

Resumen de Ingeniería de Software.

Tema 1. Introducción.

1.1. Concepto de Ingeniería de Sistemas.

La Ingeniería de Sistemas es un marco general dentro del cual se pueden situar las disciplinas de ingeniería particulares.

1.1.1. Concepto de sistemas.

En el diccionario encontramos la siguiente definición: Sistema: conjunto de cosas que ordenadamente relacionadas entre sí contribuyen a determinado objeto.

La ingeniería de sistemas atiende a los aspectos de organización de estos sistemas, tanto en lo referente al sistema en sí, como al proceso de su desarrollo.

1.1.2. Sistemas basados en computador.

Los sistemas que han de concebir y construir el ingeniero en informática son sistemas basados en computadores.

1.1.3. Componentes hardware, software y humanos.

Estas son las posibilidades para el tratamiento de información:

1. La operación puede ser realizada por algún elemento físico (hardware) del sistema.
2. la operación puede ser programada, desarrollando el programa (software) apropiado.
3. La operación se realiza manualmente por el usuario del sistema.

1.2. Características del software.

Hay dos grandes grupos de elementos: Hardware y software.

Hardware:

- Se obtienen mediante un proceso de fabricación. Alto costo.
- Implica desgaste o envejecimiento.

Software:

- Bajo costo de fabricación.
- No se desgasta.

1.3. Concepto de Ingeniería de Software.

Hay muchas actividades a lo largo del tiempo que constituye lo llamado “ciclo de vida” del desarrollo de software.

1.3.1. Perspectiva histórica.

Antes de los años 70: al crecer los sistemas de empezó a aplicar metodologías de desarrollo específicas.

Años 70: Aparecen herramientas de ingeniería. PJ. CASE.

Años 80: Se aplican estas herramientas.

Años 90: Se automatizan más las herramientas. PJ. IPSE, ICASE.

1.3.2. La crisis del software.

Al evolucionar el hardware con fuerte ritmo, el software entra en crisis.

1.3.3. Mitos del software.

- El hardware es mucho más importante que el software.
- El software es fácil de desarrollar.
- El software consiste exclusivamente en programas ejecutables.
- El desarrollo de software es solo labor de programación.
- Es natural que el software contenga errores.

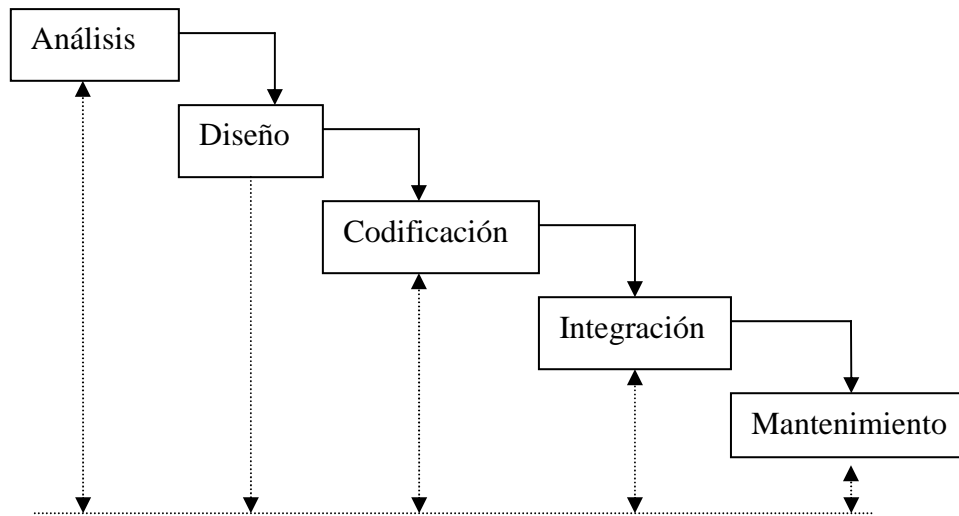
1.4. Formalización del proceso de desarrollo.

1.4.1. El ciclo de vida del software. Modelos clásicos.

Hay dos formas bien establecidas de plantear el proceso de desarrollo son:

- El modelo en cascada.
- El modelo en V.

1.4.1.1. El modelo en cascada.

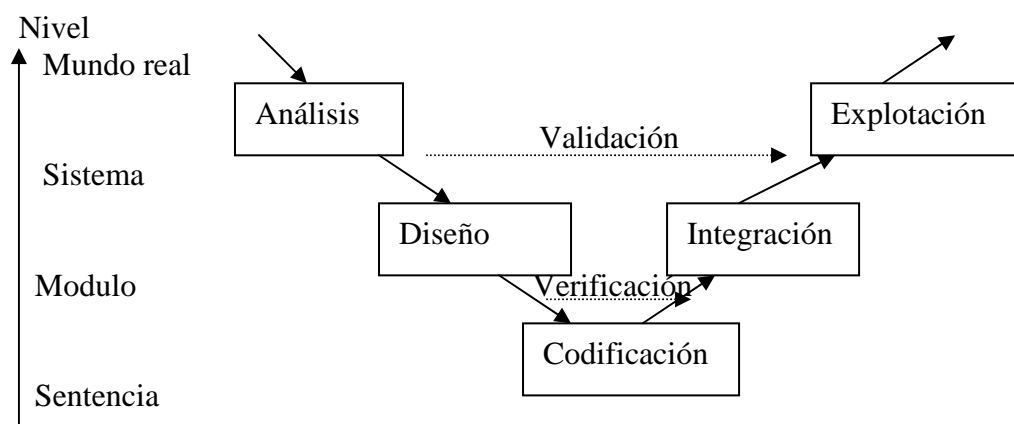


Este modelo tiene las siguientes fases y documentos:

1. Análisis: Análisis de las necesidades de usuario. Documento de requisitos del Software (SRD)
2. Diseño: Descomposición y organización del sistema en elementos componentes. Documento de Diseño del Software (SDD)
3. Codificación: Se programa cada elemento componente separado. Código fuente.
4. Integración. Combinar los elementos y probar el sistema completo. El sistema software, ejecutable.
5. Mantenimiento: Realizar cambios en la explotación del sistema. Documento de cambios.

Hay variantes con más fases, modelo ampliado.

1.4.1.2. El modelo en V.



Se basa en una secuencia de fases análoga a la del modelo en cascada, pero se da especial importancia a la visión jerarquizada de las distintas partes del sistema a medida que avanza el desarrollo.

Verificación: Comprobación de que una parte del sistema cumple las especificaciones.

Validación: Comprobación de que un elemento del sistema cumple las necesidades del usuario.

También hay un modelo ampliado.

1.5. Uso de prototipos.

Un prototipo es un sistema auxiliar que permite probar experimentalmente ciertas soluciones parciales a las necesidades del usuario o los requisitos del sistema.

1.5.1. Prototipos rápidos.

Tienen finalidad de solo adquirir experiencia, sin pretender aprovecharlos como producto. Se desarrollan en poco tiempo, de ahí su nombre.

Se aprovechan estos prototipos en fase de análisis o/y diseño.

1.5.2. Prototipos evolutivos.

Se trata de aprovechar al máximo su código. Será una realización parcial del sistema. Se harán varias versiones sucesivas del sistema final hasta llegar al sistema deseado.

Modelo evolutivo.

1.5.3. Herramientas para realización de prototipos.

- Lenguajes de 4ª generación: Adecuadas para la construcción de Sistemas de Información.

- Lenguajes de muy alto nivel: Siguen estilo declarativo en lugar de operacional. PJ. Prolog o Smalltalk.

- Lenguajes de especificación: Formalizan la especificación de requisitos de software. PJ. VDM, Z.

Otra técnica para construcción de prototipos es reutilización de software.

1.6. El modelo en espiral.

Consta de estas partes:

- Planificación: Decidir qué parte del desarrollo se abordará en este ciclo.

- Análisis de riesgo: Selección de la alternativa para la realización del proyecto más ventajoso.

- Ingeniería: Corresponde a lo indicada en los modelos clásicos.

- Evaluación: Análisis de los resultados de la fase anterior.

1.7. Combinación de modelos.

Se pueden combinar los modelos.

1.8. Mantenimiento del software.

Es la etapa final del ciclo de vida.

1.8.1. Evolución de las aplicaciones.

Los tipos de mantenimiento son:

- Mantenimiento correctivo: Corregir errores no detectados.

- Mantenimiento adaptativo: Adaptar un sistema a uno más reciente.

- Mantenimiento perfectivo: Perfección del sistema con nuevas versiones.

1.8.2. Gestión de cambios.

Tiene dos enfoques distintos:

- Nuevo desarrollo: Si los cambios afectan a la mayoría de los componentes del producto.
- Modificación de algunos elementos.

Hay dos clases de documentos:

- Informe de problema: Describe una dificultad en la utilización del producto.
- Informe de cambio: Describe la solución dada a un problema.

1.8.3. Reingeniería.

Ingeniería inversa: Consiste en tomar el código fuente y tratar de construir la documentación.

Reingeniería: Generar un sistema bien organizado y documentado a partir de un sistema inicial deficiente.

1.9. Garantía de calidad de software.

1.9.1. Factores de calidad.

Existen los siguientes:

Corrección, fiabilidad, eficiencia, seguridad, facilidad de uso, mantenibilidad, flexibilidad, facilidad de prueba, transportabilidad, reusabilidad e interoperatividad.

1.9.2. Plan de garantía de calidad.

Documento que materializa la organización del proceso de desarrollo de software.

1.9.3. Revisiones.

Consiste en inspeccionar el resultado de una actividad y determinar si es aceptable o contiene defectos en cada etapa.

1.9.4. Pruebas.

Consisten en hacer funcionar un producto software o una parte de él en condiciones determinadas.

1.9.5. Gestión de configuración.

La definición de configuración es: “disposición de las partes que componen una cosa y le dan su peculiar figura”.

La configuración del software hace referencia a la manera en que diversos elementos se combinan para construir un producto software bien organizado.

Las técnicas son:

- Control de versiones: Almacenan ordenadamente las versiones de cada elemento de configuración.
- Control de cambios: garantizan que las diferentes configuraciones de software se componen de elementos.

Línea Base (de Piattini) o configuración de referencia: Son puntos de referencia en el ciclo de vida del sistema. Son los productos tangibles del proceso de desarrollo del sistema.

Tema 2. Especificación del software.

2.1. Modelado del Sistemas.

Nos referimos a un modelo completo y preciso del comportamiento u organización del sistema software.

