TEMA 1: INTRODUCCIÓN

1.1 ¿QUÉ ES EL SOFTWARE?

El software es todo lo que no es hardware e incluye los programas que gobiernan el funcionamiento del sistema y otros elementos tales como documentos o bases de datos.
El software puede ser un producto que se venda como un procesador de textos, o solo una parte de un producto más complejo, o puede ser el medio para dar un servicio.

1.1.1 Calidad del software

Existe un esquema general de mediciones de la calidad del software propuesto por McCall y otros, basado en valoraciones a tres niveles diferentes:

- Factores: Constituyen el nivel superior, y son la valoración propiamente dicha de la calidad.
- Criterios: Aspectos de nivel intermedio que influyen en los factores.
- Métricas: Están en el nivel inferior, son mediciones puntuales de determinados atributos y son la base para evaluar los criterios intermedios.

Entre los factores de calidad se encuentran los siguientes:

- Corrección: Grado en que un producto software cumple con sus especificaciones.
- Fiabilidad: Grado de ausencia de fallos durante la operación del producto software.
- Seguridad: Dificultad para el acceso a los datos.
- Facilidad de uso: Es la inversa del esfuerzo requerido para aprender a usar un producto.
- Mantenibilidad: Facilidad para corregir el producto en caso necesario.
- Flexibilidad: Facilidad para modificar el producto software.
- Facilidad de prueba: Inversa del esfuerzo requerido para ensayar un producto y comprobar su corrección.
- Transportabilidad: Facilidad para adaptar el producto a una plataforma diferente.
- Reusabilidad: Facilidad para emplear partes de un producto en otros desarrollos posteriores.
- Interoperabilidad: Capacidad del producto software para trabajar en combinación con otros productos.
- Eficiencia: Relación entre la cantidad de resultados suministrados y los recursos requeridos durante la operación.

Estos factores de calidad se centran en características del producto software.
Comprobar la calidad de un software es una tarea compleja, Las pruebas o ensayos consisten en hacer un producto software y comprobar si los resultados son correctos. El objetivo es descubrir los errores que pueda contener el software ensayado.
Las pruebas no permiten garantizar la calidad de un producto; si se descubre algún error, se sabe que el producto no cumple con algún criterio de calidad, pero, si la prueba no descubre ningún error, no se garantiza con ello la calidad del producto.

Cada vez que se introducen cambios en nuevas versiones, el número de errores se dispara, haciendo de nuevo necesario la corrección de los mismos.

1.1.2 Tipos de software

- Software de sistemas: Lo forman todos aquellos programas necesarios para dar soporte a otros programas. Su principal característica es su alto grado de interacción con el hardware.

- Software de aplicación: Son aplicaciones desarrolladas para resolver problemas específicos de los negocios.

- Software de ingeniería y ciencias: El objetivo es la programación de elaborados algoritmos matemáticos para modelar y simular complejos sistemas o procesos.

- Software incrustado: Reside en el interior de un producto o sistema, y su objetivo es controlarlo y definir su comportamiento. Suele ser muy específico y de pequeñas dimensiones con la necesidad de operar en tiempo real.

- Software de línea de producto: Su objetivo es dar una determinada funcionalidad al consumidor.

- Aplicaciones web ("webapps"): Tienen capacidad de cómputo y están integradas con aplicaciones y bases de datos. La comodidad, rapidez y vistosidad son determinantes a la hora de que tengan éxito.

- Software de inteligencia artificial: Utilizan algoritmos no numéricos para la resolución de los problemas.

1.2 ¿CÓMO SE FABRICA EL SOFTWARE?

En las décadas iniciales se planteaba como una actividad artesanal, basada en la labor de personas habilidosas y más o menos creativas, que actuaban de forma individual y de manera relativamente poco disciplinada.
Al aumentar la capacidad de los computadores, aumentó también la complejidad de las aplicaciones y se apreció la necesidad del trabajo en equipo, con la consiguiente división y organización del trabajo.

El software tiene una particularidad especial frente a cualquier producto físico que se pueda imaginar: una vez diseñado, este se puede replicar con tremenda facilidad, sin necesidad de un proceso de fabricación propiamente dicho.

La ingeniería de software amplia la visión del desarrollo del software como una actividad esencialmente de programación, contemplando además otras actividades de análisis y diseño previos, y de integración y verificación posteriores.

Ingeniería de software es la aplicación de un enfoque sistemático, disciplinado y cuantificable al desarrollo, operación y mantenimiento de software, y el estudio de estos enfoques que hoy, se encuentra en una situación permanente de fuerte evolución, con avances continuos en las técnicas que resultan sin, sin embargo, insuficientes en cada momento.

1.3 MITOS DEL SOFTWARE

- El hardware es mucho más importante que el software: Falso. Ya que, al usar un computador, nuestra interacción es fundamentalmente con el software, y solo de una manera muy limitada el usuario accede directamente al hardware.

- El software es fácil de desarrollar: Falso. El desarrollo de grandes sistemas es muy complejo y costoso, incluso aunque esos sistemas no empleen ningún material o hardware específico.

- El software consiste exclusivamente en programas ejecutables: Falso. Al concebir un sistema informático de manera global hay que pensar en todos los elementos que intervienen: hardware, software y personas.

- El desarrollo de software es solo una labor de programación: Falso, pues no se puede limitar el trabajo de desarrollo solo a la fase de codificación. Las tareas de análisis y diseño son el fundamento para todo el resto del desarrollo, igual que el proyecto de un arquitecto es necesario para acometer la construcción de un edificio.

- Es natural que el software contenga errores: Falso. No es admisible que los productos software siempre contengan errores, pero por desgracia, el software erróneo no puede simplemente sustituirse, ya que todas las copias del mismo son exactamente iguales.

TEMA 2: EL CICLO DE VIDA DEL SOFTWARE

2.1 EL CICLO DE VIDA DE UN PRODUCTO

El ciclo de vida en la ingeniería pasa por etapas como prediseño, diseño básico, diseño ejecutivo y diseño final. También se añaden a este ciclo de vida las acciones de diseño necesarias durante el periodo de explotación o funcionamiento del producto, acciones de mantenimiento y mejora del producto.

Si se plantea como seria el ciclo de vida en la producción industrial, también tendría sus fases. Desde la implantación de su producción, introducción de cambios, adaptaciones o nuevos elementos en la planta de producción, al mantenimiento de la producción del bien, aseguramiento de la calidad de la producción, optimización de recursos, eficiencia de los procesos que intervengan… etc.

Si se analiza el ciclo de vida de un producto en el mercado, se encontraría la evolución que experimentan las ventas de ese producto mientras se encuentra disponible en el mercado, se encontraría la evolución que experimentan las ventas de ese producto mientras se encuentra disponible en el mercado. Las fases por las que pasa este ciclo suelen se: introducción en el mercado, madurez y declive.

2.2 EL CICLO DE VIDA DEL SOFTWARE

Cuando el software se está diseñando, se debe tratar como un producto de ingeniería. Cuando se está produciendo, como cualquier producto producido por la industria. Finalmente, cuando se está comercializando, deben seguirse las pautas habituales del mercado.

Las fases en el proceso de desarrollo del software suelen ser las siguientes:

- Análisis.
- Diseño.
- Codificación.
- Integración.
- Mantenimiento.

Cada una de estas fases lleva consigo una serie de tareas que deben realizarse, conocidas como actividades, que generan documentos donde se presenta el trabajo realizado.

Algo extremadamente importante es que estos mismo s documentos servirán para que todas las partes implicadas puedan validar sus compromisos.

2.3 FASES DEL CICLO DE VIDA DEL SOFTWARE

2.3.1 Análisis

Se analizan las necesidades que tienen los usuarios del futuro sistema software y que deben ser satisfechas mediante el funcionamiento del mismo. La empresa elabora una especificación precisa del sistema a desarrollar.

2.3.2 Diseño

Consiste en elaborar un esquema o diseño donde se contemplen los elementos necesarios para que el sistema funcione según con lo especificado en el análisis. Debe establecerse la organización del sistema para su construcción. Un adecuado diseño permite la optimización de los recursos en la producción del mismo, siendo el resultado un documento de carácter gráfico, donde se presentan todos los elementos y la organización pormenorizada de cada uno. Se elaboran los planos de lo que se va a construir.

2.3.3 Codificación

Se produce lo que va a hacer funcionar el sistema software. Se construirá cada uno de los elementos que se han definido en la fase de diseño. Los elementos necesarios para comprobar que los construido funciona correctamente.

2.3.4 Integración

Después de construidos todos los elementos se procede a unirlos todos con el objetivo de construir el sistema completo. En esta fase deben realizarse pruebas exhaustivas para garantizar que el conjunto funciona durante la explotación.

2.3.5 Explotación

Comprende el periodo de funcionamiento de la aplicación. Es el objetivo final del producto desarrollado y según se devenir, marcara las fases posteriores de desarrollo como la de mantenimiento.

2.3.6 Mantenimiento

Ante nuevas situaciones de funcionamiento el sistema debe evolucionar para responder a las nuevas demandas.

El resultado de cada una de estas fases se plasma en un documento, que permiten independizar las fases y posibilitan que grupos de personas distintas trabajen en cada fase especializándose según la fase en la que se trabaje.

2.4 DOCUMENTOS QUE SE GENERAN EN EL CICLO DE VIDA

2.4.1 Documento de requisitos de software (SRD)

Producto de la fase de análisis. Consiste en una especificación precisa y completa de lo que debe hacer el sistema, prescindiendo de los detalles internos de cómo lo debe hacer.

2.4.2 Documento de diseño de software (SDD)

Producto de la fase de diseño. Consiste en una descripción de la estructura global del sistema y de la especificación de que debe hacer cada una de sus partes asi como la combinación de unas con otras.

2.4.3 Código fuente

Producto de la fase de codificación. Contiene los programas fuente, en el lenguaje de programación elegido, debidamente comentados para conseguir que se entiendan con claridad.

2.4.4 El sistema software

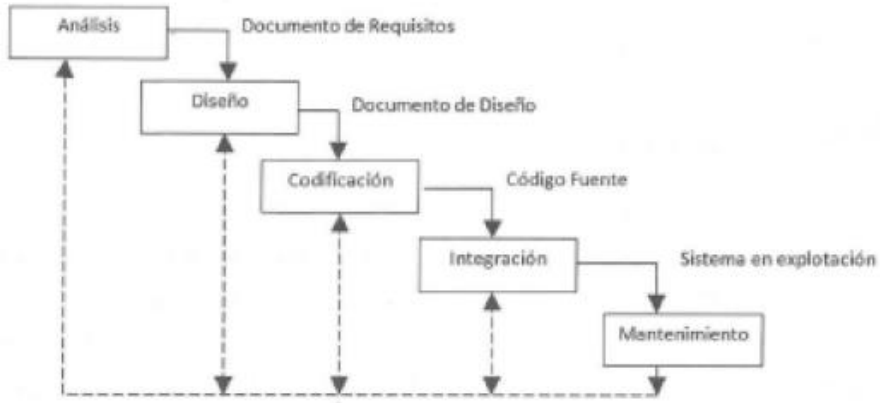
Es el ejecutable como producto de la fase de integración. Deben documentarse las pruebas realizadas para su puesta en marcha y debe describirse el procedimiento de integración.

2.4.5 Documentos de cambios

Tras cada modificación realizada en la fase de mantenimiento debe quedar constancia escrita de las acciones realizadas. Estos documentos suelen tener la siguiente estructura para cada modificación: información sobre el problema detectado, descripción de la solución adoptada y las modificaciones realizadas sobre el sistema.

2.5 TIPOS DE CICLO DE VIDA DEL SOFTWARE

2.5.1 Ciclo de vida en cascada



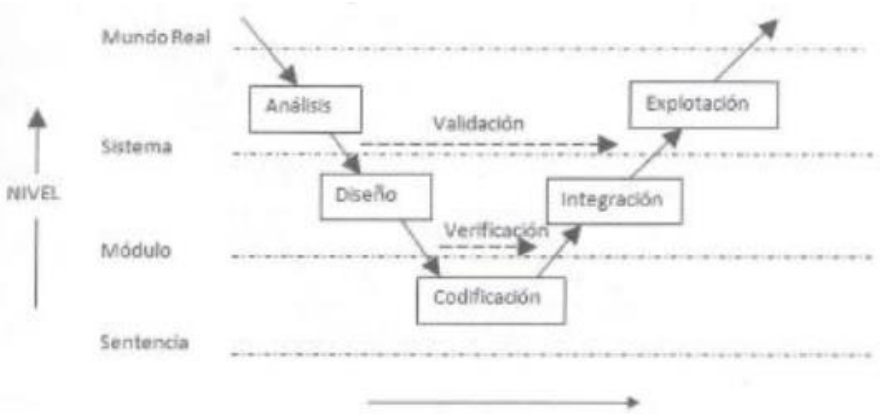
Es la secuenciación de las distintas fases de la producción del software anteriores. Como elementos de unión entre cada fase aparecen los diferentes documentos que se generan en cada fase.

Cada fase se separa claramente de la siguiente lo que permite la independización de su realización. Este modelo obliga a terminar cada fase antes de comenzar la siguiente.
Para detectar errores se establecen procesos de revisión al completar cada fase.

Si se detectan errores en una fase es necesario corregirlos en esa fase y todos los puntos del ciclo de vida anteriores.

Este modelo de ciclo de vida se puede ampliar y pormenorizar hasta el nivel que se desee dependiendo de la complejidad del sistema que se esté desarrollando.

2.5.2 Ciclo de vida en V



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

En la rama izquierda descendente, el sistema software se va descomponiendo en elementos cada vez más sencillos, hasta llegar a las sentencias del lenguaje de programación. A partir de ahí el sistema se va construyendo poco a poco a base de integrar los elementos que lo componen hasta disponer del sistema completo.

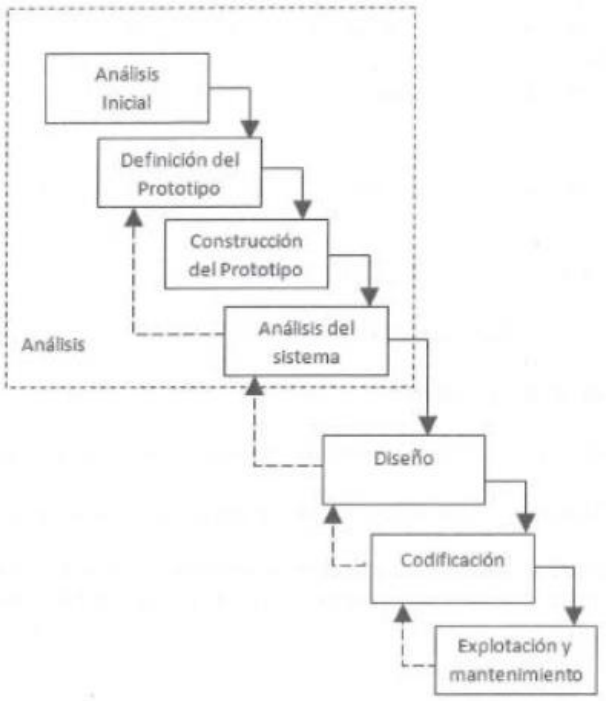
El resultado de una fase debe utilizarse en fases posteriores para comprobar que el desarrollo es correcto. La comprobación de que una parte del sistema cumple con sus especificaciones se denomina verificación y la comprobación de que un elemento satisface las necesidades del usuario identificadas durante el análisis se denomina validación.

2.6 PROTOTIPOS

Un prototipo es un sistema auxiliar que permite probar experimentalmente ciertas soluciones parciales a las necesidades del usuario o a los requisitos del sistema. Si se desarrolla con un costo sensiblemente inferior al del sistema final, los errores cometidos en el mismo no resultaran demasiados costosos.
Para reducir el costo del desarrollo se puede:

- Limitar las funciones y desarrollar solo unas pocas.
- Limitar su capacidad, permitiendo que solo se procesen unos pocos datos.
- Limitar su eficiencia, permitiendo que opere en forma lenta.
- Evitar limitaciones de diseño usando un soporte hardware más potente.
- Reducir la parte a desarrollar usando un apoyo software más potente.

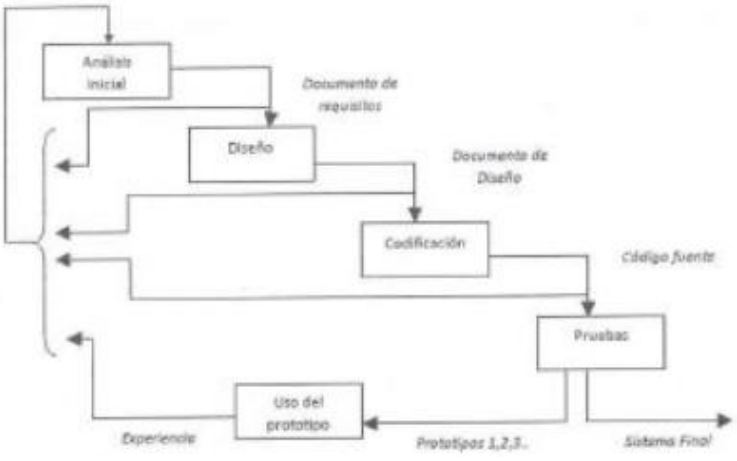
2.6.1 Prototipos rápidos, de usar y tirar o maquetas



Son aquellos cuya finalidad es solo adquirir experiencia, sin pretender aprovecharlos como producto. Se aprovechan dentro de las fases de análisis y/o diseño de un sistema, para experimentar algunas alternativas y garantizar en lo posible que las decisiones tomadas son correctas.

Lo importante es desarrollarlos en poco tiempo para evitar alargar5 excesivamente la duración de las fases de análisis y diseño.

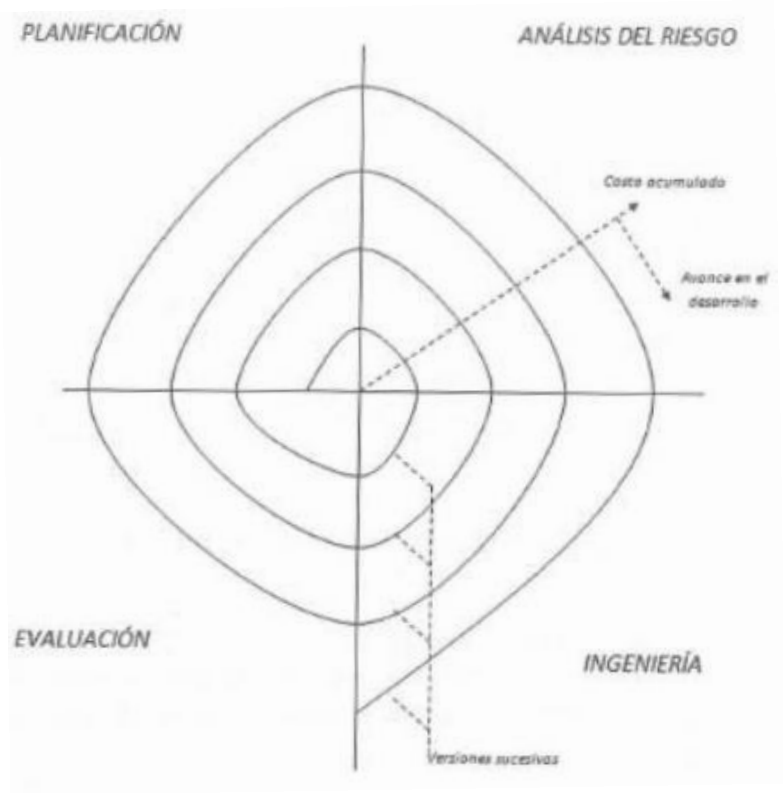
2.6.2 Prototipos evolutivos



El prototipo se desarrolla sobre el mismo soporte hardware/software que el sistema final, pero solo realizará algunas funciones, o en general, será solo una realización parcial del sistema deseado.

