INGENIERÍA DE SOFTWARE

Jose Antonio Ceballos Leva

2º Grado en ingeniería Informática [UNED]

Índice

BLOQUE I. INTRODUCCIÓN A INGENIERIA DE SOFTWARE Y DOCUMENTOS DE REQUISITOS

Índice

Tema 1. Introducción a la ingeniería de software

1. Concepto de ingeniería de sistemas
2. Concepto de ingeniería de software
3. Formalización del proceso de desarrollo
4. Uso de prototipos
5. Modelo en espiral
6. Mantenimiento o explotación del software
7. Garantía de calidad de software

Concepto de ingeniería de sistemas

Concepto de sistema

Un sistema se define como un conjunto de cosas que ordenadamente relacionadas entre sí contribuyen a una determinada finalidad. Requiriendo la generación de sistemas complejos de un trabajo colectivo, que requiere de una labor de organización, la cual se estudia en la ingeniería de sistemas, la cual se centra en los aspectos de organización como en lo referente al desarrollo.

Sistemas basados en un computador

Son sistemas formados por uno o varios computadores , los cuales se dedican a tareas de control con un propósito final. A nivel informáticos podemos distinguir los elementos materiales, como hardware de los programas que gobiernan el funcionamiento y a los que denominamos software.

Componentes hardware, software y humanos

Las tareas de tratamiento de la información desarrolladas por los sistemas informáticos se dividen en: almacenamiento, elaboración y presentación de datos. Siendo los encargados de la realización de estas funciones las siguientes fuentes: componentes físicos o hardware, componentes programados o software y usuarios del sistema.

La concepción final del sistema informático se podría definir como la selección de hardware, software deben ser adquiridos y desarrollados y de la asignación o reparto de las actividades de tratamiento de información entre cada una de las posibles fuentes de tratamiento.

Concepto de ingeniería de software

Perspectiva histórica

Mientras que en las décadas iniciales el desarrollo de software se ejercía de una manera artesanal y poco disciplinada, el aumento de la capacidad del hardware aumenta las necesidad de programas más complejos, que provocan la necesidad del trabajo en equipo, la separación de las tareas, la especialización y el empleo de herramientas que agilizaran las actividades repetitivas y triviales.

Esto da paso a la aparición de las metodologías de desarrollo específicas con la finalidad de promover la especialización y siendo especialmente aplicadas a los sistemas de información. Estas se desarrollan durante los años 70, empezando a ser empleadas en los 80, con la aparición de las herramientas CASE, las cuales apoyaban las actividades anteriores a la programación o codificación, siendo en los años 90 cuando se amplia el campo de aplicación de las herramientas de ingeniería de software, reconociendo la necesidad de emplear herramientas que cubran todo el ciclo productivo de software.

Mitos del software

Algunos de los mitos mas extendidos serían: el hardware es más importante que el software, el software es fácil de desarrollar, el software solo consiste en programas ejecutables, el software es solo una tarea de programación y es normal que el software contenga errores.

Formalización del proceso de desarrollo

Modelo en cascada

Es el modelo mas primitivo de ciclo de vida, pero imprescindible debido a que ya integra todas las clases de actividades distintas que intervienen en el desarrollo y explotación. Existen diferentes variantes de este modelo que se diferencian por el reconocimiento o no como fases separadas de diferentes actividades. Siendo las principales fases:

- **Análisis**: Consiste en analizar las necesidades de los futuros usuarios del software, para determinar las funciones a desarrollar por el susodicho sistema.
- **Diseño:** Consiste en descomponer y analizar el sistema en subsistemas que permitan ser desarrollados por separado, siendo el resultado del diseño la colección de especificaciones de cada elemento componente.
- **Codificación**: Durante esta fase se programa cada elemento por separado, haciéndose algunas pruebas que puedan garantizar el correcto funcionamiento del sistema.
- **Integración**: Este proceso consiste en la combinación de todos los elementos que componen el sistema, permitiendo la comprobación del sistema completo, siendo necesario hacer pruebas exhaustivas que garanticen el correcto funcionamiento del conjunto.
- **Mantenimiento:** Esta fase engloba todos los necesarios cambios adicionales tras la publicación de aplicación con el fin de corregir errores existentes o mejorar el sistema con funciones adicionales.