2.1.1. Concepto de modelo.

Estudia QUE debe hacer el sistema. Los objetivos de los modelos son:

- Facilitar la comprensión del problema a resolver.
- Establecer un marco para la discusión.
- Fijar las bases para realizar el diseño.
- Facilitar la verificación del cumplimiento de los objetivos del sistema.

2.1.2. Técnicas de modelado.

2.1.2.1. Descomposición. Modelo jerarquizado (Top-Down).

El problema global queda subdividido en un cierto número de subproblemas.

2.1.2.2. Aproximaciones sucesivas.

Adaptación sucesiva de una aplicación hasta obtener la aplicación deseada. Puede ser una aplicación contigua o un prototipo que mejoramos continuamente.

2.1.2.3. Empleo de varias notaciones.

Para simplificar el modelo se usarán notaciones alternativas o complementarias.

2.1.2.4. Considerar distintos puntos de vista.

Para poder concretar la creación de un modelo es necesario adoptar un determinado punto de vista, eligiendo el más adecuado siempre.

2.1.2.5. Realizar un análisis del dominio.

Por dominio entenderemos el campo de aplicación en el que se encuadra el sistema a desarrollar. P.J. Sistema contable.

Para realizar este análisis es aconsejable estudiar los siguientes aspectos:

- Normativa.
- Otros sistemas con los que se comunique.
- Bibliografía sobre el tema.

Las ventajas de este estudio son:

- Facilita la comunicación entre analista y usuario del sistema: El analista realiza el análisis y el usuario es el experto que conoce los detalles de la aplicación.
- Eliminación de elementos superfluos: Sólo se crea lo realmente necesario.
- Reutilización del software desarrollado.

2.2. Análisis de requisitos del software.

Con el análisis de requisitos se trata de caracterizar el problema a resolver, mediante el modelo global.

Analista: Tratará de buscar al cliente adecuado.

El cliente puede ser:

- Usuario final de la aplicación.
- Encargadote elaborar las especificaciones del software.
- Persona que financia la aplicación.

2.2.1. Objetivos del análisis.

El objetivo global es obtener las especificaciones que debe cumplir el sistema a desarrollar.

El modelo global del sistema tendrá estas propiedades:

1. Completo y sin omisiones.
2. Conciso y sin trivialidades.
3. Sin ambigüedades.

4. Sin detalles de diseño o implementación.
5. Fácilmente entendible por el cliente.
6. Separando requisitos funcionales (detalles de funcionamiento del sistema) y no funcionales (detalles técnicos de hardware y software, seguridad, mantenimiento, etc.).
7. Dividiendo y jerarquizando el modelo.
8. Fijando los criterios de validación del sistema.

2.2.2. Tareas del análisis.

El análisis se desarrolla en los siguientes pasos:

1. Estudio del sistema en su contexto.

Analista: Conocer el medio en el que se va a desenvolver y análisis del dominio de la aplicación.

2. Identificación de necesidades.

Analista: Concretar las necesidades con los medios disponibles, dentro del presupuesto y plazos de entrega. Convencer de que la opción adoptada es la mejor

Cliente: Solicitar todas las funciones que siente la necesidad.

3. Análisis de alternativas. Estudio de viabilidad.

Analista: Buscar alternativa que cubra las necesidades reales detectadas en el paso anterior y que tenga en cuenta su viabilidad técnica y económica. Si no es viable la solución encontrar alternativas.

4. Establecimiento del modelo del sistema.

Con las conclusiones anteriores, estas se deben plasmar en el modelo del sistema. Se usarán los medios disponibles (herramientas CASE, herramientas graficas, etc.) para simplificar la elaboración del modelo y facilitar la comunicación entre analista, cliente y diseñador.

5. Elaboración de Documento de Especificación de Requisitos (SRD).

Documento que recoge todas las conclusiones del análisis y fija las condiciones de validación del sistema concluido su desarrollo e implementación.

Tiene que estar bien redactado por ser útil para el cliente y analista.

6. Revisión continuada del análisis.

Se podrán cambiar requisitos a lo largo del desarrollo del sistema y en el análisis e implementación.

Cliente: Puede cambiar de criterio y modificar los requisitos.

Revisión continuada del análisis y el SRD cuando hay cambios.

2.3. Notaciones para la especificación.

Deben ser fácil entendible por el cliente y todos los que participan en el análisis y desarrollo del sistema.

2.3.1. Lenguaje natural.

Es suficiente para sistemas de baja complejidad. Los inconvenientes son las imprecisiones, repeticiones e incorrecciones que se producen.

Se empleará esta notación para aclarar cualquier aspecto concreto del sistema que no reflejen el resto de anotaciones. Para organizar y estructurar los requisitos se tomará cada uno como una cláusula de contrato entre analista y cliente. PJ. 1. Funcionales, 2. Calidad, etc.

El lenguaje natural estructurado es más formal que el lenguaje natural. Tiene la misma estructura. PJ. SI .. ENTONCES.

2.3.2. Diagrama de Flujo de Datos (DFD).

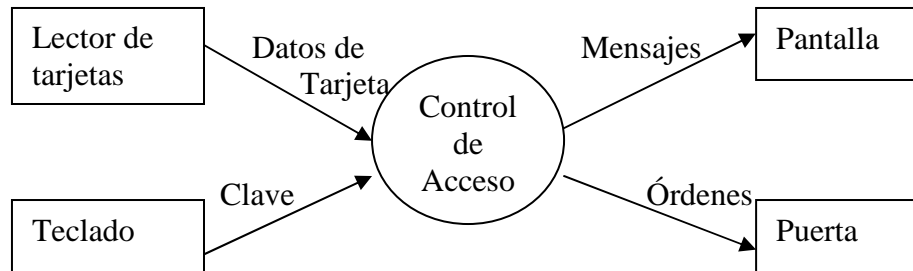
Mediante una notación gráfica se modela de forma sencilla las transformaciones y los flujos de datos (de entrada y de salida).

La notación DFD básica es:



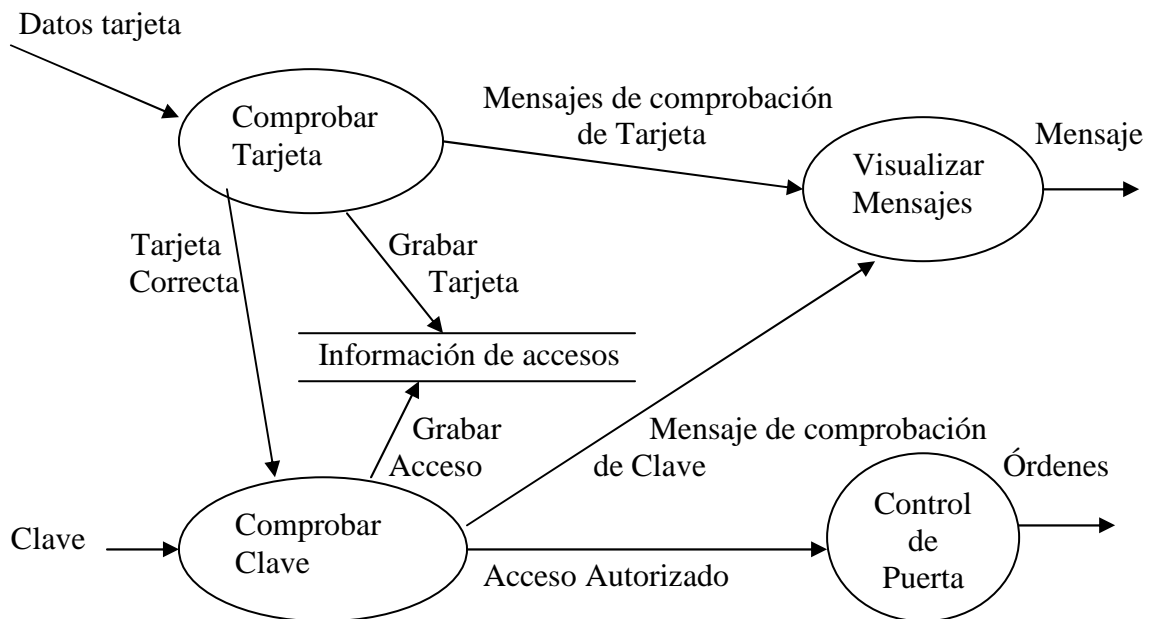
NOTA. Si utilizamos otra notación lo explicamos.

En un nivel 0, tendremos un DFD de contexto (DC). En nuestro ejemplo de sistema de control de acceso tendremos:



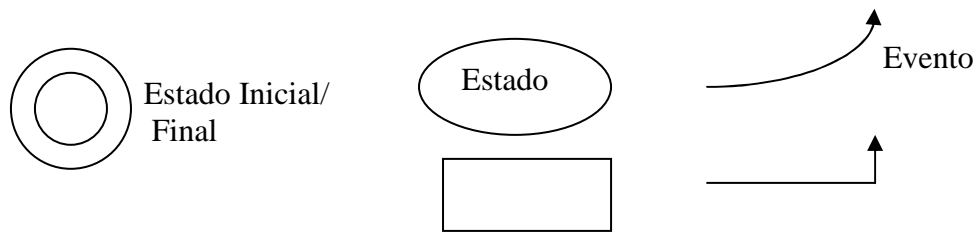
Sucesivamente en cada nivel subdividiremos cada proceso en subprocesos. En todos los niveles habrá flujo de entrada y de salida del nivel anterior, entidades, flujos, almacenes de datos y flujos propios de ese nivel. Se verá en el siguiente ejemplo.

La notación a emplear será el lenguaje natural estructurado.