Este modelo puede considerarse como un proceso iterativo en bucle sobre el modelo en cascada, de manera que en cada iteración se hace solo una parte del desarrollo, avanzando un poco en cada fase.
Cada iteración utiliza todo lo que se ha generado en la iteración anterior, y produce un nuevo prototipo, que es una nueva versión parcial del sistema, hasta llegar finalmente al sistema completo.

2.7 EL MODELO EN ESPIRAL



Puede considerarse como un refinamiento del modelo evolutivo general, introduciendo la actividad de análisis de riesgo como elemento fundamental para guiar la evolución del proceso de desarrollo.

Las distintas actividades se representan sobre unos ejes cartesianos, conteniendo cada cuadrante una clase particular de actividades:

- Planificación: Sirven para establecer el contexto del desarrollo y decidir que parte del mismo se abordara en ese ciclo de la espiral.

- Análisis de riesgo: Consisten en evaluar diferentes alternativas para la realización de la parte del desarrollo elegida, seleccionando la más ventajosa.

- Ingeniería: Corresponden a las indicadas en los modelos clásicos: análisis, diseño, codificación … etc. Su resultado será ir obteniendo en cada ciclo una versión más completa del sistema.

- Evaluación: Analizan los resultados de la fase de ingeniería, habitualmente con la colaboración del cliente para quien se realiza el desarrollo.

2.8 PROGRAMACIÓN EXTREMA

Es una nueva metodología con un único objetivo: ser capaz de responder de una forma rápida y de calidad a las exigencias de los clientes, dando por sentado que los requisitos del cliente cambian a lo largo del proceso y que hay que ser capaz de adaptarse a ellos de una forma muy ágil.

"La programación extrema es un proceso ligero, de bajo riesgo, flexible, predecible, científico y divertido de desarrollar software". El equipo de desarrollo tiene que basarse en cuatro valores principales.

- Sencillez: Se debe programar solo aquello que han pedido, sin pensar en lo que podrían pedir mañana. Se utiliza la recodificación de forma continua.

- Comunicación: Se potencia el trabajo en equipo, tanto dentro del grupo de desarrollo como el cliente. Algunas prácticas son el código autodocumentado o programación por parejas.

- Retroalimentación: El cliente está integrado dentro del equipo de desarrollo y los ciclos del proceso software son muy cortos. Las pruebas unitarias sistemáticas en el momento y la integración continua permiten tener conocimiento en tiempo real sobre la respuesta del sistema.

- Valentía: Es la que permite a los programadores afrontar la recodificación del programa. La programación extrema fomenta que los miembros del equipo trabajen con confianza, sin esconderse, aportando valor y eliminando algunos de los problemas de los grandes grupos de trabajo bajo una metodología más tradicional.

La programación extrema propone ciclos del proceso muy cortos y rápidos, realizando pruebas de unidad inmediatas y una integración continua.

2.9 MANTENIMIENTO DEL SOFTWARE

Consiste normalmente en repetir o rehacer parte de las actividades de las fases anteriores para introducir cambios en una aplicación de software ya entregada al cliente y puesta en explotación.

2.9.1 Evolución de las aplicaciones

Un sistema software funcionara con la misma corrección al cabo de los años que el primer día de su empleo. Hay varios motivos para modificarlo:

- Mantenimiento correctivo: Tiene como finalidad corregir errores en el producto software que no han sido detectados y eliminados durante el desarrollo inicial del mismo.

- Mantenimiento adaptativo: Se produce en aplicaciones cuya explotación dura bastantes años, de manera que los elementos básicos hardware y software que constituyen la plataforma o entorno en que se ejecutan evoluciona a lo largo de ese tiempo.

- Mantenimiento perfectivo: Aparece sobre todo en aplicaciones sujetas a la competencia del mercado, y es necesario para ir obteniendo mejoradas del producto que permitan mantener el éxito del mismo.

2.9.2 Gestión de cambios

Se pueden distinguir dos enfoques diferentes en la gestión de cambios, dependiendo del mayor o menor grado de modificación del producto:

- Si el cambio a realizar afecta a la mayoría de los componentes del producto, dicho cambio se puede plantear como un nuevo desarrollo.

- Si el cambio afecta a una parte localizada del producto, entonces se puede organizar como simple modificación de algunos elementos.

Desde el punto de vista de gestión la realización de cambios se puede controlar mediante dos clases de documentos, que a veces se refunden en uno solo:

- Informa de problema: Describe una dificultad en la utilización del producto que requiere alguna modificación para subsanarla.

- Informe de cambio: Describe la solución dada a un problema y el cambio realizado en el producto software.

2.9.3 Reingeniería

Consiste en tomar el código fuente y a partir de él, tratar de construir, si es posible de manera automática, alguna documentación, en particular documentación de diseño, con la estructura modular de la aplicación y las dependencias entre módulos y funciones.

La actividad se plantea como algo mas general, que trata de generar un sistema bien organizado y documentado a partir del sistema inicial deficiente.

2.10 GARANTÍA DE CALIDAD DEL SOFTWARE

2.10.1 Plan de garantía de calidad

Una adecuada calidad es inalcanzable sin una buena organización del desarrollo. Debe materializarse en un documento formal, denominado Plan de Garantía de Calidad Software (SQAP), que debe contemplar aspectos tales como:

- Organización de los equipos de personas y la dirección y seguimiento del desarrollo.
- Modelo de ciclo de vida a seguir, con detalle de sus fases y actividades.
- Documentación requerida, especificando el contenido de cada documento y un guion del mismo.
- Revisiones y auditorias que se llevaran a cabo durante el desarrollo.
- Organización de las pruebas que se realizaran sobre el producto software a distintos niveles.
- Organización de la etapa de mantenimiento, especificando como ha de gestionarse la realización de cambios.

2.10.2 Revisiones

Consisten en inspeccionar el resultado de una actividad para determinar si es aceptable o, por el contrario, contiene defectos que han de ser subsanados.
Las revisiones deben ser formalizadas, es decir, deben estar contempladas en el modelo de ciclo de vida y den existir una normativa para su realización. Algunas recomendaciones para formalizar las revisiones son:

- Deben ser realizadas por un grupo de personas, y no solo por un individuo.

- El grupo que realiza la revisión debe ser reducido, para evitar excesivas discusiones y facilitar la comunicación entre sus miembros.

- La revisión no debe ser realizada por los autores del producto, sino por otras personas diferentes.

- Se debe revisar el producto, pero no el productor ni el proceso de producción.

- Si la revisión ha de decidir la aceptación o no de un producto, se debe establecer de antemano una lista formal de comprobaciones a realizar.

- Debe levantarse acta de la reunión de la revisión, conteniendo los puntos importantes de discusión y las decisiones tomadas.

2.10.3 Pruebas

Consisten en hacer funcionar un producto software o una parte de el en condiciones determinadas, y comprobar si los resultados son los correctos.
Una prueba tiene éxito si se descubre algún error, con lo que se sabe que el producto no cumple con algún criterio de calidad; por el contrario, si la prueba no descubre ningún error, no se garantiza con ello la calidad del producto, ya que pueden existir otros errores que habrían de descubrirse mediante pruebas diferentes.

2.10.4 Gestión de configuración

La configuración de software hace referencia a la manera en que diversos elementos se combinan para construir un producto software bien organizado desde el punto de vista de su explotación por el usuario y de su desarrollo o mantenimiento.

Para cubrir la doble visión del software hay que considerar como elementos componentes de la configuración todos los que intervienen en el desarrollo, que son:

- Documentos de desarrollo.
- Codigo fuente de los módulos.
- Programas, datos y resultados de las pruebas.
- Manuales de usuario.
- Documentos de mantenimiento.
- Prototipos intermedios.
- Normas particulares del proyecto.

Para mantener bajo control la configuración o configuraciones de software hay que apoyarse en técnicas particulares de:

- Control de versiones: Consiste en almacenar de forma organizada las sucesivas versiones de cada elemento de la configuración, de manera que al trabajar sobre una configuración concreta del producto software se pueda acceder cómodamente a las versiones apropiadas.

- Control de cambios: Consiste en garantizar que las diferentes configuraciones del software se componen de elementos compatibles entre sí.

El control de cambios se realiza normalmente usando el concepto de línea base, que es una configuración particular del sistema en la que cada una se construye a partir de otra mediante la inclusión de ciertos cambios.

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

2.10.5 Normas y estándares

Algunas de las normas han sido recogidas por organizaciones internacionales y establecidas como estandares a seguir en el desarrollo de determinadas aplicaciones. Entre estas normativas están:

- IEEE (Institute of Electrical and Electronics Engineer): Ha establecido una colección de normas, muchas de ellas admitidas como normas ANSI.

- DoD (Department of Defense): La norma fundamental es la DoD-STD-2167ª, que se complementa con otras normas adicionales. En la actualidad está en revisión y será sustituida por la MIL-STD-SDD.

- ESA (European Space Agency): Posee una norma general para el desarrollo de software que se apoya en algunos aspectos en las normas del IEEE.

- ISO (International Standards Organization): Integrado por los organismos nacionales de normalización de un gran número de países (AENOR en España).

- CMMI (Capability Maturity Model Integration): Es un modelo desarrollado para la mejora de los procesos de las empresas de software que califica las compañías según su nivel de madurez. Los niveles son 5:

  - Nivel 0: Inexistente.
  - Nivel 1: Inicial.
  - Nivel 2: Repetible.

- Nivel 3: Definido.
- Nivel 4: Gestionado.
- Nivel 5: Optimizado.

Entre las normas españolas se encuentran:

- METRICA-2: Desarrollada por el Consejo Superior de Informática del Ministerio para las Administraciones Publicas para el desarrollo de sistemas de Información de las administraciones públicas.

- METRICA-3: Es la evolución natural de METRICA-2 y se tienen en cuenta la división en procesos. Tiene soporte para desarrollos orientados a objetos, interfaces y consideración del mantenimiento dentro de la norma.

TEMA 3: ESPECIFICACIÓN DE REQUISITOS

3.1 MODELADO DE SISTEMAS

El modelado de los sistemas software tiene como objetivo entender mejor el funcionamiento requerido y facilitar la comprensión de los problemas planteados y no se busca un modelo físico de su comportamiento.

Un aspecto común para realizar el análisis de los requisitos es que tratan de facilitar la obtención de uno o varios modelos que detallan el comportamiento deseado del sistema.

3.1.1 Concepto de modelo

Un modelo conceptual es todo aquello que nos permite conseguir una abstracción lógico-matemática del mundo real y que facilita la comprensión del problema a resolver.
El modelo de un sistema software debe establecer las propiedades y restricciones del sistema. El modelo debe especificar QUÉ debe hacer el sistema y no CÓMO lo debe hacer.

Los objetivos que se deben cubrir con los modelos se pueden concretar en los siguientes:

- Facilitar la comprensión del problema a resolver.
- Establecer un marco para la discusión, que simplifique y sistematice la labor de análisis y revisión.
- Fijar las bases para realizar el diseño.
- Facilitar la verificación del cumplimiento de los objetivos del sistema.

3.1.2 Técnicas de modelado

- Descomposición. Modelo jerarquizado.

Cuando un problema es complejo, la primera técnica que se debe emplear es la de “divide y vencerás”. Así se establece un modelo jerarquizado en el que el problema global queda subdividido en un cierto número de subproblemas.
Este tipo de descomposición se denomina horizontal y trata de descomponer el problema funcionalmente. En el análisis de cada uno de estos subsistemas se pueden emplear las mismas técnicas que con el sistema global.

Cuando se descompone un modelo, tratando de detallar su estructura, este tipo de descomposición se denomina vertical. Esta técnica supone aplicar el método de refinamientos sucesivos al modelado del sistema.

- Aproximaciones sucesivas.

Aunque el sistema a desarrolla nunca será igual a alguno ya en funcionamiento, siempre se podrá tomar como modelo de partida el modelo de otro sistema.

Para lograr un buen resultado mediante aproximaciones sucesivas, además del analista, es fundamental contar con la colaboración de alguien que conozca muy bien el sistema anterior y que sea capaz de incorporar mejoras dentro del nuevo sistema y discutir las ventajas e inconvenientes de cada uno de los modelos intermedios elaborados.

- Empleo de diversas notaciones.

Si el modelo resulta muy complejo o incluso imposible de realizar utilizando una única notación, es importante tratar de utilizar notaciones alternativas o complementarias que simplifiquen el modelo.

Una posible notación es mediante el lenguaje natural, pero puede dar lugar a un modelo difícil de apreciar en su conjunto por lo farragoso de las explicaciones. Una figura o cualquier método grafico suele ser más fácil de entender.

Es incluso aconsejable emplear varias notaciones juntas cuando sea necesario. Asi, existen diversas herramientas de modelado para la ayuda al análisis y diseño de los sistemas llamadas herramientas CASE (Computer Aided Software Engineering) que combinan texto, tablas, diagramas, gráficos, …, etc.

- Considerar distintos puntos de vista.

Para poder concretar la creación de un modelo es necesario adoptar un determinado punto de vista. Después, el modelo que resulte estará necesariamente influido por el punto de vista adoptado.

En ocasiones será más adecuado adoptar el punto de vista del usuario futuro y en otras será más importante considerar el punto de vista del mantenedor del sistema … etc. Para esta elección será conveniente esbozar distintas alternativas y elegir aquella que resulte la más idónea.

- Realizar un análisis del dominio.

El dominio es el campo de aplicación en el que se encuadra el sistema a desarrollar.
Si bien las peculiaridades de cada aplicación hacen que necesariamente deba ser estudiada como un caso único, es importante analizar el dominio de la aplicación para situarla dentro de un entorno más global. Para realizar este análisis es aconsejable estudiar los siguientes aspectos:

- Normativa que afecte al sistema.
- Otros sistemas semejantes.
- Estudios recientes en el campo de aplicación.
- Bibliografía clásica y actualizada.
- etc.

Este estudio facilitara la creación de un modelo más universal. Como ventajas de este enfoque se tienen:

- Facilitar la comunicación entre analista y usuario del sistema.
- Creación de elementos realmente significativos del sistema.
- Reutilización posterior del software desarrollado.

3.2 ANÁLISIS DE REQUISITOS DE SOFTWARE

Con el análisis de requisitos se trata de caracterizar el problema a resolver. El primer problema que se le presenta al analista es conseguir un interlocutor válido para poder llevarla a cabo. Este interlocutor (cliente) puede estar constituido por una o varias personas expertas, en todo o solo en una parte del trabajo que se pretende automatizar.

3.2.1 Objetivos del análisis

El objetivo global es obtener las especificaciones que debe cumplir el sistema. El medio que se emplea es obtener un modelo valido, necesario y suficiente para recoger todas las necesidades y exigencias que el cliente precisa del sistema y también todas aquellas restricciones que el analista considere debe realizar el sistema.
Para lograr una especificación correcta, el modelo global del sistema deberá tener las siguientes propiedades:

- Completo y sin omisiones: Es difícil de conseguir dado que a priori no es fácil conocer todos los detalles del sistema que se pretende especificar. Por otro lado, cualquier omisión puede tener una gran incidencia en el diseño posterior e incluso desvirtuar el modelo del sistema.

- Conciso y sin trivialidades. Si el tamaño de la documentación crece desmesuradamente suele ser una prueba inequívoca de que no está siendo revisada y que tiene trivialidades o alguna inexactitud.

- Sin ambigüedades: Las consecuencias son dificultades en el diseño, retrasos y errores en la implementación e imposibilidad de verificar si el sistema cumple las especificaciones. Si se considera que el modelo resultado del análisis es un contrato con el cliente, es evidente que nada debe quedar ambiguo o de lo contrario se producirán inevitablemente fricciones y problemas de consecuencias difíciles de prever.

- Sin detalles de diseño o implementación: No se debe entrar en ningún detalle del diseño o implementación en la etapa de análisis.

- Fácilmente entendible por el cliente: Es importante que el lenguaje que se utilice sea asequible y que facilite la colaboración entre analista y cliente. Es muy interesante el empleo de notaciones graficas.

- Separando 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. Existe además las restricciones o requisitos no funcionales que están destinados a encuadrar el sistema dentro de un entorno de trabajo, ya que tienen un origen más técnico con menos interés para el cliente.

- Dividiendo y jerarquizando el modelo: Es la forma más sencilla de simplificar un modelo necesariamente complejo y dará lugar a otros submodelos.

- Fijando los criterios de validación del sistema: Es un buen método para fijar los criterios de validación, realizando con carácter preliminar el Manual de Usuario del sistema.

3.2.2 Tareas del análisis

- Estudio del sistema en su contexto.

  Los sistemas realizados mediante software son subsistemas de otros sistemas más complejos en los que se agrupan subsistemas hardware, mecánicos, sensoriales, organizativos, …, etc. Por tanto, la primera tarea del análisis del sistema software es conocer el medio en el que se van a desenvolver.

- Identificación de necesidades.

  La labor del analista es concretar las necesidades del cliente que se pueden cubrir con los medios disponibles y dentro del presupuesto y plazos de entrega asignados a la realización del sistema.

Como resultado de la labor del estudio de las sugerencias del cliente deben quedar descartadas aquellas funciones muy costosas de desarrollar y que no aportan gran cosa al sistema.

- Análisis de alternativas: Estudio de viabilidad.

La labor del analista se debe centrar en buscar aquella alternativa que cubra las necesidades reales detectadas en la tarea anterior y que tenga su viabilidad tanto técnica como económica. Con cada una de las alternativas planteadas es necesario realizar su correspondiente estudio de viabilidad.

- Establecimiento del modelo del sistema.

Según se van obteniendo conclusiones de las tareas anteriores, estas se deben ir plasmando en el modelo del sistema. El resultado final será un modelo del sistema global jerarquizado en el que aparecerán subsistemas que a su vez tendrán que ser desarrollados hasta concretar suficientemente todos los detalles del sistema que se quieren especificar.

- El documento de especificación de requisitos.

El resultado final del análisis será el documento de especificación de requisitos, que debe recoger absolutamente todas las conclusiones del análisis. Este documento será el que utilizara el diseñador como punto de partida de su trabajo y es el encargado de fijar las condiciones de validación del sistema una vez concluido su desarrollo e implementación.

- Revisión continuada del análisis.

Se debe proceder a una revisión continuada del análisis y de su documento de especificación de requisitos según se produzcan cambios. Si se prescinde de esta tarea, se corre el peligro 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.

3.3 NOTACIONES PARA LA ESPECIFICACIÓN

La especificación será fundamentalmente una descripción del modelo del sistema que se pretende desarrollar. Dependiendo de la notación que se emplee para su descripción, se obtendrán distintas especificaciones. En general, siempre será preferible utilizar una tabla o una notación grafica que el texto que las pueda describir.

La notación o notaciones empleadas deberán ser fáciles de entender por el cliente, el usuario, y en general por todos aquellos que puedan participar en el análisis y el desarrollo del sistema.

3.3.1 Lenguaje natural

Mediante explicaciones más o menos precisas y más o menos exhaustivas es posible describir prácticamente cualquier cosa.
Cuando la complejidad del sistema es mediana o grande, resulta prácticamente imposible utilizar solo el lenguaje natural por las imprecisiones, las repeticiones e incorrecciones que se pueden producir. Siempre que se utilice el lenguaje natural es muy importante organizar y estructurar los requisitos escogidos en la especificación. asi, se pueden agrupar los requisitos según su carácter:

- Funcionales.
- Calidad.
- Seguridad.
- Fiabilidad.

y dentro de cada grupo, establecer y numerar correlativamente las cláusulas de distintos niveles y subniveles que estructuren los requisitos:

1. Funcionales
    R 1.1: Modos de funcionamiento.
    R 1.2: Formatos de entrada.
    R 1.3 Formatos de salida.
    ….

Para limitar las imprecisiones y ambigüedades propias del lenguaje natural es preferible que se utilicen frases siempre con la misma construcción que tengan siempre la misma interpretación.

