al desarrollo en un equipo sw.

La primera que se exige es la compilación separada que permite compilar y separar cada módulo.

Si es posible comparar en tiempo de compilación que el uso de un elemento es consistente con su definición, se dirá que la compilación separada es segura.

5.5 CRITERIO DE SELECCIÓN DEL LENGUAJE

El Lenguaje de programación es uno de los elementos más importantes de cualquier desarrollo ya que es la herramienta de trabajo que usarán mayor número de personas y además tiene una influencia decisiva en la depuración y mantenimiento de la aplicación.

Alguno de los criterios de selección a analizar son:

1- Imposición del cliente: Es el cliente el que fija el lenguaje que se debe utilizar.

2- Tipo de aplicación: Aunque con las prestaciones de cualquier lenguaje de última generación se pueden realizar aplicaciones para diversos campos hay que tener en cuenta que existen lenguajes orientados a un campo de aplicación concreta. Ej. En aplicaciones de tiempo real muy crítico o Hw muy especial estará justificado el empleo de lenguajes ensambladores para aplicaciones de gestión, lo normal es utilizar Cobol, para aplicaciones de Inteligencia Artificial utilizar Lisp y Prolog para aplicaciones de diseño O.O, C++, Java