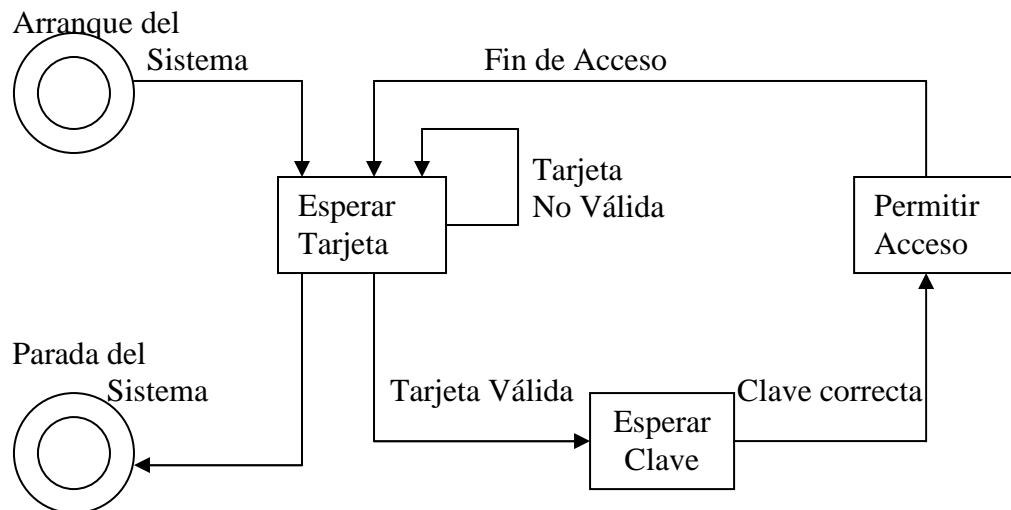


2.3.3. Diagrama de Transición de Estados (DTE).

Es la notación específica para describir el comportamiento dinámico del sistema o partir de los estados elegidos como más importantes. Se implementa como DFD desde otro punto de vista. Las especificaciones son los estados. Tiene dos notaciones:



En nuestro ejemplo tendremos lo siguiente:



2.3.4. Descripciones funcionales. Pseudocódigo.

Los requisitos funcionales son una parte muy importante de las especificaciones del sistema. Por tanto, es esencial que las descripciones funcionales se realicen empleando una notación precisa y sin ambigüedades. Se usará lenguaje natural estructurado, pero si puede se usara pseudocódigo, que es más preciso.

Las notaciones básicas del pseudocódigo o PDL serán:

- A. Selección. IF...THEN.
- B. Selección por casos. CASE.
- C. Interacción con pre-condición. WHILE...DO.
- D. Interacción con post-condición. REPEAT...UNTIL.
- E. Número de iteraciones conocido. FOR...DO.

2.3.5. Descripción de datos (diccionario de datos).

Se trata de detallar la estructura interna de los datos que maneja el sistema. Solo se deben escribir aquellos que resulten relevantes para entender el sistema.

La notación se conoce como diccionario de datos. Tendrán esta información:

1. Nombres: La denominación que usa ese dato en el resto de la especificación.
2. Utilidad: Se indicarán todos los procesos, descripciones funcionales que utilice el dato.
3. Estructura: Se indicaran los elementos de los que está constituido el dato, utilizando esta notación:

A + B: Concanetación de A y B.

[A | B]: Selección entre A y B.

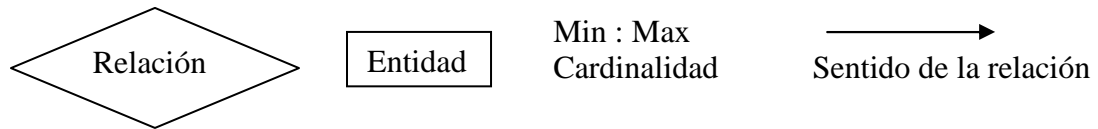
{A}^N: Repetición N veces de A.

/ /: Comentario.

= : Separador de nombre y descripción del elemento.

2.3.6. Diagrama de modelo de datos (modelo de E-R).

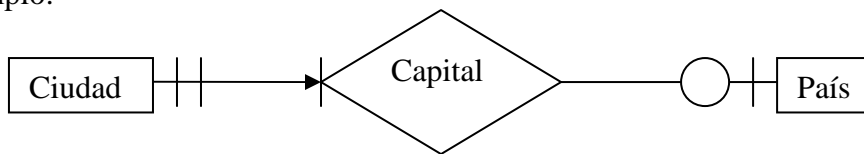
Si un sistema maneja una cierta cantidad de datos relacionados entre si, es necesario establecer una organización que facilite las operaciones que se requieren realizar con ellos. La notación básica es:



En el diagrama E/R se indica la cardinalidad como sigue:



Ejemplo:



2.3.7. Documento de Especificación de Requisitos (SRD).

Es el encargado de recoger todo el fruto de análisis del sistema y es el punto de partida para el diseño.

Recogerá estas partes:

1. Introducción: Visión general del documento.
2. Descripción general: Visión general del dominio + (DFD/DTE/E-R).
3. Requisitos específicos: Contiene lista detallada y completa de los requisitos del sistema, especificando si es obligatorio, recomendable u opcional.
 - 3.1. Requisitos funcionales: Describen las funciones o QUÉ debe hacer el sistema.
 - 3.2. Requisitos no funcionales: Tales como capacidad, interfase, operación, calidad, etc.
4. Apéndices: Todos los elementos que completen el contenido del documento y no están recogidos en otros documentos accesibles.

Tema 3. Fundamentos del Diseño de Software.

Estudia el CÓMO se debe hacer el sistema.

3.1. Introducción.

La definición de diseño es: “descripción o bosquejo de alguna cosa, hecha por palabras”. Se trata de definir y formalizar la estructura del sistema con el suficiente detalle como para permitir su realización física.

La etapa de diseño es la más importante en la fase de desarrollo del software.

En el diseño se pasará las ideas informales recogidas en el SRD a definiciones detalladas y precisas, siendo este documento el punto de partida del diseño. En él se utilizan la mayor parte posible de módulos ya desarrollados.

La experiencia juega un papel muy importante en el diseño del software, ya que sin ella es imposible llegar a hacerlo.

Las actividades en el diseño del sistema son:

1. Diseño arquitectónico: Se deben abordar los aspectos estructurales y de organización del sistema y su posible división en subsistemas o módulos.
2. Diseño detallado: Se aborda la organización de los módulos. Se trata de determinar cual es la estructura mas adecuada para cada uno de los módulos. Para esto se crean los módulos de definición.
3. Diseño procedimental: Se encarga de abordar la organización de las operaciones o servicios que ofrecerá cada uno de los módulos. Se emplearán módulos de implementación.
4. Diseño de datos: Se aborda la organización de la base de datos del sistema. El punto de partida son los diccionarios de datos y los diagramas E-R.
5. Diseño de la interfaz de usuario: Se encarga de la organización de la interfaz de usuario. Ergonomía de la interfaz.

El resultado de estas actividades constituye el SDD (Documento del Diseño de software).

3.2. Concepto de base.

En cualquier diseño se sigue estos conceptos base:

3.2.1. Abstracción.

En el diseño del software es importante identificar los elementos realmente significativos de los que consta y abstraer la unidad específica de cada uno. Se quiere conseguir elementos fácilmente mantenibles y utilizables.

Hay tres formas de abstracción:

- Abstracción funcionales: Sirve para crear expresiones o acciones parametrizadas mediante el empleo de funciones o procedimientos. PJ. Comprobar clave.
- Tipos abstractos: Sirven para crear los nuevos tipos de datos que se necesitan para abordar el diseño del sistema. PJ. Leer Tarjeta.
- Máquinas abstractas: Se define de una manera formal el comportamiento de una maquina. PJ. Protocolo de comunicaciones.

3.2.2. Modularidad.

Se divide al sistema en módulos o partes claramente diferenciadas. Las ventajas son:

- Claridad: Es más fácil se entender y manejar el sistema.
- Reducción de costos: Es más barato de desarrollar, depurar, documentar, probar y mantener.
- Reutilización: Los módulos se podrán usar en otras aplicaciones.

PJ. MODULE de Modula-2.

3.2.3. Refinamiento.

Se deberá acercar al lenguaje natural (código del SRD) y el de programación mediante sucesivas aproximaciones o refinamientos.

3.2.4. Estructura de datos.

La organización de la información es una parte esencial del diseño de un sistema software. Las decisiones respecto a que datos se manejan y la estructura afectan directamente al diseño: abstracciones, módulos, algoritmos, etc.

Las estructuras fundamentales son: registros, conjuntos, formaciones, listas, pilas, colas, árboles, grafos, tablas y fichero.

3.2.5. Ocultación.

Cuando se diseña la estructura de cada uno de los módulos de un sistema, se debe hacer de tal manera que dentro de él queden ocultos todos los detalles que son irrelevantes para su utilización. Esconder todos los detalles del diseño que resultan irrelevantes para la utilización del software. Las ventajas son:

- Depuración: Es más sencillo de detectar el módulo que no funcione adecuadamente.
- Mantenimiento: Un cambio en cualquier módulo del sistema no afecta al resto.

3.2.6. Genericidad.

Consiste en agrupar aquellos elementos del sistema que utilizan estructuras semejantes o que necesitan de un tratamiento similar. Se diseña un elemento genérico con las características comunes a todos los elementos agrupados. Posteriormente cada elemento agrupado se diseñará como un caso particular del genérico.

Se suele desvirtuar el uso de la genericidad en lenguajes como Pascal o Modula-2 al imponer unas fuertes restricciones en el manejo de datos de distinto tipo.

3.2.7. Herencia.

Cuando hay elementos con características comunes se establece una clasificación o jerarquía entre esos elementos del sistema partiendo de un elemento “padre” que posee una estructura y operaciones básicas, heredando esto sus “hijos” para ampliarlos, mejorarlos o adaptarlos.

Se consigue reutilizar una gran cantidad de software ya desarrollado. Hay herencias simples (de un “padre”) y múltiples (de varios “padres”)

3.2.8. Polimorfismo.

Se define polimorfismo como propiedad de los cuerpos que pueden cambiar de forma sin variar su naturaleza.

El concepto de polimorfismo engloba varias probabilidades:

1. El concepto de genericidad.
2. El concepto de polimorfismo está relacionado con el de herencia. Hay dos tipos de polimorfismo:
 - Anulación: una operación particular anula la más general. P.J. Notación de círculo anula la de elipses.
 - Diferido: Se plantea la operación para el elemento “padre”, pero su corrección se deja diferida para que cada uno de los “hijos” concrete su forma específica.
3. Sobrecarga: Consiste en definir varias operaciones iguales, pero que tendrán usos distintos dependiendo del elemento que lo invoque. P.J. La suma.

2.3.9. Concurrencia.

Hay concurrencia al haber varios procesos o tareas solicitando acceso a la misma región de memoria. Se tiene en cuenta lo siguiente:

1. Tareas concurrentes: determinar qué tareas se deben ejecutar en paralelo.
2. Sincronización de tareas: Determinar los puntos de sincronización entre las distintas tareas.
3. Comunicación entre tareas: Determinar si la cooperación se basa en el empleo de datos compartidos o mediante el paso de mensajes entre las tareas.