Ejemplo: En lugar de escribir en distintos apartados:

Cuando se teclee la clave 3 veces mal, debe invalidarse la tarjeta…
Cuando el saldo sea menor de cero se aplicará un interés…
Para los clientes mayores de 65 años se establecerá un descuento de…

es mejor que todas las frases se construyan de igual manera:

SI se teclea 3 veces mal ENTONCES invalidar tarjeta…
SI el saldo es menor de cero ENTONCES el interés será…
SI el cliente es mayor de 65 años ENTONCES el descuento será…

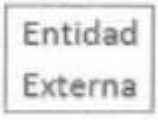
3.3.2 Diagramas de flujo de datos (DFD)

El enfoque del análisis estructurado es considerar que un sistema software es un sistema procesador de datos. Recibe datos como entrada y los procesa.

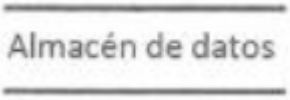
Con los diagramas de flujo de datos (DFD) se pueden modelar de forma muy sencilla las transformaciones y los flujos de datos utilizando una notación gráfica.

Flujo de datos en el sentido de la flecha.

Proceso o transformación de datos.



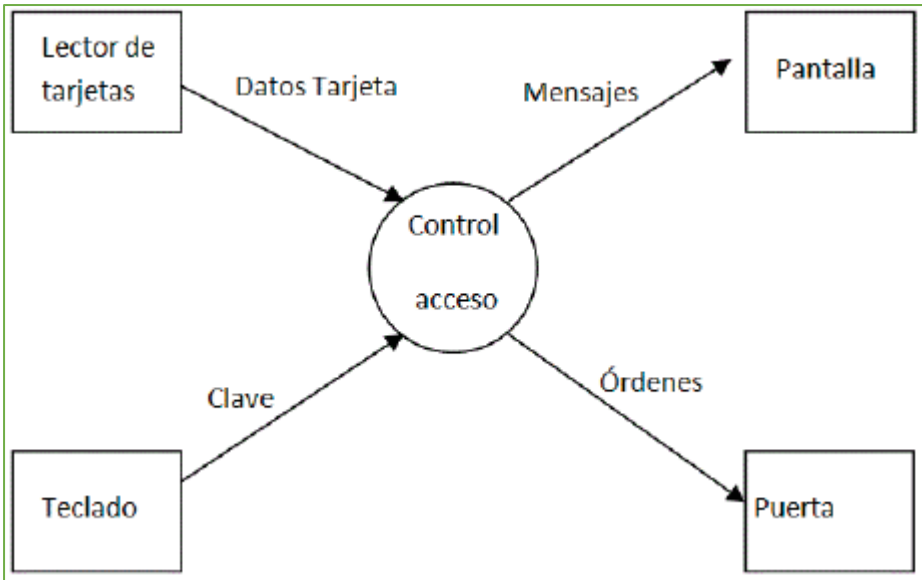
Entidad externa que produce o consume datos.



Almacén de datos con cualquier organización.

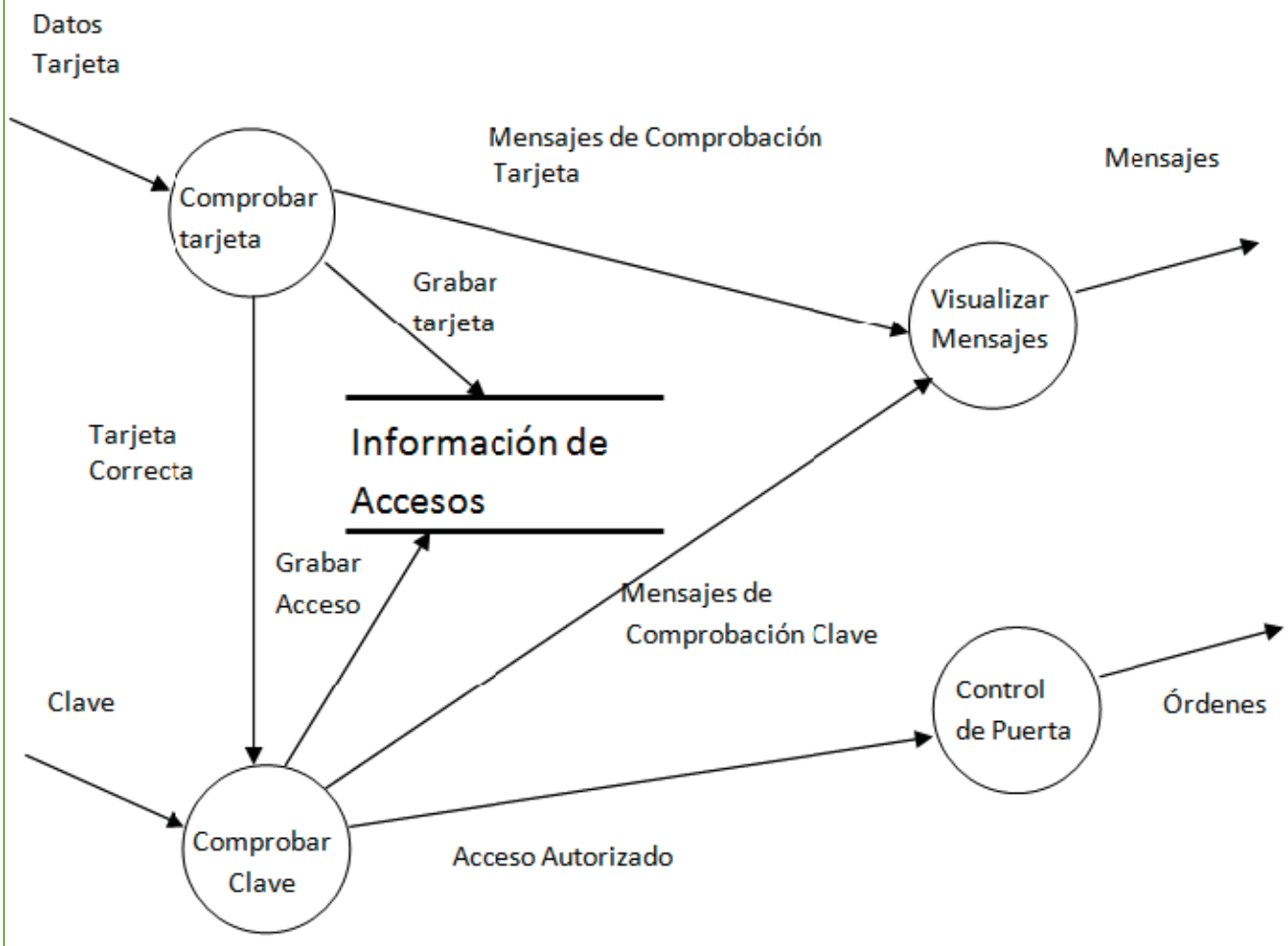
Ejemplo: Se trata de especificar un sistema para el control de acceso a un recinto mediante una tarjeta y una clave de acceso. La tarjeta magnética lleva grabada los datos personales del propietario y la clave de acceso. El sistema dispondrá de un teclado numérico para introducir la clave y un display o pantalla para escribir los mensajes. El sistema registrara todos los intentos de accesos, aunque sean fallidos por no teclear correctamente la clave o por tratar de utilizar un tipo de tarjeta incorrecta.

El DFD del sistema global o de contexto es el siguiente:



El DFD de contexto nunca es suficiente para describir el modelo del sistema que se trata de especificar. Para refinar el modelo, los DFD se usan de forma jerarquizada por niveles. El DFD de contexto se denomina nivel 0. Luego cada burbuja puede explotar y mostrar cómo se organiza interiormente mediante otros DFD.

El DFD de nivel 1 es el siguiente, en el que los flujos de entrada y salida al DFD corresponden exactamente con los que tiene el proceso control de acceso del diagrama de contexto.



La elaboración del nuevo DFD es el resultado de la labor de análisis del sistema que se está especificando. Teniendo en cuenta las necesidades del cliente, se ve que hace falta utilizar un almacén para guardar los datos de la tarjeta que intenta acceder, y si finalmente se consigue o no.

El proceso comprobar tarjeta será el encargado de verificar si la tarjeta introducida por el lector es correcta.

El proceso comprobar clave es el encargado de verificar si la clave tecleada es la misma que la que está grabada en la tarjeta. Si la clave es correcta se autoriza el acceso. En cualquier caso, se graba el resultado del intento de acceso.

El proceso visualizar mensajes es el encargado de sacar por pantalla los mensajes que le envían los dos anteriores para establecer el dialogo con la persona que trata de acceder al recinto.

El refinamiento del DFD de nivel 1 puede continuar con cada uno de los procesos que en el aparecen. Una forma de facilitar la identificación de los sucesivos DFD es numerar de forma correlativa los distintos procesos antes de su refinamiento. De esta manera los sucesivos diagramas se numeran de la forma:

Nivel 0 Diagrama de contexto
Nivel 1 DFD 0
Nivel 2 DFD 1 hasta DFD n de los procesos 1 hasta n del nivel 1
Nivel 3 DFD 1.1 hasta DFD 1.i de los procesos 1.1 hasta 1.i del nivel 2
DFD 2.1 hasta DFD 2.j de los procesos 2.1 hasta 2.j del nivel 2
…
…
DFD n.1 hasta DFD n.k de los procesos n.1 hasta n.k del nivel 2
…
Nivel 4 DFD 1.1.1 hasta DFD 1.1.x
…
DFD 1.1.1 hasta DFD 1.1.x
…
…
…
… etc.…

En todos ellos los flujos de entrada y salida antes de la explosión del proceso debe coincidir con los flujos de entrada y salida del DFD resultado de la explosión o refinamiento.

En líneas generales se tratará de describir mediante una tabla, todos los elementos básicos de información que constituyen los diversos datos que se manejan en los distintos DFD del modelo del sistema.
Los DFD sirven para establecer un modelo conceptual del sistema que facilita la estructuración de su especificación. El modelo es fundamentalmente estático dado que refleja los procesos necesarios para su desarrollo y la interrelación que existe entre ellos.
La única premisa de carácter dinámico que se puede establecer en un DFD es que en ellos se utiliza un modelo abstracto de computo del tipo flujo de datos.

3.3.3 Diagramas de transición de estados

Una manera de caracterizar un sistema dinámico es mediante la descripción de los diferentes estados por los que puede pasar y la transición entre los mismos.
El número de estados posibles que se pueden dar en un sistema software crece de una forma exponencial con su complejidad.

Para realizar la especificación de un sistema, solo es importante resaltar aquellos estados que tienen cierta trascendencia desde un punto de vista funcional. La notación para elaborar los estados es la siguiente:


Estado inicial/final del sistema. Arranque/parada del sistema.

Estado intermedio del sistema.


Evento que provoca el cambio de estado.

Los diagramas de estado mediante círculos son los más utilizados para representar máquinas de estado finitos. Sin embargo, para evitar la confusión con la notación empleada en la elaboración de los DFD, para la especificación de un sistema software se suele emplear el rectángulo.

Los eventos que provocan el cambio de estado se indican mediante una flecha dirigida desde el estado viejo al nuevo estado.

3.3.4 Descripciones funcionales. Pseudocódigo

- Selección.

```
SI <Pseudocódigo de la Condición> ENTONCES
    <Seudocódigo>
SI-NO
    <Seudocódigo>
FIN-SI
```

- Selección por casos.

```
CASO <Especificación del elemento selector>
    SI-ES <Descripción del caso 1> HACER <Seudocódigo>;
    SI-ES <Descripción del caso 2> HACER <Seudocódigo>;
    …
    SI-ES <Descripción del caso N> HACER <Seudocódigo>;
    OTROS <Seudocódigo>
FIN-CASO
```

- Iteración con pre-condición.

```
MIENTRAS <Seudocódigo de la condición> HACER
    <Seudocódigo>
FIN-MIENTRAS
```

- Iteración con post-condición.

```
REPETIR
    <Seudocódigo>
HASTA <Seudocódigo de la condición>
```

- Numero de iteraciones conocido.

```
PARA-CADA <Especificación del elemento índice> HACER
    <Seudocódigo>
FIN-PARA
```

3.3.5 Descripción de datos

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

La notación adoptada en la metodología de análisis estructurado es lo que se conoce como diccionario de datos. Esta notación es bastante más informal que cualquier definición de tipos de un lenguaje de programación, pero con ella se logra una descripción de los datos suficientemente precisa para la especificación de requisitos.

En esencia todos los formatos aconsejan que al menos se pueda describir la siguiente información para cada dato:

- Nombre o nombres.

  Es la denominación con la que se utilizara este dato en el resto de la especificación. Para mayor claridad se utilizan distintos nombres para datos que tienen una misma descripción.

- Utilidad.

  Se indican todos los procesos, descripciones funcionales, almacenes de datos, etc. en los que se utilice el dato.

- Estructura.

  Se indican los elementos de los que está construido el dato, utilizando la siguiente notación:

Operación	Descripción
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 A
/descripción/	Descripción en lenguaje natural como comentarios
=	Separador entre el nombre de un elemento y su descripción

3.3.6 Diagramas de modelo de datos

Si un sistema maneja una cierta cantidad de datos relacionados entre sí, es necesario establecer una organización que facilite las operaciones que se quieren realizar con ellos.

Es fundamental que en la especificación se establezca el modelo de datos que se considera más adecuado para conseguir todos los requisitos del sistema. Este modelo se conoce como modelo E-R y permite definir todos los datos que manejara el sistema junto con las relaciones que se desea que existan entre ellos.

Existen diversas propuestas de notación para realizar los diagramas E-R, aunque los elementos de notación más habituales son:

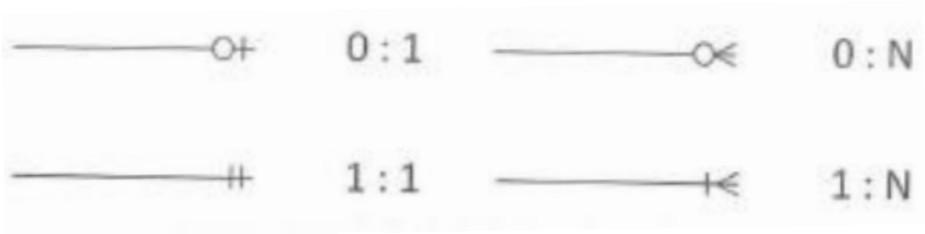
Relación entre entidades/datos.

Entidad/datos relacionados.

Min : Max Cardinalidad entre mínimo y máximo.

Sentido de la relación.

Otro elemento fundamental es la cardinalidad de la relación, esto es, entre que valores mínimo y máximo se mueve la relación entre entidades, y se indica de la siguiente forma:



3.4 DOCUMENTO DE ESPECIFICACIÓN DE REQUISITOS (SRD)

Es el documento encargado de recoger todo el fruto del trabajo realizado durante la etapa de análisis del sistema de una forma integral en un único documento.

Dado que el SRD será un documento que deberá ser revisado con cierta frecuencia a lo largo del desarrollo de la aplicación, es muy conveniente que se redacte de una forma fácil de modificar. Esto hace que la mejor manera de redactar este documento sea en forma de un contrato con sus distintas clausulas organizadas y agrupadas según el carácter de los requisitos.

1. INTRODUCCION

Debe dar una visión general de todo el documento SRD.

1.1 Objetivo

Debe exponer brevemente el objetivo del proyecto, a quien va dirigido, los participantes y el calendario.

1.2 Ámbito

Se identifica y da nombre al producto y se explica que hace y no hace cada uno y se detallan las posibles aplicaciones y beneficios del proyecto.

1.3 Definiciones, siglas y abreviaturas

Se incluye un glosario que contiene una lista de definiciones, siglas y abreviaturas

1.4 Referencias

Si el documento contiene referencias concretas a otros, se da una lista con la descripción bibliográfica de los documentos referenciados y la manera de obtener acceso a dichos documentos

1.5 Panorámica del documento

Describe la organización y el contenido del resto del documento.

2. DESCRIPCIÓN GENERAL

Se da una visión general del sistema, ampliando el contenido de la sección de introducción.

2.1 Relación con otros proyectos

Analogías y diferencias de este proyecto con otros similares o complementarios o con otros sistemas ya existentes. Si no hay se indica “No aplicable”.

2.2 Relación con proyectos anteriores y posteriores

Se indica si este proyecto es continuación de otro o si se continuara el desarrollo en proyectos posteriores. Si no hay se indica “No aplicable”.

2.3 Objetivos y funciones

Se debe describir el sistema en su conjunto con los objetivos y las funciones principales.

2.4 Consideraciones de entorno

Se describen las características especiales que debe tener el entorno en que se utilice el sistema a desarrollar. Si no se necesitan, se indica “No existen”.

2.5 Relaciones con otros sistemas

Se describen las conexiones del sistema con otros. Si el sistema no necesita intercambiar información con ningún otro, se indica “No existen”.

2.6 Restricciones generales

Se describen las restricciones generales a tener en cuenta a la hora de diseñar y desarrollar el sistema.

2.7 Descripción del modelo

Este es el apartado más extenso. Debe describir el modelo conceptual que se propone para desarrollar el sistema en su conjunto y para cada una de sus partes más relevantes.

3. REQUISITOS ESPECÍFICOS

Debe contener una lista detallada y completa de los requisitos que debe cumplir el sistema a desarrollar, que deben exponerse en la forma más precisa posible, pero sin que la descripción de un requisito individual resulte demasiado extensa.

Es importante no incluir aspectos de diseño o desarrollo y es ventajoso enunciar los requisitos en forma de una lista numerada, para facilitar su seguimiento y la validación del sistema. Cada requisito debe ir acompañado de una indicación del grado de cumplimiento necesario, es decir, si es obligatorio, recomendable u opcional.

En el documento, si no hay requisitos en alguno de ellos, se indica “No existen”.

3.1 Requisitos funcionales

Son los que describen el QUÉ debe hacer el sistema y están muy ligados al modelo conceptual propuesto

3.2 Requisitos de capacidad

Son los referentes a los volúmenes de información a procesar, tiempo de respuesta, tamaños de ficheros o discos, etc. Cuando sea necesario, se darán valores para el peor, mejor y caso más habitual.

3.3 Requisitos de interfase

Son los referentes a cualquier conexión a otros sistemas con los que se debe interactuar o comunicar.

3.4 Requisitos de operación

Son los referentes al uso del sistema en general e incluyen los requisitos de la interfase de usuario, el arranque y parada, copias de seguridad, requisitos de instalación y configuración.

3.5 Requisitos de recursos

Son los referentes a elementos hardware, software, instalaciones, etc. necesarios para el funcionamiento del sistema.

3.6 Requisitos de verificación

Son los que debe cumplir el sistema para que sea posible verificar y certificar que funciona correctamente.

3.7 Requisitos de pruebas de aceptación

Son los que deben cumplir las pruebas de aceptación a que someterá el sistema.

3.8 Requisitos de documentación

Son los referentes a la documentación que debe forma parte del producto a entregar.

3.9 Requisitos de seguridad

Son los referentes a la protección del sistema contra cualquier manipulación o utilización indebida.

3.10 Requisitos de transportabilidad

Son los referentes a la posible utilización del sistema en diversos entornos o sistemas operativos de una forma sencilla y barata.

3.11 Requisitos de calidad

Son los referentes a aspectos de calidad.

3.12 Requisitos de fiabilidad

Son los referentes al límite aceptable de fallos o caídas durante la operación del sistema.

3.13 Requisitos de mantenibilidad

Son los que debe cumplir el sistema para que se pueda realizar adecuadamente su mantenimiento durante la fase de explotación.

3.14 Requisitos de salvaguarda

Son los que debe cumplir el sistema para evitar que los errores en el funcionamiento o la operación del sistema tengan consecuencias graves en los equipos o personas.