Este sistema busca la relativa independencia de cada uno de los procesos, lo que permite un mejor reparto de las actividades. Para conseguir esta independencia es muy importante que cada fase genere a su salida una información precisa y suficiente para que los equipos dependientes de esta puedan ejercer su función, siendo los documentos producidos los siguientes:

- **Documento de requisitos (SRD):** Es producto del análisis y consiste en una especificación precisa y completa de lo que debe hacer el sistema, prescindiendo de detalles técnicos.
- **Documento de diseño de software (SDD):** Es producto del diseño y consiste en una descripción de la estructura global del sistema y la distribución de tareas.
- **Código fuente**: Es producto de la fase de codificación, conteniendo los programas fuente debidamente comentados, lo que permite obtener una claridad y fácil intercambio entre componentes.
- **El sistema de software**: Es producto fase de integración consistente en el ejecutable y siendo necesarias integrarse las pruebas realizadas al software completo.
- **Documentos de cambios**: Es producto de la fase de mantenimiento e incluye los datos de las modificaciones realizadas durante la reparación de cada problema encontrado o descripción de las soluciones aplicadas a las nuevas funciones.

Modelo en V

Este modelo se basa en una secuencia de fases similar al modelo en cascada pero con la particularidad de dar una mayor importancia a la jerarquización, con lo que cuando avanza el desarrollo de la aplicación determinadas fases se va trabajando con mayor o menor detalle. De tal forma que en las fases iniciales el nivel de talle es menor, el cual va aumentando según avanza el proyecto, ya que el proyecto se va descomponiendo hasta llegar a las sentencias en el lenguaje que corresponda tras lo que vuelve a disminuir el detalle en pos de la integración del sistema, hasta que disponemos de un sistema completo. Al igual que en el modelo en cascada, existen diferentes versiones de este, caracterizadas por reconocer o no determinadas divisiones de las actividades.

En las actividades situadas en un nivel determinado se trabaja sobre una unidad de detalle superior, que se organiza en varias unidades del nivel de detalle inferior. Por ejemplo, durante la codificación se trabaja con módulos que se organizan mediante sentencias de un lenguaje, mientras que durante las fases de diseño e integración se trabaja con un sistema de software o se construye con varios módulos. De lo que se entiende que el resultado de una fase no sólo sirve como entrada a la siguiente, sino que en fases posteriores debe utilizarse como documento de referencia para corroborar el correcto progreso de la aplicación.

Uso de prototipos

Definición

Un prototipo es un sistema auxiliar que permite probar experimentalmente ciertas soluciones a las necesidades del usuario o a los requisitos del sistema, siendo el costo de este sensiblemente inferior al del sistema final, pues los errores cometidos no son demasiados costosos, ya que su incidencia se encuentra limitada por el costo total del desarrollo del prototipo, siendo inferior gracias a que parte de este es aprovechable para el resto del desarrollo. Algunas opciones para reducir los costes de este son:

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

Prototipos rápidos

Estos prototipos tienen como finalidad adquirir experiencia, sin pretender aprovecharlos como producto, aquellos con la funcionalidad más limitada se los denomina maquetas. Su función es importante durante las fases de análisis y diseño permitiendo experimentar las distintas posibilidades y garantizar que las decisiones que se toman son las correctas, pero teniendo en cuenta que la codificación del sistema final se hará partiendo de cero. La más importante cualidad de estos es que se desarrollan rápidamente, con el objetivo de intentar alargar lo mínimo las fases de análisis y diseño.

Prototipos evolutivos

Otra forma de utilizar los prototipos consiste en aprovechar al máximo su código, de manera que este sea una realización parcial del sistema final deseado. Este prototipo se construye tras unas fases parciales de análisis y diseño, para a continuación irlo convirtiendo en el producto final. De manera que se van construyendo sucesivas versiones del prototipo inicial cada vez mas complejas.