4. Interbloqueos (Deadlock): Estudiar los posibles interbloqueos entre tareas.

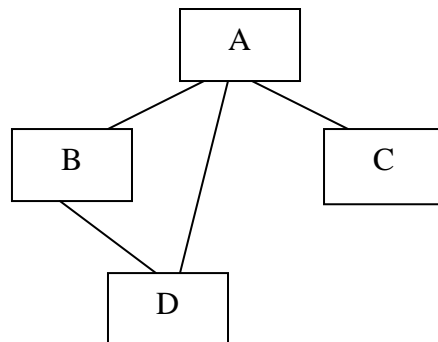
3.3. Notaciones para el diseño.

Para concretar y precisar las descripciones resultantes de todo el proceso de diseño se pueden usar numerosas notaciones. El objetivo de cualquier notación es resultar precisa, clara y sencilla de interpretar. Para ciertos aspectos del diseño se emplearán notaciones semejantes a las del análisis. Habrá distintas notaciones: Notaciones estructurales, estáticas, dinámicas e híbridas.

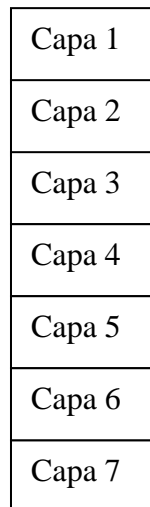
3.3.1. Notaciones estructurales.

Sirven para cubrir un primer nivel del diseño arquitectónico y con ellas se trata de desglosar y estructurar el sistema en sus partes fundamentales. Hay varias notaciones:

- Diagramas de bloques: Indica la conexión entre ellas.

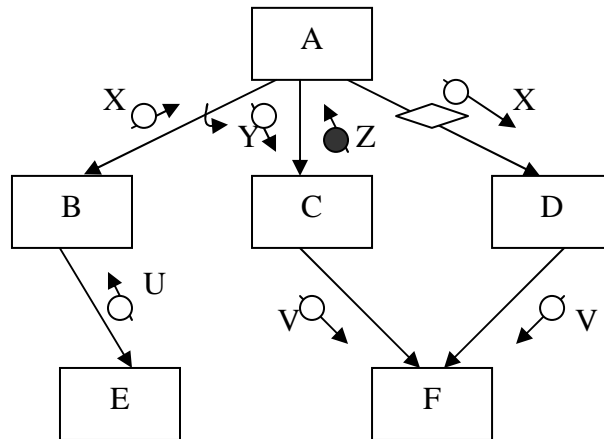


- Diagrama de cajas adosadas: Cada capa tiene una función distinta.



3.3.1.1. Diagrama de estructura.

Sirve para describir la estructura de los sistemas software como una jerarquía de subprogramas o módulos en general.



Siendo:

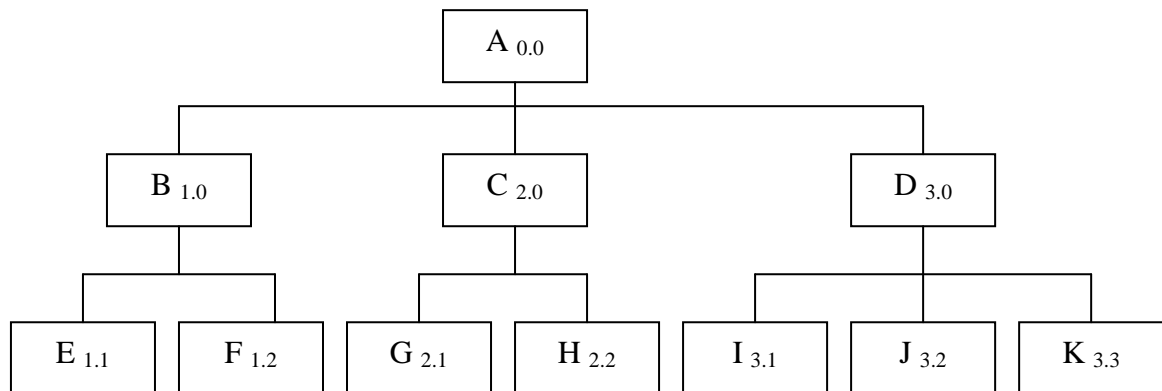
- Rectángulo: representa un módulo o subprograma cuyo nombre se indica en el interior.
- Línea: Une a dos rectángulos e indica que el módulo superior utiliza el inferior.
- Rombo: Indica que esa llamada o utilización es opcional.
- Arco: Indica que esa llamada o utilización se efectúa de manera repetitiva.
- Círculo con flecha: Representa el envío de los datos cuyo nombre acompaña al símbolo, desde un módulo a otro. Círculo cerrado si es información de control.

3.3.1.2. Diagrama HIPO.

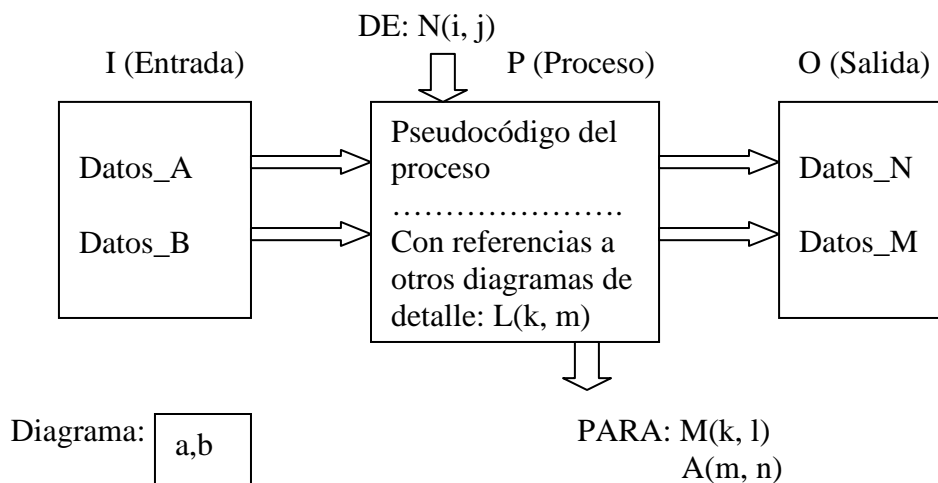
Sirven para facilitar y simplificar el diseño y desarrollo de sistemas software destinados fundamentalmente a gestión.

Hay varias notaciones:

- Diagrama HIPO de contenidos: Establece la jerarquía entre los módulos del sistema.



- Diagrama HIPO de detalle:



Título del programa: A

Siendo:

DE: $N(i, j)$: Indica el diagrama de detalle de nivel superior.

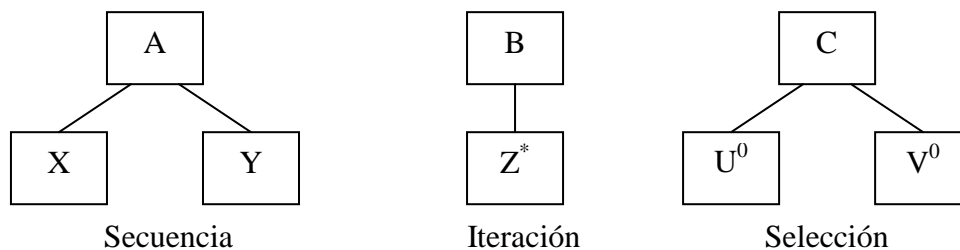
PARA: El del nivel inferior.

3.3.1.3. Diagrama de Jackson.

El proceso de diseño se lleva a cabo en tres pasos:

1°. Especificación de las estructuras de datos de entrada y salida.

2°. Obtención de una estructura del programa capaz de transformar las estructuras de datos de entrada en las de salida.



Siendo:

Secuencia: Colección de elementos de tipos diferentes combinados en un orden fijo (tupla).

Selección: Selección de un elemento entre varios posibles (unión).

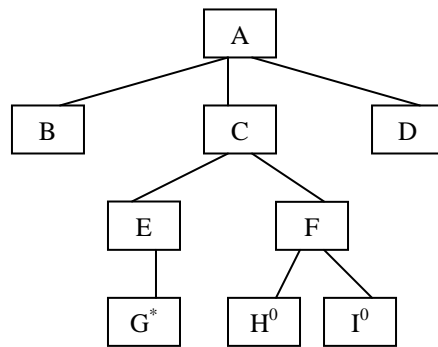
Iteración: Colección de elementos del mismo tipo (formación).

3°. Expansión de la estructura del programa para lograr el diseño detallado del sistema.

La metodología de Jackson está englobada dentro de las de Diseño Dirigido por los Datos, que ha sido utilizado fundamentalmente para diseñar sistemas relativamente pequeños de procesamiento de datos.

La notación de Jackson sirve para especificar la estructura de los datos como la del programa.

Un diagrama de Jackson y su estructura de programa equivalente es:



```

B;
IF Condición THEN
  FOR i:=1 TO N DO
    G
  END
ELSE
  H; I
END;
D;

```

Si los elementos del diagrama representan datos, la estructura equivalente, expresada en notación de diccionario de datos es:

$A = B + C + D$ (Secuencia)
 $C = [E \mid F]$ (Selección)
 $E = \{G\}$ (Iteración)
 $F = H + I$ (Secuencia)

3.3.2. Notaciones estáticas.

Sirvan para describir características estáticas del sistema, tales como la organización de la información sin tener en cuenta su evolución durante el funcionamiento del sistema. En el SRD esta información solo se realiza una propuesta de organización a grandes rasgos. En la fase de diseño se completa teniendo en cuenta aspectos internos: datos auxiliares, datos redundantes,... teniéndose un nivel de detalle mucho mayor. Las notaciones usadas serán las mismas que las empleadas en la especificación.

3.3.2.1. Diccionario de datos.

Se detalla la estructura interna de los datos que maneja el sistema. Para el diseño se partirá del diccionario definido del SRD y mediante refinamientos sucesivos se ampliará y se completará hasta alcanzar el nivel de detalle exigido para iniciar la codificación.

3.3.2.2. Diagrama Entidad-Relación.

Permite definir el modelo de datos, las relaciones entre los datos y, en general, la organización de la información. Se ampliará en la fase de diseño el documento SRD así como las nuevas entidades aparecidas en esta fase.

3.3.3. Notaciones dinámicas.

Permiten describir el comportamiento del sistema durante el funcionamiento. Se detallará su comportamiento externo y se añadirá la descripción de un comportamiento interno capaz de garantizar que se cumplan los requisitos especificados en el documento SRD. Las notaciones son:

3.3.3.1. Diagrama de flujo de datos.

Serán más detallados que en la especificación por describir CÓMO se hace y no solo QUÉ.

3.3.3.2. Diagrama de transición de estados.

Son los diagramas que existan en la especificación, preferiblemente sin ampliar.