4. APÉNDICES

Se incluyen como apéndices aquellos elementos que completen el contenido del documento y que no estén recogidos en otros documentos accesibles a los que pueda hacerse referencia.

TEMA 4: FUNDAMENTOS DEL DISEÑO DE SOFTWARE

4.1 ¿QUÉ ES EL DISEÑO?

Es la descripción del sistema a desarrollar, que se debe hacer mediante otras notaciones más formales y no solo empleando palabras. Diseñar se trata de definir y formalizar la estructura del sistema con el suficiente detalle como para permitir su realización física.

La elaboración del diseño permite una clasificación del producto de acuerdo con las características que se plasmen en el documento resultante.

El punto de partida del diseño es el documento de especificación de requisitos (SRD). El diseño es un proceso creativo que se lleva a cabo mediante prueba y error.

El método más eficaz para adquirir experiencia en el diseño es participar en alguno y aprender de los diseñadores sus técnicas de trabajo. Durante la etapa de diseño se tiene que pasar de una forma gradual del QUÉ debe hacer el sistema al CÓMO lo debe hacer, estableciendo la organización y estructura física del software. Se tiene que pasar de las ideas informales recogidas en el SRD a definiciones detallas y precisas para la realización del software mediante refinamientos sucesivos.

Las actividades habituales en el diseño de un sistema son:

1. Diseño arquitectónico: Se deben abordar los aspectos estructurales y de organización del sistema y su posible división en subsistemas estableciendo las relaciones y definiendo las interfases entre ellos. Esto proporciona una visión global del sistema.

2. Diseño detallado: Se aborda la organización de los módulos. Se trata de determinar cuál es la estructura más adecuada para uno de ellos. Aparecen nuevos módulos que se deben incorporar al diseño global, componentes que es razonable agrupar en un único modulo, o módulos que desaparecen por estar vacíos de contenido.

3. Diseño procedimental: Se encarga de abordar la organización de las operaciones o servicios que ofrecerá cada uno de los módulos. En esta actividad se detallan en pseudocódigo o PDL solamente los aspectos más relevantes de cada algoritmo.

4. Diseño de datos: Se aborda la organización de la base de datos del sistema y 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 E-R de la especificación del sistema. Se trata de concretar el formato exacto para cada dato y la organización que debe existir entre ellos.

5. Diseño de la interfaz de usuario: Cada día resulta más importante el esfuerzo de diseño destinado a conseguir un dialogo más ergonómico entre el usuario y el computador. Esta actividad es la encargada de la organización de la interfaz de usuario.

El resultado de todas las actividades es el producto de diseño y se recoge en el documento de diseño de software (SDD).

Las normas ESA establecen un Documento de Diseño Arquitectónico (ADD) y un Documento de Diseño Detallado (DDD) al que se pueden añadir como apéndices los listados de los programas una vez completado el desarrollo.

4.2 CONCEPTOS DE BASE

4.2.1 Abstracción

Una abstracción es cualquier elemento que forme parte del sistema que se quiere modelar, con la suficiente entidad o importancia para distinguirlo del resto y que este dotado de funciones relevantes para el sistema.

Cuando se diseña un nuevo sistema software es importante identificar los elementos realmente significativos de los que consta y, además, abstraer la utilidad específica de cada uno, incluso más allá del sistema software para el que se está diseñando.

Durante el proceso de diseño se debe aplicar el concepto de abstracción en todos los niveles de diseño. El proceso de abstracción se debe continuar con cada elemento software que haya que diseñar. Una buena fuente de abstracciones aplicables al diseño proviene del análisis del dominio del sistema.

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

- Abstracción funcional: Sirve para crear expresiones o acciones parametrizadas mediante el empleo de funciones o procedimientos.

- Tipos abstractos: Sirven para crear los nuevos tipos de datos que se necesitan para abordar el diseño del sistema. Además, se deben diseñar todos los métodos u operaciones que se pueden realizar con él.

- Maquina abstracta: Permite establecer un nivel de abstracción superior y en él se define de una manera formal el comportamiento de una máquina.

4.2.2 Modularidad

Uno de los primeros pasos que se debe dar al abordar un diseño es dividir el sistema en módulos claramente diferenciados, lo que permite encargar a personas diferentes el desarrollo de cada módulo y que todas ellas puedan trabajar simultáneamente.

Las ventajas de utilizar un diseño modular son:

- Claridad: Es más fácil de entender y manejar cada una de las partes o módulos que tratar de entenderlo como un todo completo.

- Reducción de costos: Es más barato desarrollar, depurar, documentar, probar y mantener un sistema modular que otro que no lo es.

- Reutilización: Si los módulos se diseñan teniendo en cuenta otras posibles aplicaciones resultara inmediata su reutilización.

Cualquier actividad productiva que se desarrolle de manera discontinua en el tiempo por un grupo de personas debe ser descompuesta y desarrollada de manera modular.

4.3 REFINAMIENTO

En un diseño siempre se parte inicialmente de una idea no muy concreta que se va refinando en sucesivas aproximaciones hasta perfilar el más mínimo detalle.

El refinamiento es la forma natural de acercar el lenguaje natural y el de programación. EL objetivo global de un nuevo sistema software expresado en su especificación se debe refinar en sucesivos pases hasta que todo quede expresado en el lenguaje de programación.

4.3.1 Estructuras 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 de cada uno de ellos afectan de forma decisiva. Una decisión errónea respecto a si un determinado resultado se guarda o se calcula nuevamente cada vez que se necesite, puede hacer tan lento el sistema que resulte inservible.

Para el diseño basta tener en cuenta las estructuras fundamentales:

- Registros.
- Conjuntos.
- Formaciones.
- Listas.
- Pilas.
- Colas.
- Arboles.
- Grafos.
- Tablas.
- Ficheros.

4.3.2 Ocultación

Para poder utilizar correctamente un programa no es necesario conocer cuál es su estructura interna. Al programador usuario de un módulo desarrollado por otro programador del equipo puede quedarle completamente oculta la organización de los datos internos que maneja y el detalle de los algoritmos que emplea.

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 resultan irrelevantes para su utilización.

Generalmente se trata de ocultar al usuario todo lo que pueda ser susceptible de cabio en el futuro y además es irrelevante para el uso. Las ventajas de aplicar este concepto son las siguientes:

- Depuración: Es más sencillo detectar que modulo concreto no funciona correctamente y es más fácil establecer estrategias y programas de prueba que verifiquen y depuren cada módulo por separado en base a lo que hacen sin tener en cuenta cómo.

- Mantenimiento: Cualquier modificación u operación de mantenimiento que se necesite en un módulo concreto no afectara al resto de los módulos del sistema.

4.3.3 Genericidad

Un posible enfoque de diseño es agrupar aquellos elementos del sistema que utilizan estructuras semejantes o que necesitan tratamiento similar. Si se prescinde de los aspectos específicos de cada elemento concreto es bastante razonable diseñar un elemento genérico con las características comunes a todos los elementos agrupados. Posteriormente, cada uno de los elementos agrupados se pueden diseñar como un caso particular del elemento genérico.

El concepto de genericidad es de gran utilidad en el diseño y da lugar a soluciones simples y fáciles de mantener. Sin embargo, la implementación de los elementos genéricos obtenidos como resultado del diseño puede resultar bastante compleja e incluso desvirtuar el propio concepto de genericidad.

En el modelo orientado a objetos, se maneja el concepto de clase, que se construyen en base a tipos de datos. Si se sustituyen algunos de los tipos de datos por parámetros formales que los representan, y que pueden materializarse en los tipos de datos, se puede considerar que estos parámetros formales son tipos genéricos.

El uso de plantillas en C++ permite definir funciones y clases genéricas mediante la utilización de datos genéricos, que se escriben como parámetros formales.

4.3.4 Herencia

Los elementos hijos heredan del padre su estructura y operaciones para ampliarlos, mejorarlos o simplemente adaptarlos a sus necesidades. A su vez los elementos hijos pueden tener otros hijos que hereden de ellos de una forma semejante.
La herencia permite reutilizar una gran cantidad de software ya desarrollado.

La aplicación del concepto de herencia en la fase de diseño es posible sin mayor problema. Sin embargo, para trasladar de una manera directa y sencilla los resultados del diseño a la codificación es aconsejable utilizar un lenguaje de programación orientado a objetos, como:

- Smalltalk.
- Object Pascal.
- Objective-C.
- Eiffel.
- C++.
- Java.

4.3.5 Polimorfismo

Engloba distintas posibilidades utilizadas habitualmente para conseguir que un mismo elementos software adquiera varias formas simultáneamente:

- El concepto de genericidad es una manera de lograr que un elemento genérico pueda adquirir distintas formas cuando se particulariza su utilización.

- El concepto de polimorfismo está muy unido al concepto de herencia. Las estructuras y operaciones heredadas se pueden adaptar a las necesidades concretas del elemento hijo.

- Existe otra posibilidad de polimorfismo que se conoce como Sobrecarga. En este caso quienes adquieren múltiples formas son los operadores, funciones o procedimientos.

Todas estas posibilidades de polimorfismo redundan en una mayor facilidad para realizar software reutilizable y mantenible. Los elementos que se propongan deberán resultar familiares al mayor número posible de usuarios potenciales.

4.3.6 Concurrencia

Los computadores disponen de una gran capacidad de proceso que no se debe desaprovechar. mientras el operador decide la siguiente tecla que debe pulsar o durante el tiempo en que se está imprimiendo, el computador puede realizar de manera concurrente otras tareas.

Cuando se trata de diseñar un sistema con restricciones de tiempo se debe tener en cuenta lo siguiente:

- Tareas concurrentes: Determinar que tareas se deben ejecutar en paralelo para cumplir con las restricciones impuestas.

- Sincronización de tareas: Determinar los puntos de sincronización entre las distintas tareas. Normalmente las tareas nunca funcionan cada una por separado sin tener en cuenta las demás.

- Comunicación entre tareas: Determinar si la cooperación se basa en el empleo de datos compartidos o mediante el paso de mensajes entre las tareas. En el caso de utilizar datos compartidos se tendrá que evitar que los datos puedan ser modificados en el momento de la consulta.

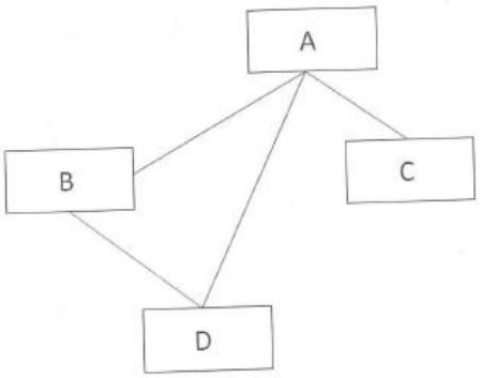
- Interbloqueos: Un interbloqueo se produce cuando una o varias tareas permanecen esperando por tiempo indefinido una situación que no puede producir nunca.

El concepto de concurrencia introduce una complejidad adicional al sistema y por tanto solo se debe utilizar cuando no exista una solución de tipo secuencial sencilla que cumpla con los requisitos especificados en el documento SRD-

4.4 NOTACIONES PARA EL DISEÑO

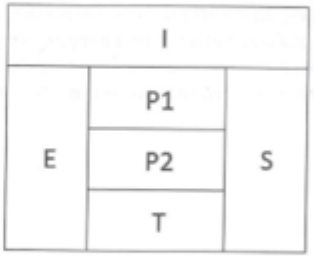
4.4.1 Notaciones estructurales

Sirven para cubrir un primer nivel del diseño arquitectónico y con ellas se trata de desglosar y estructurar el sistema en sus partes fundamentales. Una notación habitual para desglosar un sistema en sus partes es el empleo de diagramas de bloques:



Cuando la notación es informal, es necesario concretar explícitamente si con el diagrama se indica algo más que la simple conexión existente entre los bloques.

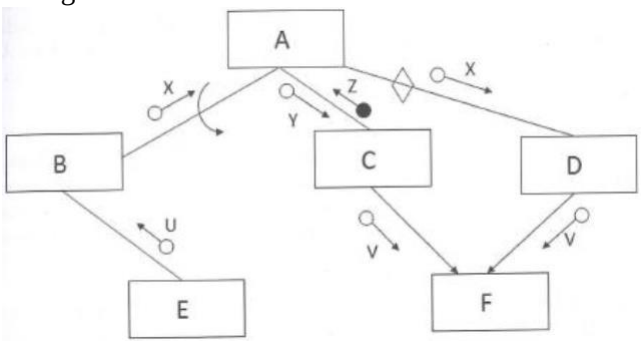
Otra notación informal para estructurar el sistema es el de las cajas adosadas para delimitar bloques.



La conexión entre los bloques se pone de manifiesto cuando entre dos cajas existe una frontera.

Algunas notaciones más formales para describir la estructura de un sistema son:

- Diagramas de estructura:



Describen la estructura de los sistemas software como una jerarquía de subprogramas o módulos en general. El significado de los símbolos es el siguiente:

- Rectángulo: Representa un módulo cuyo nombre se indica en su interior.

- Línea: Indica que el módulo superior llama o utiliza al módulo inferior. A veces la línea acaba en una flecha junto al módulo inferior al que apunta.

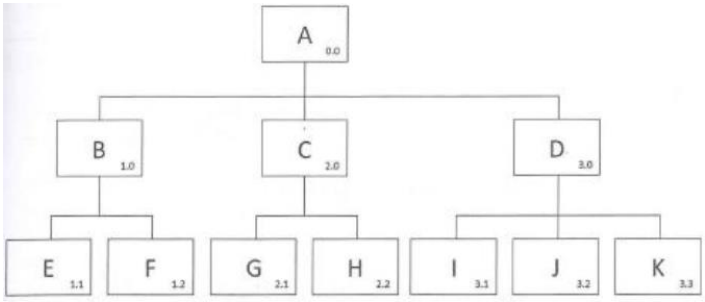
- Rombo: Se sitúa sobre una línea e indica que esa llamada o utilización es opcional.

- Arco: Se sitúa sobre una línea e indica que esa llamada o utilización se efectúa de manera repetitiva.

- Circulo con flecha: Se sitúa paralelo a una línea y representa el envío de datos, cuyo nombre acompaña al símbolo, desde un módulo a otro. Para indicar que los datos son una información de control se utiliza un circulo relleno.

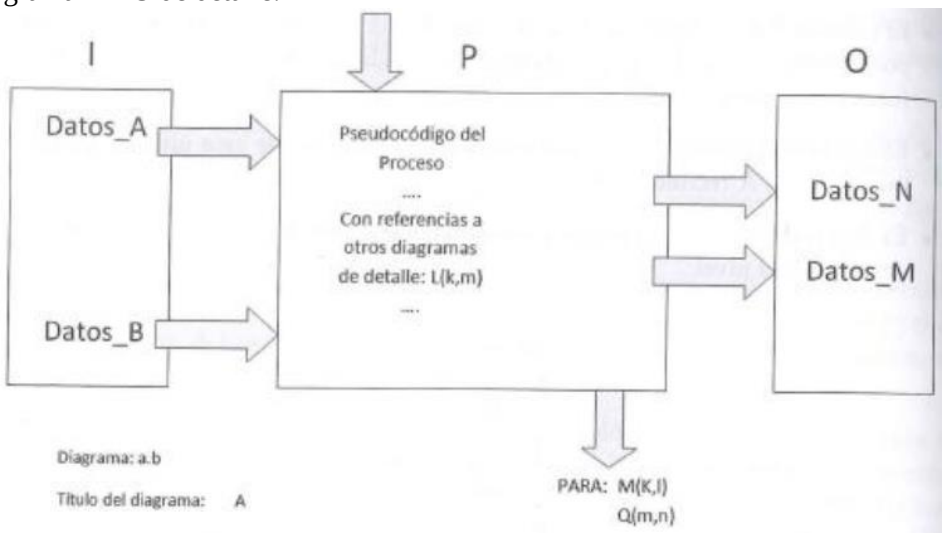
- Diagramas HIPO (Hierachy-Input-Process-Output): Facilita y simplifica el diseño y desarrollo de sistemas software, destinados a gestión. La mayoría de estos sistemas se pueden diseñar como una estructura jerarquizada (Hierachy) de subprogramas o módulos. Además, el formato de todos los módulos se puede adaptar a un mismo patrón caracterizado por los datos de entrada (Input), el tipo de proceso (Process) que se realizar con los datos y el resultado de salida que proporciona (Output). Hay dos tipos de diagramas HIPO:

- Diagrama HIPO de contenidos.



Se utilizan para establecer la jerarquía entre los módulos del sistema.

- Diagrama HIPO de detalle.



Constan de tres zonas: Entrada (I), Proceso (P) y Salida (O). En las zonas de entrada y salida se indican respectivamente los datos que entran y salen. En la zona central se detalla el pseudocódigo del proceso con referencia a otros diagramas de detalle de nivel inferior en la jerarquía. La lista de los diagramas referenciados se lista a continuación de la partícula PARA: Por la parte superior y a continuación de la partícula DE: se indica el diagrama de detalle de nivel superior.

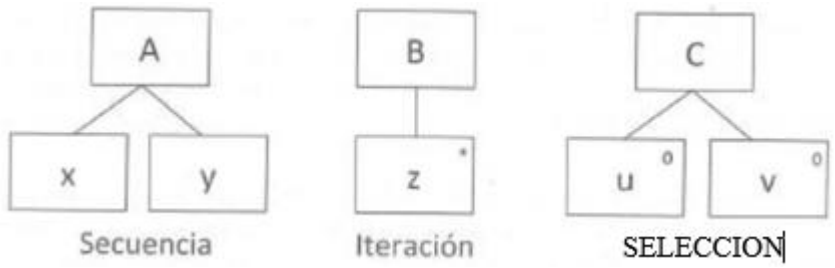
- Diagrama de Jackson: El proceso de diseño es bastante sistemático y se lleva a cabo en tres pasos:

1. Especificación de las estructuras de los 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.

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

La notación propuesta por Jackson es la siguiente:



La estructura del elemento superior se detalla según queda indicado por los elementos inmediatamente inferiores.

4.4.2 Notaciones estáticas

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.

Las notaciones que se pueden emplear para describir el resultado del diseño son las mismas que se emplean para realizar la especificación.

- Diccionario de datos: Se detalla la estructura interna de los datos que maneja el sistema.

- Diagramas E-R: Permite definir el modelo de datos, las relaciones entre los datos y en general la organización de la información.

- Notaciones dinámicas: Permiten describir el comportamiento del sistema durante su funcionamiento.

- Diagramas de flujo de datos: Desde el punto de vista del diseño, los diagramas de flujo de datos serán mucho más exhaustivos que los de la especificación debido a la necesidad de describir como se hacen internamente las cosas.

- 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.

- Lenguaje de descripción de programas (PDL): Esta notación se utiliza tanto para realizar la especificación funcional del sistema como para elaborar el diseño del mismo. La notación denominada Ada-PDL permite declarar estructuras de datos e incluso existen compiladores para verificar el pseudocódigo.

- Notaciones hibridas: Tratan de cubrir simultáneamente aspectos estructurales, estáticos y dinámicos. La metodología de Jackson utiliza una notación estructural que describe, según los casos, la organización de la información (estática) o el comportamiento (dinámico) del sistema.