Esta forma de desarrollo funcionaría como un bucle reiterativo del modelo en cascada, de manera que cada nueva versión del prototipo se basa en lo realizado en la versión anterior a este.

Herramientas para la realización de prototipos

Puesto que un prototipo es una versión del sistema a desarrollar emplearemos unas herramientas semejantes a las utilizadas en desarrollo de software, mientras que para el software de usar y tirar se emplearán herramientas diferentes de las que se emplearán en el producto final, con el fin de su producción sea más barata y/o rápida, por otro lado para el prototipo evolutivo las herramientas serán las mismas que para el final.

Entre las herramientas utilizadas están las de 4ª generación, las cuales se emplean para sistemas de información con una base de datos de uso general y utilizan lenguajes especializados en la interfaz de usuario, basados en menús y formularios de entrada y salida, siendo utilizadas en muchos casos en el sistema final. En caso de no ser empleadas por razones de eficiencia o compatibilidad, siempre pueden ser empleadas en la construcción de un prototipo previo.

Los lenguajes de 4ª generación son un caso de lenguajes de alto nivel con un estilo declarativo en lugar de operacional, con lo que permiten describir el resultado que se desea obtener, en vez de describir las operaciones para alcanzarlo, siendo algunos ejemplos Smalltalk y Prolog. Análogos a estos tenemos los lenguajes de especificación, los cuales tienen como objetivo formalizar la especificación de los requisitos del software, entre los que se encuentran VDM y Z. El empleo de estos nos permitirá obtener directamente un prototipo ejecutable.

Modelo en espiral

Definición

El modelo en espiral consiste en un refinamiento del modelo evolutivo, introduciendo como distintivo la actividad de análisis de riesgo como elemento fundamental para la evolución del desarrollo, lo que provoca que la iteración del modelo produzca una espiral al añadir como dimensión radial una indicación del esfuerzo total realizado en cada momento, que será siempre un valor creciente. Las diferentes actividades se representan sobre unos ejes cartesianos con una actividad en cada cuadrante.

Componentes

Las actividades de planificación sirven para establecer el contexto del desarrollo y decidir que parte del mismo se abordará en ese ciclo de espiral.

El análisis del riesgo consiste en evaluar diferentes alternativas para la realización de la parte de desarrollo seleccionada por ser la mas ventajosa a la vez que se toman precauciones para evitar los inconvenientes previstos.

Las actividades de ingeniería son las englobadas en otros ciclos como análisis, diseño, codificación, … de manera que el resultado en cada ciclo sea una versión más completa.

Las actividades de evaluación analizan los resultados de la fase de ingeniería, con la colaboración del cliente, de manera que el resultado obtenido se usa como entrada para la planificación del siguiente ciclo.

Mantenimiento o explotación del software

Evolución de las aplicaciones

Mantenimiento correctivo: Consiste en la detección y eliminación de errores que no han sido detectados durante el desarrollo de la aplicación y si a posteriori.

Mantenimiento adaptativo: Se aplica a aplicaciones cuyo medio de explotación evoluciona a lo largo del tiempo (SO) de manera que exige que esta se adapte a las nuevas características.

Mantenimiento perfectivo: Se promueve entre las aplicaciones sujetas a competencia, lo que implica que estas deben ir produciendo versiones mejoradas del producto que mantenga mantener su competitividad, realizándose también sobre aplicaciones en las que las necesidades del usuario evolucionan.

Gestión de los cambios

Cambio general: Si el cambio afecta a la mayoría de los componentes del producto, dicho cambio se puede plantear como un nuevo desarrollo, aplicando un nuevo ciclo de vida desde el principio.

Cambio específico: En caso de que el cambio afecte a una parte localizada del producto, lo que permite organizarlo como una modificación de algunos elementos. Un cambio en el código debe dar lugar a un cambio en los elementos de documentación afectados.

La realización de los cambios se puede controlar mediante dos clases de documentos, por un lado el **informe de problema**, el cual describe la dificultad del producto que requiere alguna modificación del producto y por otro lado el **informe de cambio**, que describe la solución dada a un problema y el cambio realizado en el software.