3.3.3. Lenguaje de Descripción de Programas (PDL).

Se utiliza para realizar la especificación funcional como elaborar el diseño del mismo. Es equivalente a pseudocódigo.

Al ser más detallado este nivel que la especificación se requiere de ciertas estructuras de algún lenguaje de alto nivel. P.J. ADA.

3.3.4. Notaciones híbridas.

La abstracción es fundamental para la estructuración de sistemas complejos.

Hay una cierta similitud entre la programación basada en abstracciones y la orientada a objetos. Analizaremos la estructura de los tipos abstractos de datos y las clases de objetos.

La estructura de una abstracción y de un objeto es:

Nombre
Contenido
Operaciones

Nombre
Atributos
Métodos

Siendo:

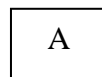
Nombre: Identificador de la abstracción.

Contenido: Elemento estático de la abstracción y en él se define la organización de los datos que constituyen la abstracción. En Modula-2 sería la información en el módulo de definición.

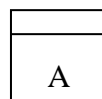
Operaciones: Elementos dinámicos de la abstracción y en él se agrupan todas las operaciones definidas para manejar el contenido de la abstracción. En Modula-2 sería la información en el módulo de implementación.

Un subprograma constituye una operación abstracta llamada abstracción funcional.

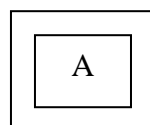
Hay distintas formas de abstracción:



Abstracción funcional



Tipo Abstracto de Datos



Datos encapsulado

Siendo:

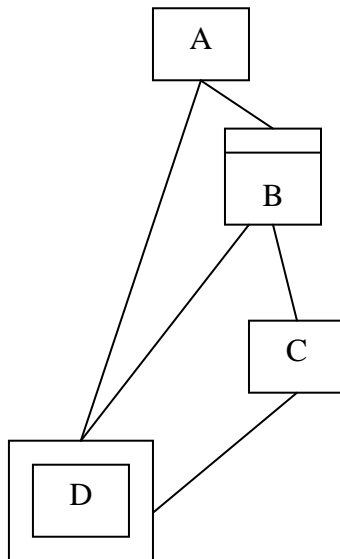
- Abstracción funcional: Está formado por funciones (expresiones parametrizadas) o procedimientos (acciones parametrizadas), que son las definiciones de subprogramas. Sólo está constituido por una operación y no tiene contenido.

- Tipo Abstracto de Datos: En un diseño se agrupan en una nueva entidad la estructura del tipo de datos con las operaciones necesarias para su manejo, que se llama tipo abstracto de datos. Tiene una parte de contenido y también sus correspondientes operaciones.

Permiten crear nuevos tipos de datos, además de los predefinidos en el lenguaje de implementación.

- Dato encapsulado: Cuando sólo 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 sin necesidad de indicarlo de manera implícita. Tiene contenido y operación pero no permite declarar variables de su mismo tipo.

Un ejemplo es el siguiente:



3.3.4.2. Diagrama de objetos.

Las abstracciones pueden ser una propuesta de los expertos en programación y los objetos son de los expertos en inteligencia artificial.

Las diferencias fundamentales entre ambos conceptos son:

1. No existen datos encapsulados ni abstracciones fundamentales cuando se usan objetos.
2. Sólo entre objetos se contempla una relación de herencia.

Entre los objetos se pueden establecer dos tipos de relaciones especiales:

A. Clasificación, especialización o herencia (No contemplada en abstracciones):

Como vimos antes en las herencias, los elementos “hijos” pueden adaptar las operaciones heredadas a sus necesidades concretas. No es necesario indicar la cardinalidad.

B. Composición (válida también entre abstracciones):

La relación de composición permite describir un objeto mediante los elementos que lo forman. Sólo se indica la cardinalidad en un sentido.

3.4. Documentos de diseño.

El resultado de la labor realizada en la etapa de diseño se recoge en un documento que se usará como elemento de partida para sucesivas etapas. Se llama SDD o Documento de Diseño de Software.

Se usarán dos documentos:

3.4.1. Documento ADD (Documento de Diseño Arquitectónico).

Describe el sistema en su conjunto. Contiene:

1. Introducción: Visión general del documento ADD.
2. Panorámica del sistema: Visión general de los requisitos funcionales del sistema.

3. Contexto del sistema: Indicara si este sistema posee conexiones con otros y si debe funcionar de una forma integrada con ellos.

4. Diseño del sistema.

4.1. Metodología de diseño de alto nivel.

4.2. Descomposición del sistema.

5. Diseño de los componentes: Contiene identificador del componente, tipo, objetivo, función, subordinado, dependencias, interfases, recursos referencias, proceso y datos.

6. Viabilidad y recursos estimados.

7. Matriz de requisitos/componentes.

3.4.2. Documento DDD (Documento de Diseño Detallado).

Ira creciendo con el desarrollo de proyecto. Contendrá los mismos puntos que el ADD separado en dos partes:

Parte 1. Descripción general.

1. Introducción.

2. Normas, convenios y procedimientos (la sección mas importante).

Parte 2. Especificaciones de diseño detallado.

Apéndice A. Listado fuente.

Apéndice B. Matriz de requisitos/componentes.

Tema 4. Técnicas generales de Diseño de Software.

El diseño de software es una actividad que requiere cierta experiencia previa, que es difícil de adquirir sólo a través del estudio de las técnicas de diseño. Se hará repaso a las técnicas más importantes de diseño. Los objetivos que tratan de conseguir las técnicas serán:

- a. La descomposición modular del sistema.
- b. La decisión sobre los aspectos de implementación o representación de datos que son importantes a nivel global.

Se estudian las técnicas de diseño agrupadas en diseño funcional descendente, diseño orientado a objetos y diseño de datos.

4.1. Descomposición modular.

Todas las técnicas de diseño están de acuerdo en la necesidad de realizar una descomposición modular, que se concreta en los siguientes aspectos:

- Identificar de módulos.
- Describir cada módulo.
- Describir las relaciones entre módulos.

La diferencia fundamental entre las distintas técnicas de diseño es lo que se entiende por módulo en cada una de ellas.

Un módulo es un fragmento de un sistema software que se puede elaborar con relativa independencia de los demás. Habrá los siguientes tipos:

- Código fuente: Contiene texto fuente escrito en el lenguaje de programación elegido.
- Tabla de datos: Se usa para tabular ciertos datos de inicialización, experimentales o de difícil obtención.
- Configuración: Un sistema se puede concebir para trabajar en entornos diversos según las necesidades de cada cliente.
- Otros: Puede tener otras utilidades los módulos. P.J. Uso de “make” en UNIX.

El formato concreto de cada tipo de módulo depende de la técnica, metodología o herramientas utilizadas.

Es difícil establecer un patrón de medida capaz de indicar de una manera precisa y cuantitativa lo buena o mala que es una descomposición para facilitar su mantenimiento. Sin embargo, la descomposición tendrá unas cualidades mínimas que trataremos ahora.

4.1.1. Independencia funcional.

En el documento ADD y DDD en la matriz requisitos/componentes es necesario indicar qué componente se encargará de realizar cada uno de los requisitos indicados en SRD. Los pasos para la descomposición serán:

1º paso. Cada función en módulo distinto, comprobando si son independientes funcionalmente.

2º paso. Se realizará el refinamiento del diseño en distintos módulos, usando el concepto de abstracción, ocultación, etc.

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

Se utilizarán dos criterios: acoplamiento y cohesión.

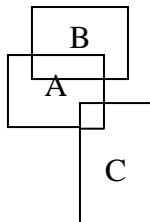
4.1.1.1. Acoplamiento.

El grado de acoplamiento entre módulos es una medida de interrelación que existe entre ellos: tipo de conexión y complejidad de la interfase. Se usa esta escala:

- Acoplamiento por Contenido. } Fuerte
- Acoplamiento Común. }
- Acoplamiento Externo. }
- Acoplamiento de Control. } Moderado
- Acoplamiento de Etiqueta. }
- Acoplamiento de Datos. } Débil.
- Sin Acoplamiento Directo. }

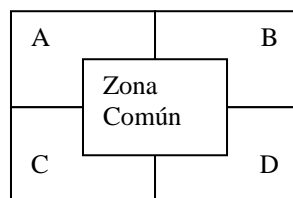
El objetivo es que durante el diseño se tenga en cuenta la escala para buscar descomposiciones con acoplamiento débil o moderado, evitándose cuando se pueda el fuerte.

El acoplamiento por contenido se produce cuando desde un módulo se puede cambiar los datos locales e incluso el código de otro módulo. Un ejemplo:



No existe una separación real entre módulos y hay solapes entre ellos. Sólo se logra utilizando un lenguaje ensamblador o de muy bajo nivel evitándose cuando se pueda, por ser imposible de entender y depurar.

Para el acoplamiento común se emplea una zona común de datos a los que tiene acceso varios o todos los módulos del sistema.

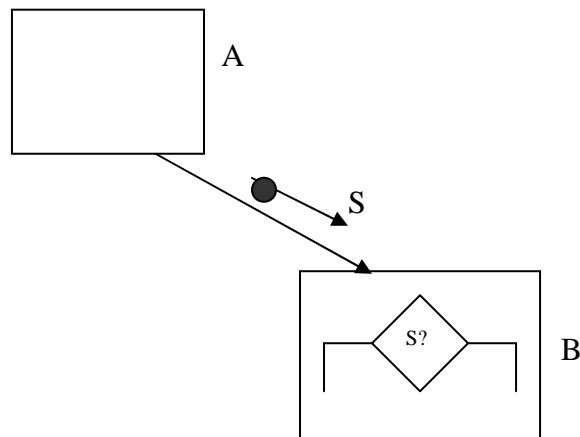


Esto significa que cada módulo puede estructurar y manejar la zona común con total libertad sin tener en cuenta el resto de módulos.

Los datos cambiados en un módulo afectan al resto que deberían ser modificados en la nueva estructura. Lo usa Fortran. Es muy difícil de mantener y depurar.