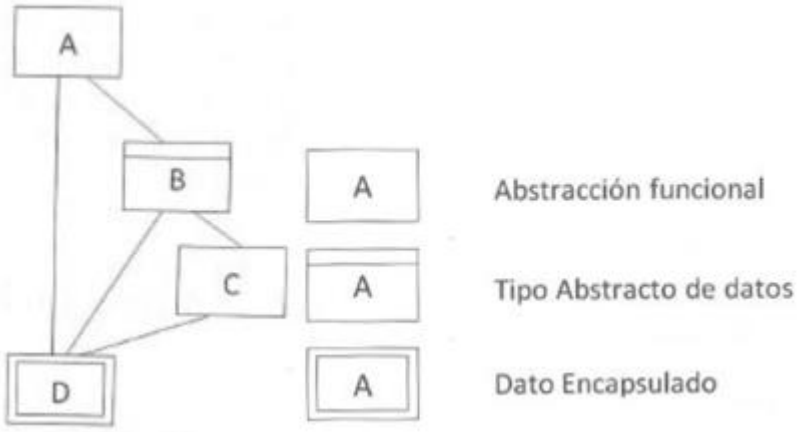
- Diagramas de abstracciones: En una abstracción se distinguen tres partes o elementos claramente diferenciados:

  - Nombre: 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 de la abstracción.

El diagrama de estructura de un sistema es:



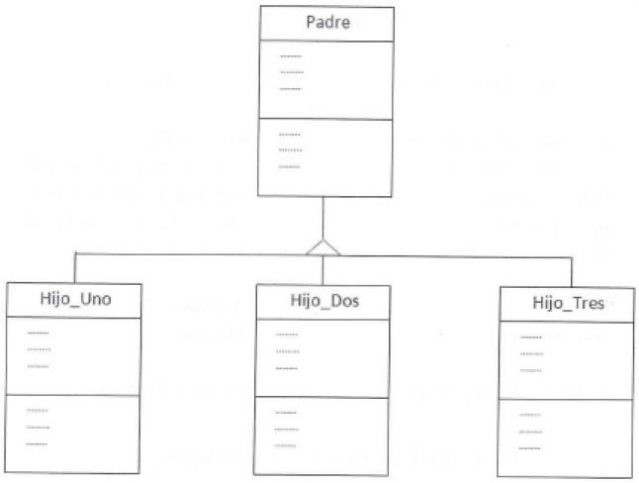
- Diagramas de objetos: Las diferencias fundamentales entre la metodología basada en abstracción y la basada en objetos son las siguientes:

  - No existe nada equivalente a los datos encapsulados ni a las abstracciones funcionales cuando se utilizan objetos en forma estricta.

  - Solo entre objetos se contempla la relación de herencia.

Si se considera un objeto formado exclusivamente por sus atributos, se tiene una estructura de datos, que puede formar parte de un diagrama de modelo de datos E-R. Debido a las propiedades particulares de los objetos, se pueden establecer dos tipos de relaciones especiales.

- Clasificación, especialización o herencia.

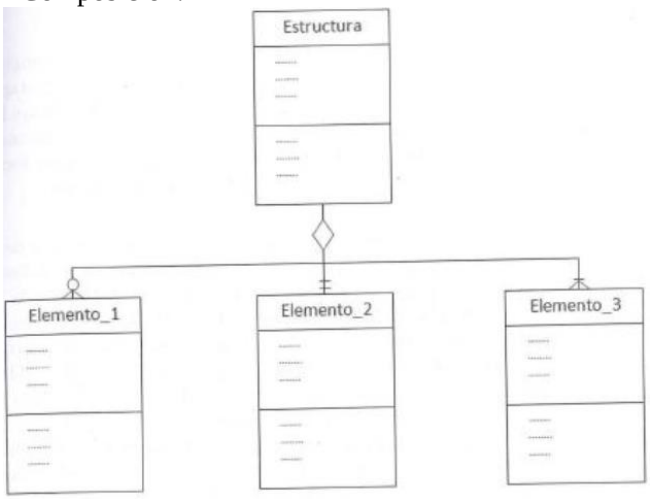


Permite diseñar un sistema aplicando el concepto de herencia.

Con el triángulo se indica que los objetos inferiores heredan atributos y las operaciones del objeto superior.

Con la relación de herencia no es necesario indicar la cardinalidad entre las clases de objetos.

- Composición.



Solo hay que indicar la cardinalidad en un sentido. Cada objeto hijo solo puede formar parte de un objeto padre. Solo falta indicar que numero mínimo y máximo de objetos de cada clase hija se pueden utilizar para componer un objeto de la clase padre.

4.5 DOCUMENTO DE DISEÑO

Las normas de la ESA establecen el empleo adecuado 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.

4.5.1 Documento ADD

1. INTRODUCCION.

   Debe dar una visión general de todo el documento ADDD. Los contenidos de sus apartados son similares a los del documento SRD, pero referidos al sistema tal y como se ha diseñado.

2. PANORAMICA DEL SISTEMA.

   Debe dar una visión general de los requisitos funcionales del sistema que ha de ser diseñado, haciendo referencia al documento SRD

3. CONTEXTO DEL SISTEMA

   Se indica si este sistema posee conexiones con otros y si debe funcionar integrada con ellos. En cada uno de sus apartados se definirá la correspondiente interfase que se debe utilizar con cada uno de los otros sistemas.

4. DISEÑO DEL SISTEMA.

   Describe el nivel superior del diseño, en que se considera el sistema en su conjunto y se hace una primera estructuración en componentes

   4.1 Metodología de diseño de alto nivel.

      Se describe brevemente la metodología a seguir en el proceso de diseño de la arquitectura del sistema.

   4.2 Descomposición del sistema.

      Se describe el primer nivel de descomposición del sistema en sus componentes principales.

5. DISEÑO DE LOS COMPONENTES.

   5.n Identificador del componente.

      Nombre del componente. Cos componentes no podrán tener nunca el mismo nombre.

      5.n.1 Tipo.

         Se describe la clase del componente.

      5.n.2 Objetivo.

         Se debe describir la necesidad de que exista el componente

      5.n.3 Función.

         Se describe que hace el componente. Esto se puede detallar mediante la transformación entrada/salida que realiza.

5.n.4 Subordinados.

Se enumeran todos los componentes usados por este.

5.n.5 Dependencias.

Se enumeran los componentes que usan a este.

5.n.6 interfases.

Se describe como otros componentes interactúan con este. Se tienen que establecer las distintas formas de interacción, las reglas y las restricciones para cada una de ellas.

5.n.7 Recursos.

Se describen los elementos usados por este componente que son externos a este diseño.

5.n.8 Referencias.

Se representan todas las referencias utilizadas.

5.n.9 Proceso.

Se describen los algoritmos o reglas que utiliza el componente para realizar su función como un refinamiento de la subsección 5.n.3.

5.n.10 Datos.

Se describen los datos internos del componente incluyendo el método de representación, valores iniciales, formato, valores válidos, etc.

6. VIABILIDAD DE 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 DE REQUISITOS COMPONENTES.

Se muestra una matriz poniendo en las filas todos los requisitos y en las columnas todos los componentes.

4.5.2 Documento DDD

La diferencia entre en ADDD y el DDD es el nivel de detalle al que desciende. En este documento existen un mayor número de componentes y para cada uno de ellos se baja incluso hasta el nivel de codificación.

Parte 1. DESCRIPCION GENERAL.

1. INTRODUCCION.

1.1 Objetivo.
1.2 Ámbito.
1.3 Definiciones, siglas y abreviaturas.
1.4 Referencias.
1.5 Panorámica del documento

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 software.

Parte 2. ESPECIFICACIONES DE DISEÑO DETALLADO.

n. Identificación 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.

APENDICE A. LISTADOS FUENTE.
APENDICE B. MATRIZ REQUISITOS/COMPONENTES.

Hay que destacar la sección 2 dedicada a recoger todas las normas, convenios y procedimientos de trabajo que se deben aplicar durante el desarrollo del sistema. la importancia de esta sección es muy grande y de ella depende que el trabajo realizado por un equipo amplio de personas tenga una estructura coherente y homogénea.

TEMA 5: TÉCNICAS GENERALES DEL DISEÑO DE SOFTWARE

5.1 DESCOMPOSICIÓN MODULAR

Para lograr la descomposición modular es necesario concretar los siguientes aspectos:

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

Un módulo es un fragmento de un sistema software que se puede elaborar con relativa independencia de los demás. Algunos de ellos pueden ser los siguientes:

- Codigo fuente: Contiene texto fuente escrito en el lenguaje de programación elegido. Es el tipo de modulo considerado como tal con mayor frecuencia y en el que se mayor hincapié en las diferentes técnicas de diseño.

- Tabla de datos: Se utiliza para tabular ciertos datos de inicialización, experimentales o simplemente de difícil obtención.

- Configuración: Un sistema se puede concebir para trabajar en entornos diversos según las necesidades de cada cliente. En estos casos, interesa agrupar en un mismo modulo toda aquella información que permite configurar el entorno concreto de trabajo.

- Otros: En general, un módulo puede servir para agrupar ciertos elementos del sistema relacionados entre sí y que se puedan tratar de forma separada del resto.

Un mismo sistema se puede descomponer en módulos de muchas formas diferentes según el diseñador, pero el objetivo fundamental de cualquier diseño es conseguir un sistema mantenible y solo en casos excepcionales se sacrificará este objetivo para lograr una mayor velocidad de proceso o un menor tamaño de código.

5.1.1 Independencia funcional

En la matriz de requisitos/componentes del final de los documentos ADD y DDD es necesario indicar que modulo se encargara de realizar cada uno de los requisitos (funciones) indicados en el documento SRD. Se puede establecer que cada función se podrá realizar en un módulo distinto.

Si el análisis está bien hecho y las funciones son independientes, estos módulos tendrán independencia funcional entre ellos. Para que un módulo posea independencia funcional debe realizar una función concreta sin apenas ninguna relación con el resto de módulos del sistema.
Al descomponer un sistema en módulos es necesario que existan ciertas relaciones entre ellos. En todo caso se trata de reducir las relaciones entre módulos al mínimo. Una mayor independencia redunda en una mayor facilidad de mantenimiento o sustitución de un módulo por otro y aumenta la posibilidad de reutilización del módulo.

Para medir de una forma relativa la independencia funcional que hay entre varios módulos se utilizan fundamentalmente dos criterios:

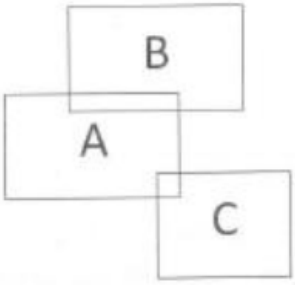
- Acoplamiento.

El grado de acoplamiento entre módulos es una medida de la interrelación que existe entre ellos. Como escala para medir el grado de acoplamiento se utiliza la siguiente:

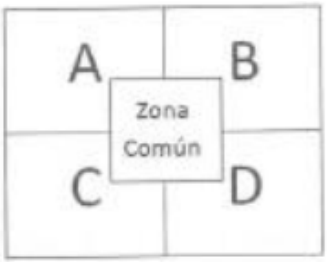
	↑	Acoplamiento por Contenido
FUERTE	|	Acoplamiento Común
	|	Acoplamiento Externo
MODERADO	|	Acoplamiento de Control
	|	Acoplamiento por Etiqueta
DÉBIL	|	Acoplamiento de Datos
	|	Sin Acoplamiento Directo

Con ella se trata de conocer lo que se debe hacer para conseguir un acoplamiento débil y lo que se debe evitar para que no exista un acoplamiento fuerte entre módulos. El objetivo es que durante el diseño se tenga en cuenta la escala para buscar descomposiciones con acoplamiento débil, o a lo sumo, moderado.

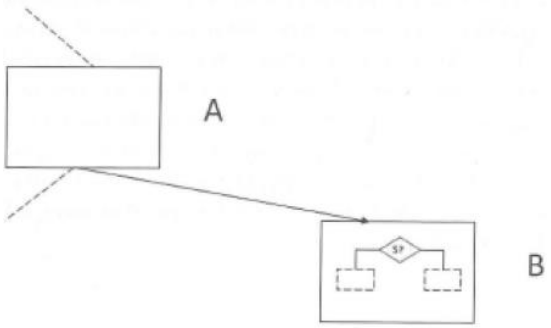
El acoplamiento por contenido se produce cuando desde un módulo se pueden cambiar los datos locales e incluso el código de otro modulo. Este tipo de acoplamiento solo se pude lograr utilizando un lenguaje ensamblador o de muy bajo nivel y debe ser evitado siempre.



El acoplamiento común emplea una zona común de datos a la que tienen acceso varios o todos los módulos del sistema, lo que significa que cada módulo puede estructurar y manejar la zona común con total libertad son tener en cuenta para nada al resto.



El acoplamiento externo tiene la zona común constituida por algún dispositivo externo al que están ligados todos los módulos. Aunque este acoplamiento es inevitable, siempre se deben buscar descomposiciones en que los dispositivos externos afecten al mínimo número posible de módulos.



El acoplamiento débil solo se produce a través del intercambio de aquellos datos que un módulo necesita de otro. Si el intercambio se realiza estrictamente con los únicos datos que se necesitan se tiene un acoplamiento de datos.

Cuando en el intercambio se suministra una referencia que facilita el acceso a la estructura completa de la que forman parte se tiene un acoplamiento por etiqueta.

Con ambos tipos de acoplamiento cualquier modificación en un módulo afecta muy poco o nada al resto. Sin embargo, cuando se utiliza un acoplamiento fuerte es necesario ser muy meticuloso y consultar siempre a todo el equipo de diseño e implementación antes de introducir ningún cambio en las zonas comunes.

- Cohesión.

Además de buscar un acoplamiento débil entre módulos, es necesario lograr que el contenido de cada uno tenga coherencia. Se debe tener en cuenta que el número de módulos no debe crecer desmesuradamente para no aumentar la complejidad del sistema. Esto hace que ciertos elementos sueltos se incorporen a un determinado modulo, aunque no tengan mucha afinidad con él y que como resultado se disminuya la cohesión del módulo. Como escala para medir de forma cualitativa la cohesión de un módulo se utiliza la siguiente.

	↑	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 cualquier relación entre los elementos del módulo no guardan ninguna relación entre ellos.

La cohesión lógica se produce cuando se agrupan en un mismo modulo elementos que realizan funciones similares desde un punto de vista de usuario. Esta cohesión es la que existe en los módulos de entrada/salida o cuando se diseña un módulo para el manejo de todos los mensajes de error que se producen en el sistema.

Una cohesión temporal es el resultado de agrupar en un mismo modulo aquellos elementos que se ejecutaran en un mismo momento.

La cohesión de comunicación es aquella que se produce cuando todos los elementos del módulo que operan con el mismo conjunto de datos de entrada o producen el mismo conjunto de datos de salida.

La cohesión secuencial es la que se produce cuando todos los elementos del módulo trabajan de forma secuencial.

En la cohesión funcional cada elemento del módulo está encargado de la realización de una función concreta y especifica. Con las técnicas de diseño funcional descendente este nivel de cohesión es el máximo que se puede lograr dentro de un módulo.

La cohesión abstraccional se logra cuando se diseña un módulo como tipo abstracto de datos con la técnica basada en abstracciones o como una clase de objetos con la técnica orientada a objetos.

Para conocer y mejorar la cohesión de un módulo se sugiere realizar una descripción de su comportamiento, en la que a partir de la descripción se puede establecer el grado de cohesión de acuerdo con los siguientes criterios:

- Si la descripción es una frase compuesta que contiene comas o más de un verbo, es probable que se incluyendo más de una función y la cohesión será secuencial o de comunicación.

- Si la descripción contiene palabras relacionadas con el tiempo como “primero”, “después”, “entonces”, etc. la cohesión será temporal o secuencial.

- Si la frase de la descripción no se refiere a algo especifico a continuación del verbo es muy probable que sea una cohesión lógica.

- Cuando se utilizan palabras como “inicializar”, “preparar”, etc. la cohesión será temporal.

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

- Comprensibilidad.

Para facilitar y posibilitar los cambios es necesario que cada módulo sea comprensible de forma aislada. El primer factor es la independencia funcional. Con una cohesión alta y un acoplamiento débil el módulo tiene menor dependencia del resto del sistema y por tanto será más fácil de entender su funcionamiento de manera aislada. Sin embargo, esto no es suficiente y es necesario cuidar los siguientes factores:

- Identificación: Una elección adecuada del nombre del módulo y de los nombres de cada uno de sus elementos facilita mucho su comprensión.

- Documentación: La labor de documentación de cada módulo y del sistema en general debe servir para facilitar la comprensión, aclarando todos aquellos aspectos de diseño o implementación que por su naturaleza no puedan quedar reflejados de otra manera.

- Simplicidad: Las soluciones sencillas son siempre las mejores. Un esfuerzo fundamental del diseñador debe estar dedicado a simplificar al máximo las soluciones propuestas.

5.2.2 Adaptabilidad

Independencia funcional y comprensibilidad son dos cualidades esenciales que debe tener cualquier diseño para posibilitar su adaptabilidad. Las adaptaciones suelen ser múltiples y variadas a lo largo de la vida del sistema. Asi, es necesario cuidar otros factores adicionales para facilitar la adaptabilidad:

- Previsión: Resulta muy complicado prever que evolución futura tendrá un determinado sistema. Si la adaptación exige una modificación de la descomposición modular resultara tremendamente más complicada y costosa de realizar.

- Accesibilidad: Previamente a cualquier adaptación es imprescindible estudiar la estructura global del sistema y todos aquellos detalles a los que afecte de manera fundamental la adaptación. Todos los entornos de lenguajes de programación basados en objetos mantienen una biblioteca de objetos de fácil accesibilidad y gracias al mecanismo de herencia resulta relativamente sencilla su adaptabilidad.

- Consistencia: Cuando se realizan adaptaciones sucesivas es vital mantener la consistencia entre todos los documentos. La modificación de los programas fuentes sin actualizar ningún otro documento de lugar a un caos en el que es imposible saber a qué adaptación pertenece cada nueva versión de modulo.

5.2 TÉCNICAS DE DISEÑO FUNCIONAL DESCENDENTE

Se incluyen todas aquella en que la descomposición del sistema se hace desde un punto de vista funcional. De esta manera los módulos corresponden a funciones concretas. Por esta razón estas técnicas de diseño conducen a estructuras modulares que pueden implementarse bien casi con cualquier lenguaje de programación.

5.2.1 Desarrollo por refinamiento progresivo

Corresponde a la aplicación en la fase de programación estructurada que condujo 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 final.

La construcción de programas basada en refinamiento consiste en plantear inicialmente el programa como una operación global e irla descomponiendo en función de otras operaciones más sencillas.
La aplicación de esta técnica a la fase de diseño consiste en realizar solo los primeros niveles de refinamiento, asignando a módulos separados las operaciones parciales que se van identificando.

La técnica de programación estructurada de Jackson sigue estrictamente las ideas de la programación estructurada en cuando a las estructuras recomendadas y el método de refinamientos sucesivos. Sa aportación principal está en las recomendaciones para ir construyendo la estructura del programa, que debe hacerse similar, en lo posible, a las estructuras de los datos de entrada y salida. La técnica original se basa en:

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

5.2.2 Diseño estructurado

Es el complemento del análisis estructurado. ambas técnicas coinciden en el empleo de los diagramas de flujo de datos (DFD) como medio fundamental de representación del modelo funcional del sistema.

La tarea de diseño consiste en pasar los DFD a los diagramas de estructura estableciendo una jerarquía o estructura de control entre los diferentes módulos, que no está implícita en el modelo funcional descrito mediante los DFD.

Para establecer una jerarquía de control razonable entre las diferentes operaciones descritas en los DFD, la técnica de diseño estructurado recomienda hacer ciertos análisis de flujo de datos global. Se recomienda realizar los análisis denominados de flujo de transformación y de transacción, que pueden realizarse con mayor facilidad si se modifica parcialmente la estructura jerárquica de los DFD, construyendo un único diagrama con todos los procesos contenidos en los primeros niveles de descomposición, y prescindiendo de los almacenes de información.

- Análisis de flujo de transformación: Consiste en identificar un flujo de global de información desde los elementos de entrada al sistema hasta los de salida. Los procesos se deslindan en tres regiones, denominadas de flujo de entrada, de transformación y de salida. Para obtener la estructura modular del programa se asignan módulos para las operaciones del diagrama y se añaden módulos de coordinación que realizan 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 o transacción distinta, de manera que solo una de estas líneas se activa para cada entrada de datos de tipo diferente. El análisis consiste en identificar el llamado centro de transacción, a partir del cual se ramifican las líneas de flujo, y las regiones correspondientes a cada una de esas líneas o transacciones.