Reingeniería

En caso de mantener productos que no han sido desarrollados con ingeniería de software, con lo que solo se cuenta con el código fuente, se aplica ingeniería inversa, consistente en tomar código fuente, para construir a través de este la documentación, en especial la de diseño, con la estructura modular de la aplicación y las dependencias entre módulos y funciones. En general se intenta producir un sistema bien organizado y documentado a partir del sistema inicial deficiente.

Garantía de calidad de software

Factores de calidad

El esquema general de valoraciones de la calidad del software se denomina McCall78 y está basado en valoraciones a 3 niveles diferentes, un nivel superior denominado factores, los cuales incluyen la valoración propiamente, un nivel intermedio denominado criterios, son valoraciones respecto a unos criterios de calidad y un nivel inferior o métricas, consistente en mediciones puntuales de determinados atributos o características.

Los principales factores de calidad son:

- Corrección: Grado en que un producto de software cumple especificaciones o requisitos fijados.
- Fiabilidad:
- Eficiencia:
- Seguridad:
- Facilidad de uso:
- Mantenibilidad:
- Flexibilidad:
- Facilidad de prueba:
- Transportabilidad:
- Reusabilidad:
- Interoperatividad:

Plan de garantía de la calidad

Para alcanzar una buena calidad en el producto es necesaria la organización del proyecto de desarrollo, la cual debe materializarse en un producto formal denominado Plan de garantía de calidad de software, el cual debe contemplar:

- 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 guión del mismo.
- Revisiones y auditorías que se llevarán a cabo durante el desarrollo para garantizar la calidad del proyecto.
- Organización de las pruebas que se realizarán sobre el software a diferentes niveles.
- Organización de la etapa de mantenimiento, especificando como han de gestionarse los cambios.

Revisiones

Las revisiones consisten en la inspección del resultado de una actividad concreta para determinar si esta es aceptable o por el contrario contiene defectos que deben ser solucionados. Estas deben ser normalizadas siguiendo las siguientes recomendaciones:

- Las revisiones deben ser realizadas por un grupo, puesto que esto facilita el descubrimiento de fallos.
- El grupo debe ser reducido para evitar discusiones y facilitar la comunicación. Recomendable 3-5.
- La revisión no debe realizarse por los autores del producto, sino por personas que garanticen imparcialidad.
- Se debe revisar el producto, mas no el proceso de producción o al propio productor. Pues el producto perdura mientras el proceso pierda importancia en el uso del producto.
- Si la revisión debe decidir la aceptación o no del producto es recomendable establecer una lista formal de comprobaciones a realizar.
- Debe levantarse acta de la reunión de revisión, conteniendo los puntos importantes de discusión y las decisiones tomadas.

Gestión de configuración

La configuración de software hace referencia a la manera en que diversos elementos se combinan para construir un producto bien organizado, tanto desde el punto de vista de explotación, como de su desarrollo o mantenimiento. Siendo algunos de los componentes de la configuración los cuales intervienen en el desarrollo:

- Documentos del desarrollo: Especificaciones, diseño, …
- Código fuente de los módulos.
- Programas, datos y resultados de pruebas.
- Manuales de usuario.
- Documentos de mantenimiento: Informes de problemas y cambios.
- Prototipos intermedios.
- Normas particulares del proyecto.

El problema de la gestión de la configuración reside en que los elementos evolucionan a lo largo del desarrollo y la explotación del software, lo que implica la necesidad de realizar configuraciones particulares, compuestas de diferentes elementos, con lo que necesitamos una serie de técnicas que nos permitan mantener la configuración o configuraciones de software controladas, las cuales son:

- Control de versiones: Este registro contiene almacenados de forma organizada las sucesivas versiones de cada elemento de la configuración, de manera que al trabajar sobre una configuración concreta del producto se pueda acceder cómodamente a las versiones apropiadas de los elementos.
- Control de cambios: Garantiza que las diferentes configuraciones del software se componen de elementos compatibles entre sí, para lo que se usa el concepto de línea de base. Cada línea base se construye a partir de otra mediante la inclusión de ciertos cambios, que pueden ser la adición o supresión de elementos, o la sustitución de algunos por versiones nuevas de los mismos.