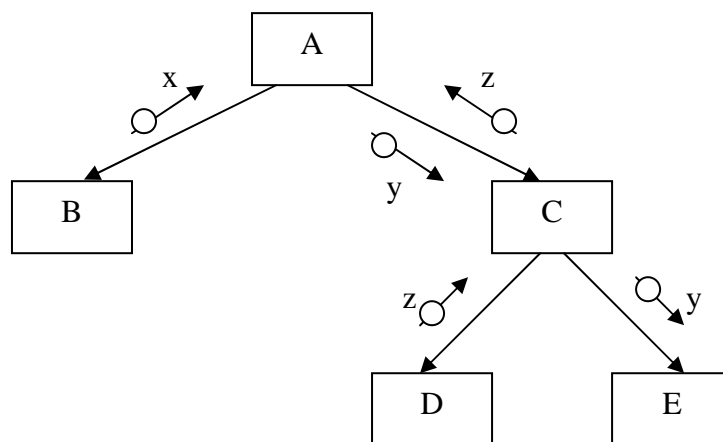
En el acoplamiento de externo la zona común esta constituida por un dispositivo externo (disco, sensor, canal de comunicaciones, etc.) al que están ligados todos los módulos. El formato de la zona común lo impondrá el formato de los datos.

En el acoplamiento de control una señal (S) o dato de control que se pasa desde un módulo (A) a uno (B) es lo que determina la línea de ejecución que se debe seguir dentro de este último (B). Como se ve en la figura.



El acoplamiento se produce sólo a través del intercambio de aquellos datos que un módulo necesita de otro. Si el acoplamiento se realiza sólo con los datos que se usan se llama acoplamiento de datos.

El acoplamiento por etiqueta será aquel en el que se referencia a los datos y a la estructura completa de la que forman parte (vector, pila, árbol, etc.).



Cualquier modificación en el módulo afecta nada o poco al resto. Son los mejores acoplamientos, pero es mejor todavía cuando no hay ningún acoplamiento directo, como en los módulos B y E del ejemplo.

4.1.1.2. Cohesión.

Es complementario al criterio del acoplamiento. Significa la relación de elementos dentro de un módulo sin tener relación (acoplamiento) con los otros módulos. Se utiliza la siguiente escala:

- Cohesión abstraccional. } Alta
- Cohesión funcional. }

- Cohesión secuencial. } Media
- Cohesión de comunicación. }

- Cohesión temporal. } Baja
- Cohesión lógica. }
- Cohesión coincidental. }

La cohesión coincidental es la peor posible y se produce cuando no hay ninguna relación entre los elementos del módulo. Significa que no se ha realizado ninguna labor de diseño.

La cohesión lógica se produce cuando se agrupan en un mismo módulo elementos que realizan funciones similares desde un punto de vista de usuario. PJ. Módulo de funciones matemáticas, seno, coseno.

La cohesión temporal es el resultado de agrupar en un mismo módulo aquellos elementos que se ejecutarían en un mismo momento. PJ. Al iniciar o finalizar el sistema se deben arrancar o parar los dispositivos, como teclado, ratón, etc.

La cohesión baja se debe evitar siempre salvo en los ejemplos antes citados.

Por cohesión de comunicaciones se entiende aquella que se produce cuando todos los elementos del módulo operan con el mismo conjunto de datos de entrada o producen el mismo conjunto de datos de salida. PJ. Cuando un módulo realiza operaciones diversas pero con la misma tabla de datos.

La cohesión secuencial se produce cuando todos los elementos del módulo trabajan de forma secuencial. PJ. La salida de un elemento es la entrada del siguiente de manera sucesiva.

Con una cohesión media se puede reducir el número de módulos, pero no se deberá modificar el grado de acoplamiento entre módulos.

En la cohesión funcional cada elemento está encargado de la realización de una función concreta y específica.

La cohesión abstraccional se logra cuando se diseña un módulo como TAD o como una clase de objetos.

La cohesión alta debe ser el objetivo a perseguir en cualquier descomposición modular.

Se puede establecer el grado de cohesión de acuerdo a estos criterios:

- A. Si la descripción es una frase con comas o más de un verbo tendrá cohesión media.
- B. Si la descripción contiene palabras relacionadas con el tiempo, tales como “primero”, “después”, la cohesión es temporal o secuencial.
- C. Si la frase de la descripción no se refiere a algo específico a continuación del verbo, como “escribir todos los mensajes de error” es una cohesión lógica.
- D. Cuando se usan verbos como “inicializar”, la cohesión es temporal.

La descripción modular con mayor independencia funcional se logra con un acoplamiento débil y una cohesión alta dentro de cada uno de ellos.

4.1.2. Comprensibilidad.

Para intentar comprender el sistema tras los cambios hechos se tendrá que hacer un gran esfuerzo. Estos cambios se realizarán en las fases de diseño e implementación y generalmente por personas que no han participado en ambas fases.

El primer factor de comprensión es su independencia funcional como vimos antes. Al no ser suficiente se atenderá a los siguientes factores:

1. Identificación: Elección adecuada de nombre de módulos y de los elementos en el módulo.
2. Documentación: La documentación de cada módulo y del sistema facilita la comprensión, aclarando aspectos del diseño e implementación.
3. Simplicidad: Las soluciones sencillas son siempre las mejores.

4.1.3. Adaptabilidad.

Es difícil tratar de adaptar el sistema, ya que el cliente siempre tratará de hacerlo conforme a sus objetivos. Dos cualidades esenciales para posibilitar su adaptabilidad es la independencia funcional y comprensibilidad, pero hay otros factores que la facilitan:

1. Previsión: resulta complicado prever la evolución futura de un determinado sistema, pero la experiencia nos indicará que partes de sistemas semejantes han sido más susceptibles a cambios o adaptaciones.
2. Accesibilidad: Para poder adaptar un sistema es necesario conocer la estructura global y todos sus detalles significativos. Será posible si los documentos de especificación, diseño e implementación son sensibles de acceder, usando herramientas CASE o similares,
3. Consistencia: Se tiende a no actualizar los documentos cuando se cambia el código fuente, lo que da lugar a inconsistencias. Se actualizará los documentos de especificación, diseño e implementación. Las herramientas a usar serán las de “control de versiones y configuración”.

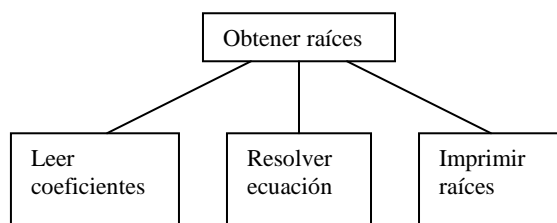
4.2. Técnicas de diseño funcional descendente.

Se incluyen las técnicas en que la descomposición del sistema se hace desde el punto de vista funcional, es decir, se atiende a la función que ha de realizar el sistema, expresándose mediante funciones más sencillas, que se encomiendan a módulos separados. Los lenguajes a usar serán Fortran o Cobol.

4.2.1. Desarrollo de refinamiento progresivo.

Esta técnica corresponde a la aplicación en la fase de diseño de la metodología conocida como programación estructurada y que condujo a la construcción de programas mediante refinamientos sucesivos. La programación estructurada recomienda emplear en la construcción de programas estructuras claras y sencillas.

La construcción de programas basados en el concepto de refinamiento consiste en plantear inicialmente el programa como una operación única global e ir descomponiendo en otras funciones más sencillas. PJ.



Obtener raíces →

Leer coeficientes

Resolver ecuación →

Calcular discriminante

Calcular raíces →

SI el discriminante es negativo

ENTONCES

Calcular raíces complejas

SI-NO

Calcular raíces reales

FIN-SI

Imprimir raíces

La aplicación de esta técnica a la fase de diseño consiste en realizar sólo los primeros niveles de refinamiento, asignando a módulos separados las operaciones parciales que se van identificando. El paso de la estructura de refinamiento a la modular es inmediato.

4.2.2. Programación Estructurada de Jackson (JSP).

Esta técnica sigue estrictamente las ideas de programación estructurada en cuanto a las estructuras recomendadas (secuencia, iteración y selección) y el método de refinamientos sucesivos para construir la estructura del programa en forma descendente. La aportación principal de esta técnica respecto a la metodología general de la programación estructurada está en las recomendaciones para ir construyendo la estructura del programa que se debe hacer similar a las estructuras de los datos de entrada y salida.

Esta técnica se basa en los siguientes pasos:

1. Analizar el entorno del problema y describir las estructuras de los datos a procesar (diseño).
2. Construir la estructura del programa basado en las estructuras de los datos (diseño).
3. Definir las tareas a realizar en términos de las operaciones elementales disponibles y situarlas en los módulos apropiados de la estructura del programa (fase de codificación).

Ejemplo de uso en esta técnica:

Queremos justificar un texto, es decir, agrupar las palabras en líneas e intercalar los espacios separados apropiados para que las líneas se ajusten a los márgenes establecidos (El texto viene en el libro Pág. 163). Seguiremos los siguientes pasos:

1º paso. Averiguaremos las estructuras de los datos de entrada y salida.

Texto de entrada = {[separador de párrafo | palabra]}: Blancos del texto de entrada.

Texto de salida = {[línea ajustada | línea final | línea en blanco]}, siendo:

[]: Selección y { }: Iteración.

2º paso. Buscamos unidad superior de información, que será en este caso el párrafo, que podría adaptarse a las unidades de datos de E/S como sigue:

Texto = {párrafo}.

Párrafo de entrada = separador de párrafo + {palabra}

Párrafo de salida = {línea en blanco} + {línea ajustada} + línea final.

4.2.3. Diseño estructurado.

Esta técnica es el complemento del análisis estructurado y coinciden en el uso de DFD, como medio fundamental de representación del modelo funcional del sistema. La tarea de diseño consiste en pasar de las DFD a diagramas de estructura, prescindiendo de los almacenes de datos.

Para establecer una jerarquía en los DFD, esta técnica recomienda hacer análisis de flujo de datos global. Se recomienda el análisis de flujo de transformación y 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 tres regiones: flujo de entrada, flujo de transformación y de salida. Se añade un módulo de coordinación que realiza el control de acuerdo con la distribución del flujo de transformación.

- Análisis del flujo de transacción: Es aplicable cuando el flujo de datos se puede descomponer en varias líneas separadas, cada una de las cuales corresponde a una función global y transacción externa, de manera que sólo una de estas líneas se activa para cada entrada de datos de tipo diferente. Identificaremos un centro de transacciones por el que se ramifican las líneas de flujo.

4.3. Técnicas de diseño basado en abstracciones.

Surgen al identificarse los conceptos de abstracción de datos y de ocultación. La idea es que los módulos se corresponden o con funciones o con TAD. Las estructuras modulares resultantes pueden implementarse con lenguajes de programación como Ada o Modula-2, así como lenguaje de programación orientado a objetos.

4.3.1. Descomposición modular basada en abstracciones.

Consiste en dedicar módulos separados a la realización de cada TAD y cada función importante. Puede aplicarse de dos formas:

- Descendente: Se considera como una ampliación de la técnica de refinamiento progresivo, en que se plantea como alternativa de su descomposición, el que la operación a refinar se defina como abstracción funcional o sobre un TAD.
- Ascendente: Se trata de ir ampliando las primitivas existentes en el lenguaje de programación y las liberaran asociadas con nuevas operaciones y tipo de mayor nivel.