5.3 TÉCNICAS DE DISEÑO BASADO EN ABSTRACCIONES

Surgen cuando se identifican con precisión los conceptos de abstracción de datos y de ocultación. La idea general es que los módulos se correspondan, o bien con funciones, o bien con tipos abstractos de datos.

5.3.1 Descomposición modular basada en abstracciones

Como técnica de programación, consiste en ampliar el lenguaje existente con nuevas operaciones y tipos de datos, definidos por el usuario de forma que se simplifique la escritura de los niveles superiores del programa. Como técnica de diseño, consiste en dedicar módulos separados a la realización de cada tipo abstracto de datos y cada función importante.

En forma descendente puede considerar como una ampliación de la técnica de refinamiento progresivo, mientras que en 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.

5.3.2 Método de Abbott

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, en general, y las operaciones como verbos o como nombres de acciones.

Se puede comenzar subrayando en el texto todos los términos de alguno de los tipos indicados, que sean significativos para la aplicación, y establecer dos listas: una de nombres y otra de verbos u operaciones. El siguiente paso es reorganizar dichas listas extrayendo los posibles tipos de datos y asociándoles sus atributos y operaciones.
Al reorganizar las listas hay que depurarlas eliminando los términos irrelevantes o sinónimos, y completarla con los elementos implícitos en la descripción, pero que no se mencionaban expresamente.

A partir de estas listas iniciales hay que elaborar la descripción de abstracciones, indicando para cada tipo abstracto de datos cuáles son sus atributos y sus operaciones. Para obtener el diseño se puede asignar un módulo

a cada abstracción de datos o grupo de abstracciones relacionadas entre sí. El módulo puede corresponder a un dato encapsulado si solo se maneja un dato de ese tipo en todo el programa. Además, hay que tratar de expresar cada operación en función de las otras, para ver las relaciones de uso entre módulos y detectar posibles omisiones. También hay que añadir abstracciones funcionales para el módulo principal y otras operaciones omitidas, en su caso.

5.4 TÉCNICAS DE DISEÑO ORIENTADAS A OBJETOS

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

5.4.1 Diseño orientado a objetos

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

1. Estudiar y comprender el problema: Puede ser necesario repetirlo, bien porque la especificación no sea suficientemente precisa o porque el diseño va a ser realizado por personas diferentes.

2. Desarrollar una posible solución: Se consideran varias alternativas y se elige la que se considere más apropiada. La solución elegida debe expresarse con suficiente detalle como para que en su descripción aparezcan mencionados casi los elementos que formaran parte del diseño.

3. Formular la estrategia en términos de clases, objetos y relaciones.

   3.1 Identificar las clases y objetos: Puede hacerse siguiendo la técnica de Abbott. Tras este primer paso puede ya confeccionarse un diagrama inicial del modelo de objetos, con las relaciones de composición, asi como otras relaciones generales que se vayan identificando.

   3.2 Identificar las operaciones sobre los objetos: También puede hacerse siguiendo la técnica de Abbott. Hay que decidir a qué objeto o clase se asocian las operaciones.

   3.3 Aplicar herencia: Estas relaciones de herencia se incluirán en el diagrama de modelo de objetos que se va desarrollando.

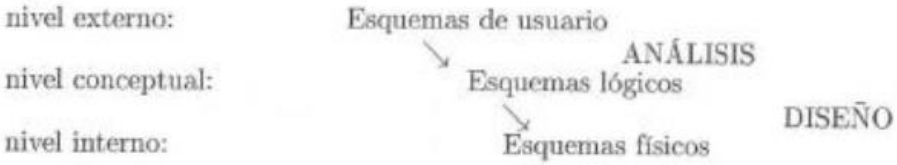
   3.4 Describir las operaciones: Cada operación se describe de manera informal o en pseudocódigo, haciendo referencia únicamente a operaciones o clases de datos en este mismo diseño, o bien predefinidos en el lenguaje de programación a usar y otros elementos de software, ya disponible.

   3.5 Establecer la estructura modular: En principio se intentará que cada módulo corresponda a una clase de objetos. Si el módulo es demasiado complicado, ciertas operaciones pueden establecerse como módulos separados. también es posible agrupar en un solo modulo varios objetos o clases muy relacionados entre sí para mejorar las características de acoplamiento y cohesión. Como resultado de esta etapa se obtendrá el diagrama de estructura del sistema.

Hay que analizar si el diseño modular resultante es apropiado para pasar a la fase de codificación. Si algún modulo es todavía demasiado complejo, o no está definido con suficiente precisión, se repetirán los pasos anteriores de diseño para ese modulo, con objeto de refinarlo.

5.5 TÉCNICAS DE DISEÑO DE DATOS

La organización de la base de datos puede realizarse desde varios puntos de vista. Una forma clásica de enfocarla es en tres niveles:



- Nivel externo: Corresponde a la visión de usuario. La organización de los datos se realiza siguiendo esquemas significativos en el campo de la aplicación.

- Nivel conceptual: Establece una organización lógica de los datos, con independencia del sentido físico que tengan en el campo de la aplicación, lo que se resume en un diagrama de modelo de datos.

- Nivel físico: Organiza los datos según los esquemas admisibles en el sistema de gestión de bases de datos y/o lenguaje de programación elegido para desarrollo. Si se utiliza una base de datos relacional los esquemas físicos serán esquemas de tablas.

5.6 DISEÑO DE BASES DE DATOS RELACIONALES

5.6.1 Formas normales

Estos criterios se numeran correlativamente de menor a mayor nivel de restricción, dando lugar a las formas normales 1ª, 2ª, 3ª, etc. Una tabla que cumpla con cierta forma normal cumple también con las anteriores.

Se dice que una tabla está en 1ª forma normal si la información asociada a cada una de las columnas es un valor único, y no una colección de valores en número variable.

Se dice que una tabla está en 2ª forma normal si está en 1ª forma normal y además hay una clave primaria que distingue cada fila, y cada casilla que no sea de la clave primaria depende de toda la clave primaria.

Se dice que una tabla está en 3ª forma normal si está en 2ª forma normal y además el valor de cada columna que no es clave primaria depende directamente de la clave primaria.

5.6.2 Diseño de las entidades

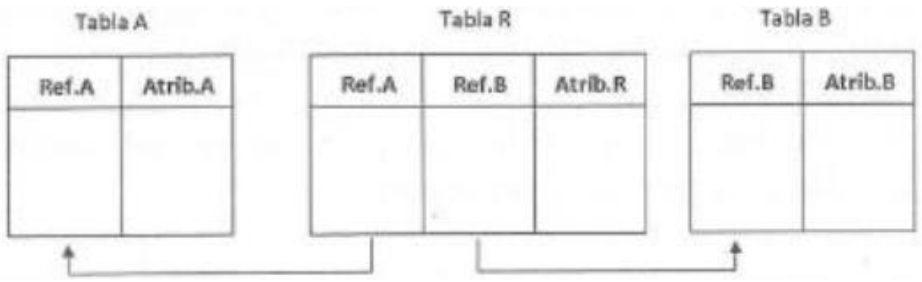
Cada entidad del modelo E-R se traduce en una tabla por cada clase de entidad, con una fila por cada elemento de esa clase y una columna por cada atributo de esa entidad.
Si una entidad está relacionada con otras, se puede incluir una columna conteniendo un código o un numero de referencia que identifique cada elemento de datos. En el modelo de objetos, esto corresponde a almacenar explícitamente en la tabla el identificador del objeto.

5.6.3 Tratamiento de las relaciones de asociación

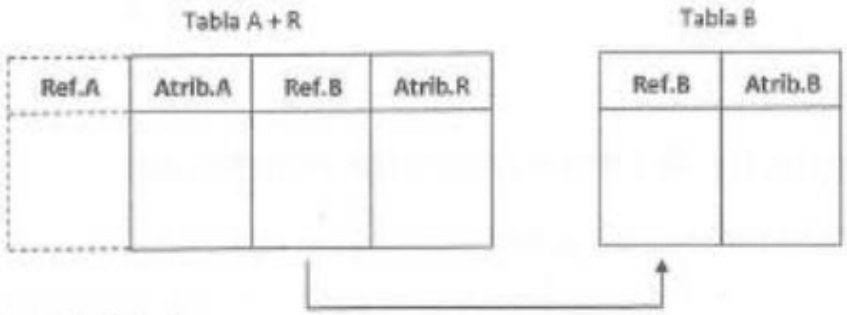
La manera de almacenar en tablas la información de las relaciones de asociación depende de la cardinalidad de la relación. La técnica general es traducir la relación a una tabla conteniendo referencias a las tablas de las entidades relacionadas, asi como los atributos de la relación, si los hay.

Si la cardinalidad es N-N, la referencia a las entidades relacionadas se hará mediante la clave primaria de cada una.

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



Si la cardinalidad es 1-1, se pueden fundir las tablas de las dos entidades en una sola.

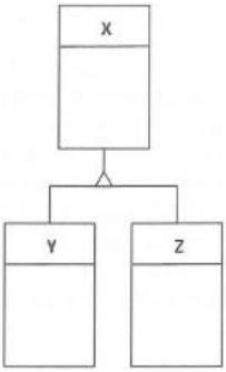


5.6.4 Tratamiento de las relaciones de composición

Las relaciones de composición se tratan de la misma manera que las relaciones asociación. En las relaciones de composición la cardinalidad del lado del objeto compuesto casi siempre es 1.

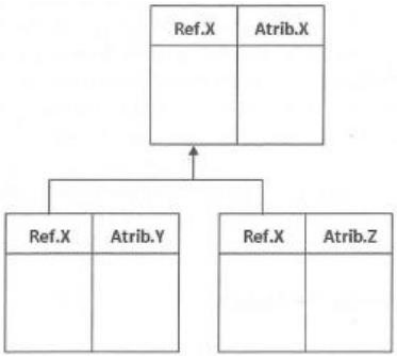
5.6.5 Tratamiento de la herencia

Cuando una clase de objetos tiene varias subclases, si la relación de herencia es:



Se pueden adoptar tres formas de almacenar en tablas la información de las entidades:

- Se usa una tabla para la superclase, con los atributos comunes, heredados por las subclases, más una tabla por cada subclase, con sus atributos específicos.



- Se han repetido los atributos comunes en las tablas de cada subclase, por lo que desaparece la tabla de la superclase.

Ref.Y	Atrib.X	Atrib.Y

Ref.Z	Atrib.X	Atrib.Z

- Se prescinde de las tablas separadas para cada subclase, y se amplía la tabla de la superclase con todos los atributos de cada una de las subclases.

Ref.X	Atrib.X	Atrib.Y	Atrib.Z
	xxxx xxxx	yyyy	zzzz

5.6.6 Diseño de índices

Los índices permiten acceder rápidamente a un dato concreto, reduciendo asi el tiempo de acceso a costa de aumentar el espacio necesario de almacenamiento y el tiempo necesario para almacenar cada nuevo dato.

Si hay que acceder a datos a través de sus relaciones con otros, es conveniente mantener índices sobre las claves primarias y columnas de referencia de las entidades relacionadas.

5.7 DISEÑO DE BASES DE DATOS DE OBJETOS

En las bases de datos de objetos hay una gran variedad de estructuras disponibles per distintas en cada caso. En general pueden adoptarse dos enfoques en el diseño físico en estas bases de datos:

- El primero resulta apropiado cuando la base de datos de objetos permite usar una gran variedad de estructuras. El sistema de gestión de base de datos aporta como complemento la persistencia de los datos.

- El segundo se aplicaría cuando no existe esa variedad de estructuras de datos, y la base de datos de objetos resulta análoga a una base de datos relacional, en que se establece implícitamente una tabla por cada clase de objetos. El sistema de gestión de base de datos aporta la existencia implícita de identificadores de objetos.

5.8 DISEÑO DE SOFTWARE CON PATRONES

Un patrón es una solución probada a un problema recurrente, de tal forma que, cada vez que aparezca dicho problema, se aplica el patrón para resolverlo.

Las dos principales características que debe cumplir un patrón son la efectividad a la hora de resolver el problema descrito y la capacidad para ser reutilizado en multitud de ocasiones. La correcta documentación del patrón es imprescindible para lograr la reutilizabilidad cuya plantilla común está compuesta, entre otros, por los siguientes campos:

- Nombre del patrón.
- Propósito.
- Motivación.
- Aplicabilidad.
- Desarrollo del patrón.
- Usos conocidos.
- Ejemplos.

Las aplicaciones de los patrones dentro de la ingeniera del software son:

- Interfaces de usuario u hombre-computador.
- Patrones de diseño en programación orientada a objetos.
- Patrones para la integración de sistemas heterogéneos que deben comunicarse y sincronizarse.
- Patrones de flujo de datos de la gestión de procesos empresariales.

Dentro de los patrones de diseño en programación orientada a objetos se clasifica a los patrones en tres grupos:

- Patrones creacionales: Resuelven problemas en la creación de instancias.
- Patrones estructurales: Resuelven problemas de composición de clases para dar lugar a otras más complejas.
- Patrones de comportamiento: Resuelven problemas en la interacción, comunicación y responsabilidad entre clases u objetos.

TEMA 6: UML: LENGUAJE UNIFICADO DE MODELADO

6.1 ¿QUÉ ES UML?

El Lenguaje Unificado de Modelado es el lenguaje de modelado de sistemas de software más conocido en la actualidad. Es un lenguaje grafico para visualizar, especificar, construir y documentar un sistema, que ofrece un estándar para describir un plano del sistema incluyendo aspectos conceptuales tales como procesos, y aspectos concretos como expresiones de lenguaje.

Con UML solo se diagrama la realidad de una utilización en un requerimiento. Mientras que, programación estructurada es una forma de programar como lo es la orientación a objetos. UML es un lenguaje de modelado visual que se usa para especificar, visualizar, construir y documentar artefactos de un sistema software. Se usa para entender, diseñar, configurar, mantener y controlar la información sobre los sistemas a construir.

Existen herramientas que pueden ofrecer generadores de código a partir de los diagramas UML para una gran variedad de lenguajes de programación, asi como construir modelos por ingeniería inversa a partir de programas existentes.

6.2 OBJETIVOS DE UML

Durante el desarrollo del UML sus autores tuvieron en cuenta los siguientes objetivos:

- Cubrir las cuestiones relacionadas con el tamaño inherentes a los sistemas complejos y críticos.
- Crear un lenguaje de modelado utilizable tanto por las personas como por las maquinas.
- Conseguir la capacidad de modelar sistemas, desde el concepto hasta los artefactos ejecutables, utilizando técnicas orientadas a objetos.

Mediante el fomento del uso de UML, OMG pretende alcanzar los siguientes objetivos:

- Proporcionar mecanismos de extensión y especialización.
- Ser independiente del proceso de desarrollo y de los lenguajes de programación.
- Proporcionar una base formal para entender el lenguaje de modelado.
- Fomentar el crecimiento de las herramientas orientadas a objetos.
- Soportar conceptos de desarrollo de alto nivel.
- Integrar las mejores prácticas utilizadas hasta el momento.
- Proporcionar a los usuarios un lenguaje de modelado visual expresivo y utilizable para el desarrollo e intercambio de modelos significativos.

6.3 ESTRUCTURA DE UML

- Estructura estática.

  Cualquier modelo preciso debe primero definir los conceptos clave de la aplicación, sus propiedades internas, y las relaciones entre cada una de ellas.
  Los conceptos de la aplicación son modelados como clases, cada una de las cuales describe un conjunto de objetos que almacenan información y se comunican para implementar un comportamiento. La información que almacena es modelada como atributos.

- Comportamiento dinámico.

  Hay dos formas de modelar el comportamiento, una es la historia de la vida de un objeto y la forma como interactúa con el resto del mundo y la otra es por los patrones de comunicación de un conjunto de objetos conectados.
  La visión de un objeto aislado es una máquina de estados que muestra la forma en que el objeto responde a los eventos en función de su estado actual.

- Construcciones de implementación.

  Un componente es una parte física reemplazable de un sistema y es capaz de responder a las peticiones descritas por un conjunto de interfaces. Un nodo es un recurso computacional que define una localización durante la ejecución de un sistema.

- Mecanismos de extensión.

  Un estereotipo es una nueva clase de elemento de modelado con la misma estructura que un elemento existente, pero con restricciones adicionales.

- Organización del modelo.

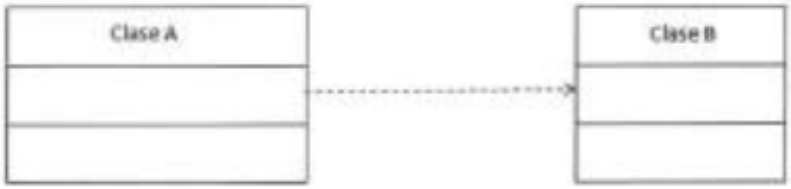
  La información debe ser dividida en piezas coherentes, para que los equipos puedan trabajar en las diferentes partes de forma concurrente. Los paquetes son unidades organizativas, jerárquicas y de propósito general, que pueden usarse para almacenamiento, control de acceso, gestión de la configuración y construcción de bibliotecas.

- Elementos de anotación.

  Son las partes explicativas de los modelos UML. Son comentarios que se pueden aplicar para describir, clasificar y hacer observaciones sobre cualquier elemento de un modelo.

- Relaciones. Existen seis tipos.

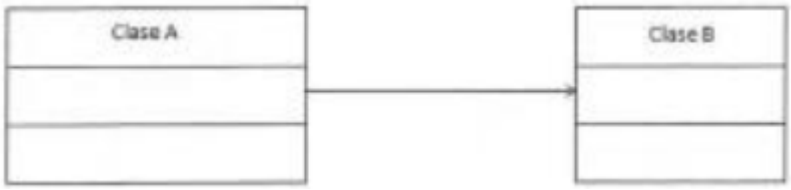
  - Dependencia.



  Es una relación semántica entre dos elementos en la cual un cambio a un elemento puede afectar a la semántica del otro. Se representa como una línea discontinua, posiblemente dirigida, que a veces incluye una etiqueta.

  La clase A usa la clase B.

- Asociación.



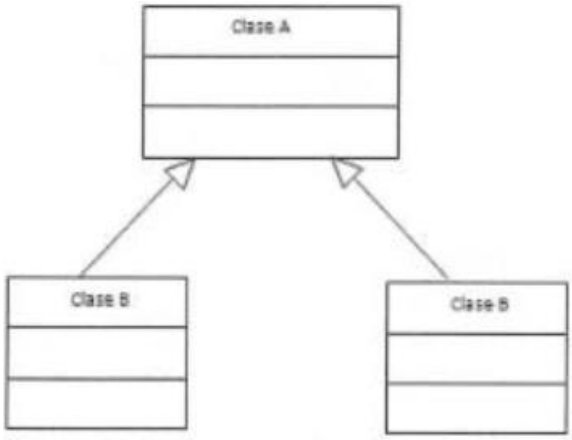
Es una relación estructural que describe un conjunto de enlaces, los cuales son conexiones entre objetos. Se representa con una línea continua, posiblemente dirigida, que a veces incluye una etiqueta.

La multiplicidad de una asociación determina cuantos objetos de cada tipo intervienen en la relación. Cada asociación tiene dos multiplicidades, una para cada extremo de la relación.

Multiplicidad	Significado
1	Uno y solo uno
0…1	Cero o uno
N…M	Desde N hasta M
*****	Cero o varios
0…*	Cero o varios
1…*	Uno o varios (Al menos 1)

La clase A necesita la clase B.