4.3.2. Método de Abott.

En el método anterior no hay ninguna guía para reconocer los elementos candidatos para ser considerados como abstracciones sean funciones o TADs. Con este método se consigue esto a partir de las descripciones o especificaciones del sistema hechas en lenguaje natural.

Se identifican los elementos significativos del diseño: tipo de datos (sustantivos genéricos), atributos (sustantivos) y operaciones (verbos).

Se siguen estos pasos:

- 1º. Subrayado del texto para sacar los elementos significativos.
- 2º. Reorganizar las listas extrayendo los posibles tipos de datos y asociándole atributos y operaciones. Se eliminará términos irrelevantes o sinónimos.

4.4. Técnicas de diseño orientadas a objetos.

Es igual al diseño basado en abstracciones, pero incluye características adicionales como la herencia y el polimorfismo.

La idea de estas técnicas es que en la descomposición modular del sistema cada módulo contenga la descripción de cada clase de objetos o varios relacionados entre si.

4.4.1. Diseño orientado a objetos.

Es una técnica general que deriva el diseño a partir de las especificaciones del sistema.

Se basa en los siguientes:

1. Estudiar y comprender el problema a resolver.
2. Desarrollar la posible solución.
3. Formalizar la estrategia. Sigue estas etapas:
 - a) Identificar las clases y objetos, sus atributos o componentes.
 - b) Identificar las operaciones sobre objetos.
 - c) Aplicar herencia.
 - d) describir las operaciones.
 - e) Establecer la estructura modular.

4.5. Técnicas de diseño de datos.

Estas técnicas sirven de apoyo a la ingeniería del software. Se podrá enfocar desde el punto de vista de varios niveles:

- Nivel externo: Corresponde a la versión del usuario. P.J. Ficha de cliente o empleado.
- Nivel conceptual: Establece una organización lógica de los datos, con independencia del sentido físico que tengan en el campo de las aplicaciones. Se resume en diagramas E-R o diagrama de Modelo de Objetos (diccionario de datos).
- Nivel físico: Organiza los datos según los esquemas admisibles en el SGBD y/o lenguaje de programación elegido para el desarrollo.

4.6. Diseño de base de datos relacionales.

Partiendo del modelo E-R o modelo de objetos es posible dar reglas prácticas para obtener los esquemas de las tablas de base de datos relacional que reflejen la visión lógica de los datos y que sean eficientes. Hay dos puntos de vista para esta eficiencia: Formas normales y empleo de índices.

4.6.1. Formas normales.

Definen criterios para establecer esquemas de tablas que sean claros y no redundantes. Se enumeran de menor (1º) a mayor nivel de restricción (2º, 3º, etc.).

1ª forma normal: Cuando la información de la tabla tiene valor único en cada una de las columnas.

2ª forma normal: Si está la 1ª forma normal y tiene una clave primaria que distingue cada fila.

3ª forma normal: Si satisface el criterio de la 2ª y no hay dependencias entre columnas que no son clave primaria.

4.6.2. Diseño de las entidades.

Se diseñará una tabla por identidad.

4.6.3. Tratamiento de las relaciones de asociación.

Hay dos tipos especiales de relaciones de composición (agregación) y herencia (especialización), siendo el resto relaciones de asociación.

La manera de almacenar en tablas las relaciones de asociación depende de la cardinalidad:

- Cardinalidad N-N: Referente a las entidades relacionadas se hará mediante la clave primaria de cada una.
- Cardinalidad 1-N: Se puede incluir los datos de la relación en la misma tabla de las entidades relacionadas.
- Cardinalidad 1-1: Las dos tablas de las entidades se puede fundir en uno.

4.6.4. Tratamiento de las relaciones de composición.

Se tratan igual que las relaciones de asociación.

4.6.5. Tratamiento de herencia.

Cuando una clase de objetos (entidad genérica) tiene varias subclases (entidades específicas) se pueden adoptar tres formas de almacenar tablas en información de las entidades:

- a. Tablas de superclase y de subclases.
- b. Sólo tabla de subclases.
- c. Sólo tabla de superclase con atributos de las de subclase.

4.6.6. Diseño de índices.

Conviene mantener un índice sobre las claves primarias y columnas de referencia de las entidades relacionadas.

4.7. Diseño de base de datos de objetos.

Hay mas estructuras que en la base de datos relacionales (sólo se maneja tabla).

Hay dos enfoques en el diseño físico con estas bases de datos:

- cuando hay gran cantidad de estructuras. El diseño se puede hacer como para las estructuras de datos en memoria.
- Cuando no hay gran variedad de estructuras de datos. Es análogo al de una base de datos relacional.

Tema 5. Codificación y pruebas.

En este tema se ve las últimas fases del ciclo de vida: codificación, pruebas de unidad, fase de integración y pruebas del sistema.

5.1. Codificación del diseño.

Constituye el núcleo central de cualquiera de los modelos de desarrollo de software.

En esta fase se elaboran los programas fuente. Un elemento esencial de la codificación es el lenguaje de programación que se emplea. Se establecerán normas y estilo de codificación para la homogeneidad de la codificación.

Cuando los resultados de las pruebas no sean satisfactorios habrá que realizar cambios en la codificación.

5.2. Lenguajes de programación.

El conocimiento de las prestaciones de los lenguajes permite aprovechar mejor sus posibilidades y salvar sus posibles deficiencias.

5.3. Desarrollo histórico.

Para tener una visión panorámica de los lenguajes de programación habrá que estudiar su evolución.

5.3.1. 1ª generación (años 50).

Los programas a ejecutar eran pequeños y se hacía acceso directo a memoria. P.J. Ensamblador. Actualmente no se usa.

5.3.2. 2ª generación (finales años 50 y principios 60).

El aumento de las capacidades de memoria y disco de los computadores hizo posible abordar programas más grandes y complejos. Se empezaron a desarrollar lenguajes de alto nivel. Se incorporan elementos realmente abstractos. Algunos lenguajes son Fortran, Cobol, Algol y Basic.

5.3.3. 3ª generación (finales años 60 y principios 70).

Pertenece gran parte de los lenguajes que se usan actualmente para el desarrollo de software. Se consolidan las fases teóricas y prácticas de la programación estructurada. Alguno de los lenguajes de programación estructurada son: Pascal, Modula-2, C, Ada.

Los lenguajes asociados a otros paradigmas de programación o modelos abstractos de cómputo: Orientación a objetos (Smalltalk, C++, Eiffel), funcional (lisp) o lógico (prolog).

5.3.4. 4ª generación (actualmente).

Tratan de ofrecer un mayor nivel de abstracción, prescindiendo por completo del computador.

Una posible agrupación según su aplicación concreta sería: Base de datos, generadores de programas, cálculo y otros.

5.4. Prestaciones de los lenguajes.

5.4.1. Estructura de control.

Nos referiremos al conjunto de sentencias o instrucciones de los lenguajes que encuadra dentro de la parte ejecutiva de un programa.

5.4.1.1. Programación estructurada.

Los lenguajes imperativos disponen de sentencias que facilitan la programación estructurada, como secuencia, selección e iteración. Las sentencias para la selección son la selección general y la selección por casos y para la iteración es repetición, bucle con contador y bucle indefinido.

Permite el uso de procedimientos o funciones. Se definen de forma recursiva.

5.4.1.2. Manejo de excepciones.

Las excepciones son los errores o sucesos inusuales que ocurren al ejecutar un programa. Los orígenes son errores humanos, fallos hardware, errores software. Las excepciones como situaciones no erróneas pero poco frecuentes son: datos de entrada vacíos, valores fuera de rango, etc. Hay lenguajes que manejan las excepciones.

5.4.1.3. Concurrencia.

Algunos aspectos para su diseño y programación son: tareas concurrentes, sincronización de tareas, comunicación entre tareas e interbloqueos. Las formas de abordar la concurrencia son: corrutinas, fork-join, cobegin-coend y procesos.

Las estructuras en los lenguajes son:

- Variables compartidas: Semáforos, regiones críticas condicionales, monitores.
- Paso de mensajes: Proceso de comunicación secuencial (CSD), llamada a procedimientos remotos y “Redezvous”.

5.4.2. Estructura de datos.

Nos referiremos a las distintas formas que usan los lenguajes para estructurar de datos que manejan.

5.4.2.1. Datos simples.

Pueden manejar enteros pero teniendo en cuenta su rango, como ristra de caracteres, tipo enumerado (lógico), tipo subrango.

5.4.2.2. Datos compuestos.

Arrays registros, tipo conjunto (TYPE Mezcla = SET OF Color;).

5.4.2.3. Constantes.

Constantes literales (número o caracteres), constantes simbólicas (nombres), constantes de tipo enumerado o conjunto además de tipos predefinidos (entero, real, carácter o booleano).

5.4.2.4. Comprobación de tipos.

Hay 5 niveles de comprobación de tipos en los lenguajes de programación:

- Nivel 0 (sin tipos): No se pueden declarar nuevos tipos de datos y todos los datos son predefinidos. PJ. Basic, Cobol, etc.
- Nivel 1 (tipado automático): El compilador decide cual es el tipo mas adecuado. PJ PL/1.
- Nivel 2 (tipado débil): conversión automática entre datos con ciertas similitudes. PJ Fortran.
- Nivel 3 (tipado semirrigido): Todos los datos deben ser declarados con sus correspondientes tipos. PJ. Pascal.
- Nivel 4 (tipado fuerte): Cualquier conversión al tipo debe ser explícita. PJ. Ada y Modula-2.

5.4.3. Abstracciones y objetos.

Hay tres niveles distintos.

5.4.3.1. Abstracciones funcionales.

En los lenguajes permanece oculta la codificación de la operación en módulos de implementación (modula-2).

5.4.3.2. Tipo Abstracto de Datos.

Deben agrupar el contenido o atributos de los datos y las operaciones definidas para el manejo del contenido. Deben existir mecanismos de ocultación. Usa módulos de implementación.

5.4.3.3. Objetos (máquinas abstractas).

Son similares a los TAD pero incluyen el polimorfismo y la herencia.

5.4.3.4. Modularidad.

Este concepto está ligado a la división del trabajo y al desarrollo en equipo de un proyecto software. Las cualidades son:

- Compilación separada: Permite preparar y compilar separadamente el código de cada módulo. Al ser consistente el uso de un elemento con su definición quiere decir que la compilación es segura, es decir, coincidir el uso de los elementos del módulo de implementación y definición. Sólo será visible el módulo de definición.