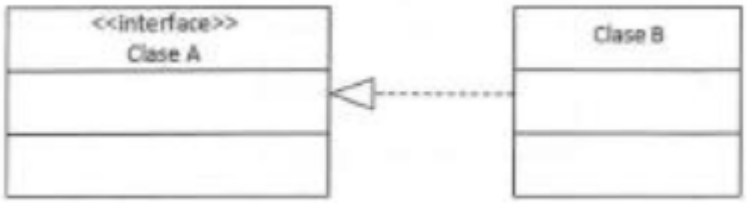
- Generalización.



Es una relación de especialización/generalización en la cual los objetos del elemento hijo pueden sustituir a los objetos del elemento padre. De esta forma el hijo comparte la estructura y el comportamiento del padre. Se representa con una línea con punta de flecha vacía.

La clase A es una generalización de la Bo la C o la clase B y C son especializaciones de la A.

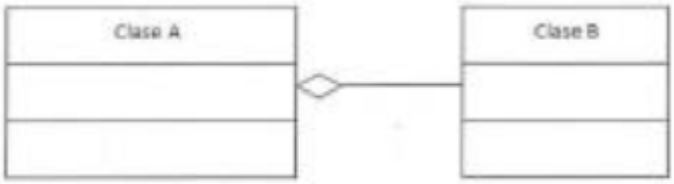
- Realización.



Es una relación semántica entre clasificadores, donde un clasificador especifica un contrato que otro clasificador garantiza que cumplirá. Se representa como una mezcla entre generalización y dependencia, una línea discontinua con una flecha de punta vacía.

La clase B es una realización de la A.

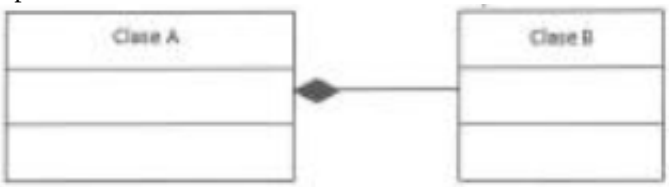
- Agregación.



Es muy similar a la asociación solo que varía en la multiplicidad ya que en lugar de ser uno a uno, es de uno a muchos. Se representa con una flecha que parte de una clase a otra cuya base hay un rombo blanco.

La clase A agrupa varios elementos del tipo B.

- Composición.



Similar a la agregación solo que esta es una relación más fuerte. El tiempo de vida de un objeto está condicionado por el tiempo de vida del objeto que lo incluye.

La clase agrupa varios elementos del tipo B. El tiempo de vida de los objetos de la clase B está condicionado por el tiempo de vida del objeto de la clase A.

- Diagramas.

UML proporciona un amplio conjunto de diagramas que normalmente se usan en pequeños subconjuntos para poder representar las cinco vistas principales de la arquitectura de un sistema.

6.4 DIAGRAMAS UML

6.4.1 Diagramas de casos de uso

Ilustran la funcionalidad proporcionada por una unidad del sistema. Describen las relaciones y las dependencias entre un grupo de casos de uso y los actores participantes en el proceso.

Los diagramas de casos de uso sirven para facilitar la comunicación con los futuros usuarios del sistema, y con el cliente, y resultan especialmente útiles para determinar las características necesarias que tendrá el sistema.

- Actor.

Un actor es una entidad externa que interacción con el sistema participando en un caso de uso. Pueden ser gente real, otros sistemas o eventos.

- Caso de uso.

Un caso de uso describe, desde un punto de vista de los actores, un grupo de actividades de un sistema que produce un resultado concreto y tangible.
Son descriptores de las interacciones típicas entre los usuarios de un sistema y ese mismo sistema. Representan la interfaz externa del sistema y especifican que requisitos de funcionamiento debe tener este.

Es importante tener presentes algunas reglas:

- Cada caso de uso esta relacionado como mínimo con un actor.
- Cada caso de uso es un iniciador (actor).
- Cada caso de uso lleva a un resultado relevante.

Los casos de uso pueden tener relaciones con otros. Los tres tipos de relaciones más comunes entre ellos pueden ser:

- <<include>>: Especifica una situación en la que un caso de uso tiene lugar dentro de otro.
- <<extends>>: Especifica que en ciertas situaciones un caso de uso será extendido por otro.
- Generalización: Especifica que un caso de uso hereda las características del "super" caso de uso.

- Descripción de casos de uso.

Son reseñas textuales del caso de uso. Normalmente tienen el formato de una nota o un documento relacionado de alguna manera con el caso de uso, y explica los procesos o actividades que tienen lugar en el caso de uso.

6.4.2 Diagrama de clases

Muestran las diferentes clases que componen un sistema y como se relacionan unas con otras. Son diagramas estáticos porque muestran las clases, junto con sus métodos y atributos, asi como las relaciones estáticas entre ellas: que clases conocen a que otras clases o que clases son parte de otras.

Una clase define los atributos y los métodos de usa serie de objetos. Todos los objetos de esta clase tienen el mismo comportamiento y el mismo conjunto de atributos.
Las clases están representadas por rectángulos, con el nombre de la clase, y también pueden mostrar atributos y operaciones de la clase en otros dos "compartimentos" dentro del rectángulo.

La clase está formada por:

- Atributos.

Se muestran al menos con su nombre, y también pueden mostrar su tipo y otras propiedades. Aparecen calificados en el diagrama dependiendo de su acceso como:

- +: Indica atributos públicos.
- #: Indica atributos protegidos.
- -: Indica atributos privados.

- Operaciones.

Se muestran al menos con su nombre y pueden mostrar sus parámetros y valores de retorno. Aparecen calificados en el diagrama como:

- +: Indica operaciones públicas.
- #: Indica operaciones protegidas.
- -: Indica operaciones privadas.

- Plantillas.

Son valores usados para una clase no especificada o un tipo. Su tipo es especifica cuando se inicia una clase.

Las clases se pueden relacionar con otras de las siguientes formas:

- Generalización.
- Asociación.
- Realización.
- Agregación.
- Composición.

Los diagramas de clases pueden contener más componentes aparte de clases. Estos pueden ser:

- Interfaces: Son clases abstractas que pueden contener operaciones, pero no atributos. Las clases pueden heredarse de las interfaces.

- Tipo de datos: Son primitivas incluidas en algunos lenguajes de programación. No pueden tener relación con clases, pero las clases si pueden relacionarse con ellos.

- Enumeraciones: Son listas de valores. No pueden relacionarse con las clases, pero las clases si pueden hacerlo con ellos.

- Paquetes: Representan un espacio de nombres. En un diagrama se emplean para representar partes del sistema que contienen más de una clase.

6.4.3 Diagramas de secuencia

Muestran el flujo de mensajes entro objetos para un determinado caso de uso. El propio diagrama explica la secuencia de llamadas que se producen entre los objetos que intervienen.

Los diagramas de secuencia tienen dos dimensiones:

- Vertical: Muestra la secuencia de llamadas ordenadas en el tiempo en el que ocurren.
- Horizontal: Muestra las diferentes instancias de objetos a las que son enviadas las llamadas.

Los objetos están representados por rectángulos con el nombre del objeto dentro y por líneas intermitentes verticales, con el nombre del objeto en la parte más alta. Es conveniente dibujar una flecha de retorno de la llamada.

Los mensajes pueden ser síncronos o asíncronos. Los mensajes síncronos tienen una caja vertical en un lateral del objeto invocante que muestra el flujo de control del programa.

6.4.4 Diagramas de colaboración

Muestran las interacciones que ocurren entre los objetos que participan en una situación determinada. Fijan el interés en las relaciones entre los objetos y su topología.

Los mensajes enviados de un objeto a otro se representan mediante flechas, mostrando el nombre del mensaje, los parámetros y la secuencia del mensaje. Estos diagramas están indicados para mostrar una situación o flujo de programa específicos y son unos de los mejores tipos de diagramas para demostrar o explicar rápidamente un proceso dentro de la lógica del programa.

6.4.5 Diagrama de estado

Muestran los diferentes estados de un objeto durante sui vida, y los estímulos que provocan los cambios de estado en un objeto. La notación de un diagrama de estados tiene cinco elementos básicos:

- Punto de inicio: Dibujado con un circulo relleno.
- Transición entre estados: Dibujado con una línea terminada con punta de flecha abierta.

- Estado: Dibujado con un rectángulo con los vértices redondeados.
- Punto de decisión: Dibujado con un circulo no relleno.
- Puntos de terminación: Dibujados con un circulo con otro relleno en su interior.

6.4.6 Diagrama de actividad

Describen la secuencia de las actividades en un sistema. Son una forma especial de los diagramas de estado, que únicamente contienen actividades. Los diagramas de actividad siempre están asociados a una clase, a una operación o a un caso de uso.

Una actividad es un único paso de un proceso. Es un estado del sistema que tiene actividad interna, y al menos, una o más transacciones salientes.
Las actividades pueden formar jerarquías, por lo que pueden estar formadas de varias actividades.

Los diagramas de actividad son indicados para modelar procesos de alto nivel y modelizar el proceso de negocio de una compañía.

Los elementos para definir un diagrama de actividades son similares a los usados en los diagramas de secuencia:

- Circulo relleno: Inicio.
- Rectángulos con los bodes redondeados: Para determinar las actividades.
- Flechas conectoras: Para unir las actividades.
- Círculos huecos: Puntos de decisión.

Como elemento distinto aparecen las líneas divisorias o calles para establecer las responsabilidades entre los distintos objetos del sistema.

6.4.7 Diagramas de componentes

Muestran los componentes del software y los artilugios de que está compuesto como los archivos de código fuente, las librerías o las tablas de una base de datos.

Ilustran las piezas del software, controladores embebidos, etc. que conformaran un sistema. Un diagrama de componentes tiene un nivel más alto de atracción que un diagrama de clase.

6.4.8 Diagrama de despliegue

Muestra como el sistema se asentará físicamente en el entorno hardware que lo acompaña. Su propósito es mostrar donde los componentes del sistema se ejecutarán y como se comunicarán entre ellos.

TEMA 7: LA CODIFICACIÓN DEL SOFTWARE

7.1 LOS LENGUAJES DE PROGRAMACIÓN

Según la guía Swebok se pueden diferenciar:

- Lenguajes de caja de herramientas (toolkits): Se utilizan para construir aplicaciones a partir de toolkits.
- Lenguajes script.
- Lenguajes de programación.
- Lenguajes de configuración: El ingeniero de software elige un conjunto de parámetros entre una serie de opciones predefinidas.

Solo un número relativamente pequeño de lenguajes de programación se utilizan en el desarrollo de software a escala industrial.

7.1.1 Primera generación de lenguajes

La programación se hacía introduciendo directamente en la memoria del computador el escaso número de instrucciones del programa en códigos binarios por lo que era esencial conocer la estructura interna del computador.
Para simplificar esta labor tan tediosa y disminuir las posibilidades de error se crearon los lenguajes ensambladores, que consisten en asociar a cada instrucción del computador un mnemotécnico que recuerde cuál es su función. Los ensambladores se consideran la primera generación de lenguajes, con un nivel de abstracción muy bajo.

La programación en ensamblador resulta compleja, da lugar a errores difíciles de detectar, exige que el programador conozca la arquitectura del computador y sobre todo se necesita adaptar la solución a las particularidades de cada computador concreto.

7.1.2 Segunda generación

Su característica más notable era que no dependían de la estructura de un computador concreto y por primera vez se programaba en alto nivel de manera simbólica.

Con ellos, se incorporaron los primeros elementos realmente abstractos, como los tipos de datos.

Todavía hoy se utilizan algunos de estos lenguajes o alguno de sus descendientes directos. Algunos de los más importantes son:

- Fortran (Formula Translator): Fue pensado para programar fundamentalmente aplicaciones científicas. Una deficiencia que aún conserva es el manejo casi directo de la memoria mediante la sentencia Common.

- Cobol (Common Business-Oriented Language): Es el lenguaje con el que están escritos casi todos los sistemas para el procesamiento de información.

- Algol: Es el precursor de muchos de los lenguajes de la tercera generación. Es el primer lenguaje que da una gran importancia a la tipificación de datos.

- Basic: Fue pensado para la enseñanza de la programación y es de destacar su sencillez y facilidad de aprendizaje.

7.1.3 Tercera generación

A esta generación pertenecen gran parte de los lenguajes que se utilizan actualmente para el desarrollo de software. Aparece una nueva generación de lenguajes fuertemente tipados, que facilitan la estructuración de código y datos, con redundancias entre la declaración y el uso de cada tipo de dato. Algunos de los lenguajes más importantes son:

- Pascal: Aunque fue diseñado para la enseñanza de programación estructurada, su sencillez y buenas prestaciones hizo que pronto se utilizara para el desarrollo de todo tipo de aplicaciones científico-técnicas. Pascal permite la estructuración del código mediante procedimientos o funciones que se pueden definir de manera recursiva.

- Modula-2: Incorpora la estructura de modulo y queda separada la especificación del módulo de su realización concreta lo que facilita una compilación segura y permite la ocultación de detalles de implementación. Se facilita la aplicación inmediata de la modularidad, abstracción y ocultación e incorpora mecanismos básicos de concurrencia.

- C: Fue diseñado para la codificación del sistema operativo UNIX. Posteriormente ha sido utilizado ampliamente para desarrollar software de sistemas y aplicaciones científico-técnicas de todo tipo.

- Ada: Es el lenguaje patrocinado por el Departamento de Defensa de los EE.UU. para imponerlo como estándar en todos sus desarrollos de sistemas con computador englobado. Los packages de Ada facilitan la codificación de un diseño modular y la aplicación de los conceptos de abstracción y ocultación. Ada permite la definición de elementos genéricos y dispone de mecanismos para la programación concurrente de tareas y la sincronización y cooperación entre ellas.

- Smalltalk: Es el precursor de los lenguajes orientados a objetos. Las estaciones de trabajo con pantallas gráficas y procesadores más potentes permitieron disponer de entornos más amigables y las nuevas versiones del lenguaje alcanzaron mayor difusión.

- C++: Es un lenguaje que incorpora al lenguaje C los mecanismos básicos de la programación orientada a objetos: ocultación, clases, herencia y polimorfismo.

- Java: Su sintaxis deriva mucho de C y C++, pero tiene menos facilidades de bajo nivel que cualquiera de ellos. Es un lenguaje de programación de propósito general, concurrente, orientado a objetos y basado en clases que fue diseñado específicamente para tener tan pocas dependencias de implementación como fuera posible.

- Eiffel: Permite la definición de clases genéricas, herencia múltiple y polimorfismo.

- LISP: Es el precursor de 105 lenguajes funcionales. Aunque actualmente compiten con él un gran número de lenguajes funcionales, continúa siendo uno de los más utilizados en aplicaciones de inteligencia artificial y sistemas expertos. Todos los datos se organizan en listas y para su manejo se emplean funciones recursivas.

- Prolog: Se utiliza fundamentalmente para la construcción de sistemas expertos. Los elementos fundamentales que utiliza son la herencia y las reglas. Ambos constituyen la base de representación del conocimiento.

7.1.4 Cuarta generación

Tratan de ofrecer al programador un mayor nivel de abstracción, prescindiendo por completo del computador. Se trata de lograr que cualquiera sin grandes conocimientos sobre programación de computadores pueda utilizar estos lenguajes para realizar aplicaciones sencillas.

- Bases de datos: Permiten acceder y manipular la información de la base de datos mediante un conjunto de ordenes de petición. La ventaja fundamental es que dotan de una gran versatilidad a la base de datos y permiten que pueda ser el propio usuario quien diseñe sus propios elementos cuando los necesite.

- Generadores de programas: Con ellos se pueden construir elementos abstractos fundamentales en un cierto campo de aplicación sin descender a los detalles concretos que se necesitan en los lenguajes de la tercera generación. La mayoría de los generadores de programas sirven para realizar aplicaciones de gestión y el lenguaje destino es Cobol.

- Calculo: Con ellos se puede programar prácticamente cualquier problema dentro de un determinado campo.

7.2 CRITERIOS DE SELECCIÓN DEL LENGUAJE

A priori, en la decisión debería prevalecer los criterios técnicos, pero existen otros factores de tipo operativo que deben ser tenidos en cuenta. Algunos de los criterios son los siguientes:

- Imposición del cliente: En algunos casos es el cliente el que fija el lenguaje que se debe utilizar. En otras ocasiones el cliente no es tan drástico y simplemente establece una relación reducida de lenguajes que se pueden usar en sus desarrollos.

- Tipo de aplicación: Para aplicaciones de gestión lo habitual es usar Cobol. En aplicaciones del área científico-técnica se usa Fortran. Para aplicaciones de sistemas y tiempo real se usan C, Modula-2 y Ada. Para inteligencia artificial se usan LISP y Prolog. Para aplicaciones con un diseño orientado a objetos se usan C++, Java o Eiffel.

- Disponibilidad y entorno: Es necesario comprobar que compiladores existen para el computador elegido. Se debe tener en cuenta si el código generado se ejecuta directamente o se interpreta posteriormente.

- Experiencia previa: Siempre que sea posible es más rentable aprovechar la experiencia previa. Cuando las condiciones de trabajo no se modifican el rendimiento aumenta y se disminuyen las posibilidades de error.

- Reusabilidad: Este aspecto es importante tanto por la posibilidad de utilizar software ya realizado como por el hecho de dejar disponibles para otros proyectos parte del software desarrollado.

- Transportabilidad: Los desarrollos se realizan para un computador concreto, pero a lo largo del tiempo este se queda obsoleto, por lo que es habitual que se traten de trasladar las aplicaciones de unos computadores a otros.

- Uso de varios lenguajes: Hay ocasiones en las que las distintas partes del mismo lenguaje resultan mucho más sencillas de codificar si se utilizan diferentes lenguajes.

7.3 ASPECTOS METODOLÓGICOS

7.3.1 Normas y estilos de codificación

Es fundamental fijar las normas que deberán respetar todos los programadores para que el resultado del trabajo de todo el equipo sea completamente homogéneo.

Para la puesta a punto y el mantenimiento de un programa es esencial que este resulte fácil de entender por todos. Por tanto, se deben fijar normas concretando un estilo de codificación. Lo importante es fijar un estilo concreto y que todo el equipo lo adopte y lo respete. Para fijar un estilo de codificación se deben concretar al menos los siguientes puntos:

- Formato y contenido de las cabeceras de cada módulo: Identificación y descripción del módulo, autor y fecha, revisiones… etc.

- Formato y contenido para cada tipo de comentario: Sección, orden, al margen… etc.

- Utilizacion de encolumnado: Tabulador, máximo indentado, formato selección, formato iteración… etc.

- Elección de nombres: Convenio para el uso de mayúsculas y minúsculas, nombres de ficheros, identificadores de elementos del programa… etc.

7.3.2 Manejo de errores

Durante la ejecución de un programa se pueden producir fallos que tienen como origen las más diversas causas:

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

Los conceptos básicos a tener en cuenta son:

- Defecto: Errata de software. Puede permanecer oculto durante un tiempo indeterminado, si los elementos defectuosos no intervienen en la ejecución del programa.

- Fallo: Es el hecho de que un elemento del programa no funcione correctamente, produciendo un resultado erróneo.

- Error: Se define como un estado inadmisible de un programa al que se llega como consecuencia de un fallo.

Al codificar un programa se pueden adoptar distintas actitudes respecto al tratamiento de los errores:

- No considerar los errores: Con esta actitud se declina toda responsabilidad si se produce algún error. Cuando no se cumpla alguno de los requisitos, no será posible garantizar el resultado correcto del programa.

- Prevención de errores: Consiste en detectar los fallos antes de que provoquen un error. La técnica general de prevención se denomina programación a la defensiva y consiste en que cada programa o subprograma este codificado de manera que desconfíe sistemáticamente de los datos o argumentos que se le introducen, y devuelva:

  - El resultado correcto si los datos son válidos.
  - Una indicación precisa del fallo si los datos no son válidos.