5.5. Criterios de selección del lenguaje.

Algunos de los criterios serian:

- Imposición del cliente: Escoge el que quiera.
- Tipo de aplicación.
- Disponibilidad y entorno: Comprobar qué compiladores existen para el computador elegido, etc.
- Experiencia previa.
- Reusabilidad.
- Transportabilidad.
- Uso de varios lenguajes.

5.6. Aspectos metodológicos.

Se repasan ciertos aspectos metodológicos que pueden mejorar la codificación: claridad, manejo de errores, eficiencia y transportabilidad.

5.6.1. Normas y estilos de codificación (claridad).

Se fijaran las normas a todos los trabajadores. Se concretaran estos puntos de estilo: formato y contenido de las cabeceras de cada módulo, formato y contenido para cada tipo de comentario, utilización de encolumnado, elección de nombres.

Se podrán incluir las restricciones para mejorar la claridad del código como el tamaño máximo de las subrutinas no debe superar P paginas, etc. y el empleo de algunas sentencias del lenguaje.

5.6.2. Manejo de errores.

Las causas de los errores pueden ser:

- Introducción de datos incorrectos o inesperados.
- Anomalías en el hardware.
- Defectos en el software.

Se tendrá en cuenta los conceptos:

- Defecto: Errata del software.
- Fallo: Un elemento que no funciona correctamente.
- Error: Estado inadmisibile de un programa por un fallo.

Por Piattini:

- Defecto (defect, fault, “bug”): defecto en el software como un proceso, una definición de dato o un paso de procesamiento incorrecto en un programa (baja calidad del software).
- Fallo: La incapacidad de un sistema o de alguno de sus componentes para realizar las funciones requeridas dentro de los requisitos de rendimiento especificados (fiabilidad del software).
- Error: El error del programador. $PJ. 2+2=5$.

Hay distintas actitudes respecto al tratamiento de los errores:

- No considerar los errores.
- Prevención de errores: Técnica de programación a la defensiva.

- Recuperación de errores: Se puede hacer un tratamiento del error con el objetivo de restaurar el programa en un estado correcto y evitar que el error se propague.

Exige dos actividades: detección de errores y recuperación del error.

Para la recuperación de errores existen dos esquemas:

- Recuperación hacia delante: Trata de identificar la naturaleza o el tipo de error para luego tomar las acciones para corregir el estado del programa y seguir ejecutándolo.
- Recuperación hacia atrás: Trata de corregir el estado del programa restaurándolo a un estado correcto anterior a la aparición del error. Se usa en sistemas basados en transacciones (usando commit).

5.6.3. Aspectos de eficiencia.

Se puede analizar desde dos puntos de vista:

- Eficiencia en memoria: Importante en sistemas que emplean gran cantidad de información.
- Eficiencia en tiempo: Importante en sistemas de tiempo real.

5.6.4. Transportabilidad de software.

Permite usar el mismo software en distintos computadores actuales y futuros.

Los factores esenciales son:

- Utilización de estándares: Para ser transportable sin cambios.
- Aislar las peculiaridades: Aislar un módulo específico para cada uno de ellos. Las propiedades de los computadores serán: arquitectura del computador, sistema operativo.

5.7. Técnicas de prueba de unidades.

El principal objetivo de las pruebas debe ser conseguir que el programa funcione correctamente y que se descubran los defectos. Para elaborar los casos de prueba se debe tener en cuenta:

- Una buena prueba es la que encuentra errores y no los encubre.
- Para detectar un error es necesario conocer cual es el resultado correcto.
- Es bueno que no participen en la prueba el codificador o diseñador.
- Siempre hay errores y si no aparecen se deben diseñar pruebas mejores.
- Al corregir un error se puede introducir otros nuevos.
- Es imposible demostrar la ausencia de defectos mediante pruebas.

Sólo se explora una parte de las posibilidades del programa. Para unos resultados fiables se debe realizar el proceso de prueba de manera automática creando un entorno adecuado de trabajo.

Hay dos técnicas de prueba de unidades: Prueba de caja negra (black box) o de caja blanca o transparente (white box).

5.7.2. Pruebas de caja negra (prueba funcional según Piattini).

Ignora la estructura interna del programa y se basa en la comprobación de la especificación de entrada y salida del software. Se trata de verificar que todos los requisitos del programa se cumplen. Se intentará detectar los casos correctos como los sospechosos (insiste mas en este punto Piattini). Los metodos mas usados son:

- Partición de clases de equivalencia: Se trata de dividir el espacio de ejecución del programa en varios subespacios. Los pasos a seguir son: determinar las clases equivalentes apropiadas y establecer pruebas para cada clase, proponiendo casos válidos y no válidos.
- Análisis de Valores Limites (AVL): Los errores tienen tendencia a aparecer en las fronteras o valores limites de los datos normales. Se proponen casos válidos y no válidos. Las directrices en la elaboración de casos de pruebas serán: entradas, salidas, memoria, recursos y otros.

- Comparación de versiones: Al cambiar la versión del software se suele actualizar otro juego de pruebas. Se elaborará un nuevo juego de pruebas y luego se someterán ambas versiones hasta que coincidan los resultados de las versiones con el original.

- Empleo de la intuición.

5.7.3. Pruebas de caja transparente (pruebas estructurales).

Se tiene en cuenta la estructura interna del módulo. Se trata de conseguir que el programa transite todos los posibles caminos de ejecución. Los casos de prueba deben conseguir:

- Todas las decisiones se ejecutan en uno y otro sentido.

- Todos los bucles se ejecutan en los supuestos más diversos posibles.

- Todas las estructuras de datos se modifiquen y consulten alguna vez.

No se pueden recorrer todos los posibles casos. Las pruebas de caja negra y blanca son complementarias y nunca excluyentes. Los métodos son:

- Cubrimiento lógico (complejidad ciclomática de McCabe en Piattini): Consiste en cubrir todas las secciones de código al menos una vez. Llamaremos camino básico a cualquier recorrido que siguiendo las flechas de las líneas de flujo nos permiten ir desde punto inicial al final.

Primero, determinamos el número de caminos básicos, siendo igual al número de predicados + 1.

Se pueden establecer distintos niveles de cubrimiento:

Nivel I. Se elabora casos de prueba que se ejecuten una vez todos los caminos básicos.

Nivel II. Se elabora casos de prueba que se ejecuten todas las combinaciones de caminos básicos por parejas.

Nivel III. Se elabora casos de prueba para que se ejecuten un número significativo de las combinaciones posibles de caminos.

- Pruebas de bucles:

1. Bucle con número no acotado de repeticiones.

2. Bucle con número máximo (M) de repeticiones.

3. Bucles anidados: El número de pruebas crece de forma geométrica con el nivel de anidamiento.

4. Bucles concatenados: Si son independientes se probarán cada uno por separado.

- Empleo de la intuición.

5.7.4. Estimación de errores no detectados.

Es una estimación estadística de los errores no detectados por el juego de pruebas. La estrategia para buscar una estimación del número de defectos sin detectar son:

a. Anotar el número de errores iniciales con el juego de casos de prueba: E_1 .

b. Corregir el módulo hasta que no tenga ningún error con el mismo juego de casos de prueba.

c. Introducir aleatoriamente en el módulo un número razonable de errores: E_A

d. Someter el módulo con los nuevos errores al juego de casos de prueba y hacer de nuevo el recuento del número de errores que se detectan: E_D .

e. El número estimado sin detectar será:

$$E_E = (E_A - E_D) * (E_1 / E_D).$$

5.8. Estrategias de integración.

Los módulos o unidades se han de integrar para conformar el sistema completo. Se añadirán errores tales como desacuerdos de la interfaz, interacción indebida entre módulos, etc.

Las estrategias básicas de integración son:

5.8.1. Integración Big Bang.

Consiste en realizar la integración de todas las unidades en un único paso. Provoca muchos errores. Evita la realización del software de “andamiaje” lo cual es una ventaja.

5.8.2. Integración descendente.

Se parte del módulo principal que se prueba con módulos de “andamiaje” o sustitutos por los otros módulos usados directamente. Estos módulos se reemplazan, uno por uno, por los verdaderos y se realizan las pruebas de integración correspondientes.

Para la codificación de los sustitutos se emplean estas soluciones:

- No hacer nada y que sólo sirva para comprobar la interfaz.
- Implementar un mensaje de seguimiento con información de la llamada.
- Suministrar un resultado fijo, aleatorio o tabulado (obtenido con un algoritmo simplificado).

La ventaja es que se ven desde el principio las posibilidades de la aplicación.

Las desventajas son:

1. Limita el trabajo en paralelo.
2. Habrá muchas limitaciones al integrar nuevos módulos a partir de otros ya integrados y definitivos.

5.8.3. Integración ascendente.

Se empieza por codificar por separado y en paralelo todos los módulos de nivel mas bajo. Para probarlos se escriben módulos gestores o conductores que las hacen funcionar independientemente o en combinaciones sencillas. Los gestores se van sustituyendo por módulos de mayor nivel.

Las ventajas de está integración son los inconvenientes de la integración descendente.

Las desventajas es que resulta difícil probar el funcionamiento global hasta el final de su integración.

La solución es utilizar una integración ascendente con los módulos de nivel mas bajo y una descendente con los del nivel mas alto. Se denomina integración sándwich.

5.9. Pruebas del sistema.

Se prueba el sistema completo.

5.9.1. Objetivos de las pruebas.

Hay distintas clases de pruebas:

- De recuperación: Comprobar la capacidad del sistema para recuperarse ante fallos.
- De seguridad: Comprobar los mecanismos de protección contra un acceso o manipulación no autorizada.
- De resistencia: Comprobar el comportamiento del sistema ante situaciones excepcionales.
- De sensibilidad: Comprobar el tratamiento que da el sistema a ciertas singularidades relacionadas con los algoritmos matemáticos que se usan.
- De rendimiento: Comprobar las prestaciones del sistema que son críticas en tiempo.

5.9.2. Pruebas alfa y beta.

Son pruebas con el apoyo del usuario final.

Las pruebas alfa son unas primeras pruebas que se realizan en un entorno controlado donde el usuario tiene el apoyo de alguna persona del equipo de desarrollo.

Las pruebas beta son las hechas por los usuarios sin apoyo de nadie, en su entorno normal. Al equipo de desarrollo le transmitirá el procedimiento que le ha llevado al error al usuario final.