  Una ventaja de la programación a la defensiva es evitar la propagación de errores facilitando el diagnostico de los defectos.

- 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. Este tratamiento exige dos actividades:

  - Detección del error: Hay que concretar que situaciones se consideraran erróneas y realizar las comprobaciones adecuadas en ciertos puntos estratégicos del programa.

  - Recuperación: Existen dos esquemas generales:

    - Recuperación hacia delante: Trata de identificar la naturaleza del error para tomar las acciones adecuadas que corrijan el estado del programa.

    - Recuperación hacia atrás: Trata de corregir el estado del programa restaurándolo a un estado correcto anterior a la aparición del error. Con este esquema se necesita guardar periódicamente el ultimo estado correcto del programa. Se utiliza habitualmente en sistemas basados en transacciones.

7.3.3 Aspectos de eficiencia

La potencia de cálculo y la cantidad de memoria disponible en los computadores actuales permite asegurar que prácticamente nunca será necesario sacrificar la claridad en la codificación por una mayor eficiencia. La eficiencia se puede analizar desde varios puntos de vista:

- Eficiencia en memoria: Cuando el volumen de información a manejar sea excesivamente grande para la memoria disponible, será durante la dase de diseño detallado cuando se deban estudiar las distintas alternativas y optar por el algoritmo que optimice más la utilización de la memoria.

- Eficiencia en tiempo: En ocasiones una mayor eficiencia en tiempo se logra disminuyendo la eficiencia en memoria.

La primera vía para conseguir un ahorro de tiempo importante es realizar en la fase de diseño detallado un estudio exhaustivo de las posibles alternativas del problema y adoptar el algoritmo más rápido. Existen formas bastante simples de obtener un ahorro de tiempo significativo utilizando técnicas de codificación tales como:

- Tabular los cálculos complejos.

- Expansión en línea empleando macros en lugar de subrutinas.
- Desplegado de bucles: Si se repite X veces el código interno de un bucle, las evaluaciones tardaran 1/X.
- Implicar las expresiones aritméticas y logicas.
- Sacar fuera de los bucles lo que no sea necesario repetir.
- Utilizar estructuras de datos de acceso rápido.
- Evitar las operaciones en coma flotante y realizarlas preferiblemente en coma fija.
- Evitar las conversiones innecesarias de tipos de datos.

7.3.4 Transportabilidad de software

La transportabilidad permite usar el mismo software en distintos computadores actuales y futuros. Por tanto, es rentable a medio y largo plazo para facilitar el mantenimiento/adaptación de la aplicación a las nuevas arquitecturas y prestaciones.

Los factores esenciales de la transportabilidad son:

- Utilizacion de estandares: Un producto software desarrollado exclusivamente sobre elementos estándar es teóricamente transportable sin ningún cambio, al menos entre plataformas que cuenten con el soporte apropiado de dichos estandares.

- Aislar las peculiaridades: La mejor manera es destinar un módulo específico para cada una de ellas. El transporte se resolverá recodificando y adaptando solamente estos módulos específicos al nuevo computador. Las peculiaridades fundamentales suelen estar vinculadas a los siguientes elementos:

  - Arquitectura del computador: La arquitectura determina la longitud de la palabra y de esto se deriva la representación interna de los valores enteros y reales. Para resolver el desbordamiento es necesario definir un nuevo tipo abstracto de dato con el rango o precisión ampliados y crear para él todas las operaciones necesarias mediante funciones.
  La tabla de códigos de caracteres utilizados es otra causa de problemas. Actualmente, todos los computadores utilizan ASCII. Sin embargo, para facilitar la transportabilidad lo mejor es no aprovechar nunca en la codificación el orden de los caracteres en una tabla concreta.

  - Sistema operativo: Los lenguajes incorporan el acceso a servicios del sistema operativo para realizar tareas como:

    - Entrada/salida.
    - Manejo de ficheros.
    - Manejo de librerías.

  Los lenguajes de alto nivel disponen de procedimientos o funciones predefinidos para la realización de las operaciones más elementales dentro de estas tareas y siempre que sea posible deben ser las únicas que se utilicen.
  Lo habitual es que se necesiten operaciones más complejas y particularizadas que las predefinidas. En este caso, se deben concretar y especificar claramente cada una de ellas. Para cada módulo y operación se definirá una interfaz única y precisa en toda la aplicación. el resto de la aplicación utilizara estos módulos de forma independiente del sistema operativo.

TEMA 8: PRUEBAS DE SOFWARE

8.1 TIPOS DE PRUEBAS

Las pruebas de software tienen un doble objetivo:

- Verificación: Persigue comprobar que se ha construido el producto correctamente.
- Validación: Comprueba que el producto se ha construido de acuerdo con los requerimientos del cliente.

Estos procesos llevan a realizar pruebas a diferentes niveles en los que se revisa la calidad de cada etapa del proceso de elaboración del software:

- Pruebas de unidades.
- Pruebas de integración.
- Pruebas de validación.
- Pruebas de sistema.

8.2 PRUEBAS DE UNIDADES

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

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

Con las pruebas se trata de alcanzar un compromiso para que, con el menor esfuerzo posible, se puedan detectar el máximo número de defectos.

Para garantizar unos resultados fiables es esencial que todo el proceso de prueba se realice de la manera más automática posible, lo que exige crear un entorno de prueba que asegure unas condiciones predefinidas y estables para las sucesivas pasadas.

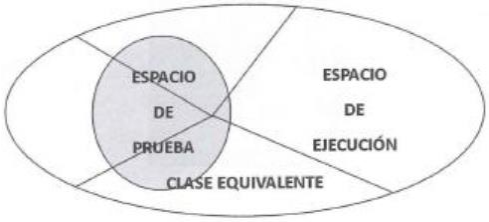
Para establecer el entorno de pruebas, serán suficientes las utilidades del sistema operativo, preparando en un fichero los casos de prueba y recogiendo en otro fichero los resultados obtenidos.

8.2.1 Pruebas de caja negra

Esta estrategia ignora por completo la estructura interna del programa y se basa exclusivamente en la comprobación de la especificación entrada-salida del software. Esta es la única estrategia que puede adoptar el cliente o cualquier persona ajena al desarrollo del programa.

Existen ciertos métodos, basados en la experiencia, que ayudan bastante en la elaboración de casos de prueba. Algunos de los más usados son:

- Partición en clases de equivalencia.



Se trata de dividir el espacio de ejecución del programa en varios subespacios en el que cada uno agrupa a todos aquellos datos de entrada al módulo que resultan equivalentes desde el punto de vista de la prueba de caja negra.

Los pasos que se deben seguir son los siguientes:

1. Definir una clase de equivalencia.
2. Definir una prueba que cubra tantos casos validos como sea posible de cualquier clase.
3. Marcar las clases cubiertas y repetir el paso anterior.
4. Definir una prueba específica para cada caso invalido.

- Análisis de valores límite.

Este método hace hincapié en las zonas del espacio de ejecución que están próximas al borde. Los errores en los límites pueden tener consecuencias más graves de lo normal, debido a que puede provocar la necesidad de unos recursos extra que no estaban previstos.

En este caso, las directrices a seguir en la elaboración de casos de prueba son:

- Entradas: Probar con los mismos valores límite y justo fuera de límites.
- Salidas: Probar con los mismo valores límite y justo fuera de límites.
- Memoria: Probar con tamaños nulos, limite y superior al límite de todas las estructuras.
- Recursos: Probar con ningún recurso, límite de recursos y superior al límite.

- Comparación de versiones.

Cuando una unidad o modulo es especialmente crítico se puede hacer un desarrollo multi-versión encargando la codificación de diferentes versiones a distintos programadores que utilizaran las mismas especificaciones y deberían obtener otros módulos completamente intercambiables.

A continuación, se someterán todas las versiones al mismo juego de casos de prueba de una forma completamente automática. Cualquier discrepancia entre las distintas versiones se debe analizar hasta discernir si una versión es errónea y sus causas.

La redundancia que se introduce con la codificación de varias versiones aumenta las garantías de que el módulo funcione correctamente y que cumpla las especificaciones.

- Empleo de intuición: Es muy importante dedicar cierto tiempo a preparar pruebas que planteen situaciones especiales y que puedan provocar algún error. Para ello, las personas ajenas al desarrollo del módulo suelen aportar un punto de vista mucho más distante y fresco que las que participan en él.

8.2.2 Pruebas de caja transparente

Se conoce y se tiene en cuenta la estructura interna del módulo. En la elaboración de los casos de pruebas se trata de conseguir que el programa transite por todos los posibles caminos de ejecución y ponga en juego todos los elementos del código.

Los casos de prueba deben conseguir que:

- Todas las decisiones se ejecuten en uno y otro sentido.
- Todos los bucles se ejecuten en los supuestos más diversos.
- Todas las estructuras de datos se modifiquen y consulten alguna vez.

Cuando se elaboran las pruebas de caja negra pueden quedar perfectamente inexplorados caminos que en un funcionamiento habitual no serán muy frecuentados, pero que, si son decisivos en situaciones concretas, y que, si no han sido probados convenientemente, pueden producir un error en el peor momento.

Los métodos más ampliamente usados en la elaboración de pruebas de caja transparente son:

- Cubrimiento lógico.

Se debe conseguir cierto cubrimiento lógico de todas las secciones de código, y no dejar ninguna sin ejecutar alguna vez.

Dado un fragmento de código se denomina camino básico a cualquier recorrido que, siguiendo las flechas de las líneas de flujo, permita ir desde el punto inicial al final.

Este método consiste en elaborar casos de prueba para que el programa recorra un determinado conjunto de camino siguiendo ciertas pautas. Para ello se pueden establecer distintos niveles de cubrimiento:

- Nivel I: Se trata de elaborar casos de prueba para que se ejecuten al menos una vez todos los caminos básicos, cada uno de ellos por separado.

- Nivel II: Se trata de elaborar casos de prueba para que se ejecuten todas las combinaciones de caminos básicos por parejas.

- Nivel III: Se trata de elaborar casos de prueba para que se ejecuten un número significativo de las combinaciones posibles de caminos.

Como mínimo las pruebas de cada módulo deben garantizar el nivel I.

- Pruebas de bucles.

- Bucle con un número no acotado de repeticiones: Se elaborarán pruebas para: Ejecutar el bucle 0, 1, 2, un numero moderado y un número elevado de veces.

- Bucle con numero M de repeticiones: Se elaborarán pruebas para: Ejecutar el bucle 0, 1, 2, Un numero intermedio, M – 1, M y M + 1 veces.

- Bucles anidados: El número de pruebas crece de forma geométrica con el nivel de anidamiento. Para reducir este número se utiliza la siguiente técnica:

1. Se ejecutan los bucles externos en su número mínimo de veces.
2. Para el siguiente nivel de anidamiento se ejecutan los bucles externos en su número mínimo de veces y los bucles internos un numero típico de veces.
3. Repetir el paso 2 hasta completar los niveles.

- Empleo de la intuición: También con la estrategia de la caja transparente merece la pena dedicar un cierto tiempo a elaborar pruebas que solo por intuición se pueden estimar que plantearan situaciones especiales.

8.2.3 Estimación de errores no detectados

Pata tener una estimación del número de defectos que quedan sin detectar se puede utilizar la siguiente estrategia:

1. Anotar el número de errores que se producen inicialmente con el juego de casos de prueba (EI).
2. Corregir el módulo hasta que no contenga ningún error con el mismo juego de casos de prueba.
3. Introducir aleatoriamente en el módulo un número razonable de errores (EA).
4. 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 (ED).

$$EE = (EA – ED) * (EI/ED)$$

5. Para el juego de casos de prueba considerado, el porcentaje de errores sin detectar será el mismo para los errores iniciales que para los errores deliberados.

8.3 PRUEBAS DE UNIDADES EN PROGRAMACIÓN ORIENTADA A OBJETOS

Las secuencias de prueba de una clase se diseñan para probar las operaciones más relevantes de ella, y es examinando los valores de los atributos de la clase donde se comprueba si existen errores. Existen varios métodos para probar una clase:

- Pruebas basadas en fallo: El objetivo es descubrir fallos que durante el diseño se hayan detectado o valorado como probables.

- Pruebas aleatorias: Se eligen de forma aleatoria una serie de métodos dentro de la clase para probarlos y se diseñan las secuencias de prueba correspondientes.

- Pruebas de partición: Se pueden crear una serie de categorías en las entradas y salidas para probar una clase de forma selectiva. Las particiones se pueden hacer con distintos criterios:

  - En base a modificar un determinado atributo.
  - En base a los atributos que utiliza cada operación.

8.4 ESTRATEGIAS DE INTEGRACIÓN

Las unidades de un producto software se han de integrar para conformar el sistema completo. Desgraciadamente, en esta fase también aparecen nuevos errores debido a:

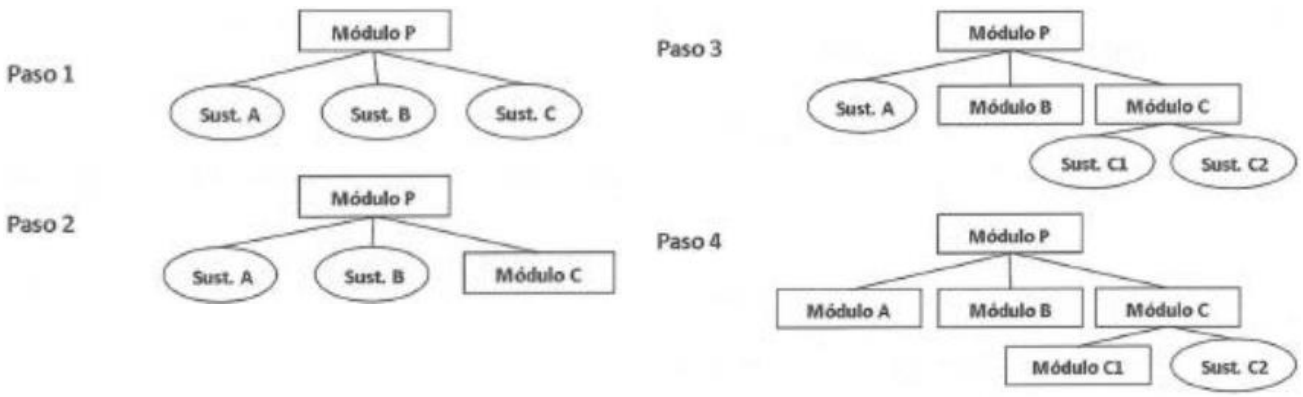
- Desacuerdos de interfaz.
- Interacción indebida entre módulos.
- Imprecisiones acumuladas.

Durante la fase de integración se debe proceder en forma sistemática, siguiendo una estrategia bien definida, para facilitar la depuración de los errores que vayan surgiendo.

8.4.1 Integración Big Bang

Consiste en realizar la integración de todas las unidades en un único paso. Esta estrategia solo se puede justificar en sistemas pequeños. La ventaja fundamental es que se evita la realización del software de andamiaje que se requiere con las otras estrategias.

8.4.2 Integración descendente



En esta estrategia se parte inicialmente del módulo principal que se prueba con módulos de andamiaje de los otros módulos usados directamente. Los módulos sustitutos se van reemplazando, uno por uno, por los verdaderos y se realizan las pruebas de integración correspondientes.

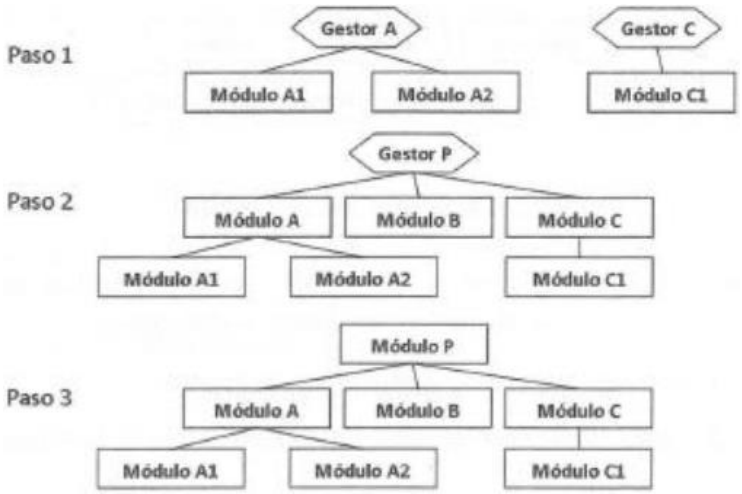
La codificación de los sustitutos es un trabajo adicional que conviene simplificar al máximo y para el que se pueden adoptar distintas soluciones:

- No hacer nada y que sirva para comprobar la interfaz.
- Imprimir un mensaje de seguimiento con información de la llamada.
- Suministrar un resultado fijo.
- Suministrar un resultado aleatorio.
- Suministrar un resultado tabulado u obtenido con un algoritmo simplificado.

La ventaja fundamental de la integración descendente es que se ven desde el principio las posibilidades de la aplicación. Los inconvenientes más importantes son:

- La integración descendente limita en cierta forma el trabajo en paralelo.
- Al concluir la integración de los nuevos módulos desde otros módulos ya integrados y definitivos, se tienen bastantes limitaciones para hacer pruebas especiales o dirigidas a un objetivo específico.

8.4.3 Integración ascendente



Se empieza a codificar por separado y en paralelo todos los módulos de nivel más bajo. Los gestores se van sustituyendo uno a uno por los módulos de mayor nivel según se vayan codificado, al tiempo que se van desarrollando nuevos gestores si hace falta.

Las ventajas de esta integración coinciden con los inconvenientes de la integración descendente:

- Facilita el trabajo en paralelo.
- Facilita el ensayo de situaciones especiales.

El inconveniente más importante es que resulta difícil ensayar el funcionamiento global del producto software hasta el final de su integración.

8.4.4 Estrategias de integración en programación orientada a objetos

Existen dos estrategias para la integración de software orientado a objetos:

- Prueba basada en hebra: Se integran todas las clases necesarias para responder a una determinada entrada o evento del sistema. Cada hebra se integra y prueba de forma independiente.

- Prueba basada en uso: Se comienza construyendo y probando las clases llamadas independientes. A continuación, se prueban las clases dependientes, que usan clases independientes.

Los módulos gestores y sustitutos también se utilizan, pero de forma diferente a la integración convencional.

8.5 PRUEBAS DE VALIDACIÓN

La validación de software se consigue a través de una serie de pruebas que comprueban la conformidad con los requerimientos en cuanto a rendimiento, ergonomía, documentación y funcionalidad.

Para un desarrollo de software es imposible prever como usara el cliente realmente el programa. Algunos problemas típicos son:

- Mensajes ininteligibles para el usuario.
- Presentación inadecuada de resultados.
- Deficiencias en el manual de usuario.
- Procedimientos de operación inusuales.

Es probable que los problemas más graves aparezcan ya al comienzo y por ello es aconsejable que alguien del equipo de desarrollo acompañe al usuario durante la primera toma de contacto.

8.6 PRUEBAS DEL SISTEMA

Tienen como objetivo probar el sistema completo basado en computadora. Diferentes tipos de pruebas encaminadas a asegurarse que el software se ha integrado de forma correcta con su entorno son:

- Pruebas de recuperación: Sirven para comprobar la capacidad del sistema para recuperarse antes de fallos.

- Pruebas de seguridad: Sirven para comprobar los mecanismos de protección contra un acceso o manipulación no autorizada.

- Pruebas de resistencia: Sirven para comprobar el comportamiento del sistema ante situaciones excepcionales.

- Pruebas de sensibilidad: Sirven para comprobar el tratamiento que da el sistema a ciertas singularidades de algoritmos matemáticos.

- Pruebas de rendimiento: Sirven para comprobar las prestaciones del sistema que son críticas en tiempo.

- Pruebas de despliegue o de configuración: Sirven para comprobar el funcionamiento de software que debe ejecutarse en varias plataformas y sistemas operativos distintos.