Normas y estándares

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

- IEEE:
- DoD:
- ESA:
- ISO:
- METRICA-2:

Tema 2. Especificación del software

1. Modelado de sistemas
2. Análisis de requisitos del software
3. Notaciones para la especificación
4. Documento de especificación de requisitos

Modelado de sistemas

Concepto de modelo

Un modelo conceptual consiste en todo que nos permite abstraernos del problema exacto permitiéndonos comprender con mayor precisión qué debe hacer el sistema y no como debe hacerlo. El modelo del sistema permite establecer las propiedades y restricciones del sistema, lo que implica una visión de alto nivel. Los principales objetivos a cumplir por los modelos son:

- **Facilitar la comprensión** del problema a resolver.
- Establecer un **marco para la comparación**, que permita simplificar y sistemizar la labor tanto del análisis inicial como de aquellos posteriores.
- **Fijar las bases para el desarrollo** del diseño final.
- Facilitar la verificación de la **cumplimentación de los objetivos iniciales**.

Técnicas de modelado

Puesto que la obtención de un modelo que cumpla con las funciones antes mencionadas no es fácil, se emplean una de las siguientes técnicas:

Descomposición. Modelo jerarquizado

La primera técnica a emplear sobre un problema complejo, es la división de este en otros más sencillos, de manera que según la división que se emplee podemos describir:

- Descomposición horizontal: Consiste en la descomposición del problema según la funcionalidad, siendo en el caso del programa para una empresa: Sistema de nóminas, sistema de contabilidad, …
- Descomposición vertical: Consiste en la descomposición del problema buscando detallar su estructura, en el caso anterior: Entrada de datos, cálculo de ingresos, pago de impuestos, …

Es posible descomponer cada uno de los problemas obtenidos en otros más simples, siendo importante establecer las interfaces entre las partes o subsistemas para lo que se aplica el método de refinamientos sucesivos al modelado del sistema.

Aproximaciones sucesivas

Puesto que es probable que el sistema que se quiere modelar provenga de una tarea que se realiza de manera manual o menos automatizada, se puede crear un modelo de partida basado en la forma de trabajo anterior de manera preliminar, siendo depurado continuamente mediante aproximaciones sucesivas hasta alcanzar un modelo final. Para lo que es importante contar con alguien que conozca bien el sistema anterior, siendo capaz de incorporar mejoras y discutir ventajas e inconvenientes de cada uno de los modelos medios.

Empleo de diversas notaciones

Puesto que la complejidad de un proyecto puede limitar la posibilidad de expresarlo mediante un único tipo de notación se hace preciso el empleo de varios modos de notación. Un método de notación sería el lenguaje natural que permite expresar el modelo mediante explicaciones, lo que puede dificultar la apreciación del conjunto, además de producirse imprecisiones, repeticiones e incluso incorrecciones. Lo ideal es emplear la notación o notaciones que mejor cubran los objetivos del modelo. Existen herramientas de modelado para la ayuda al análisis y diseño denominadas herramientas CASE.

Considerar distintos puntos de vista

Puesto que el modelo se encuentra influenciado por el punto de vista adoptado, se hace necesario elegir el punto de vista que permita obtener el modelo más adecuado para el comportamiento del sistema, teniendo que elegir entre: Punto de vista del usuario, del mantenedor del sistema, funcional, … Pudiendo en ocasiones ser necesario emplear mas de un punto de vista.

Realizar un análisis del dominio

El dominio es el campo de aplicación donde se clasifica el sistema a desarrollar, siendo importante para su calificación tener en cuenta los siguientes aspectos: Normativa que afecta al sistema, sistemas semejantes, estudios recientes en el campo de la aplicación y la bibliografía básica y especializada. Permitiendo la creación de un modelo más universal, que obtendrá las siguientes ventajas: Facilitar la comunicación entre analista y usuarios, creación de elementos realmente significativos del sistema, permitiendo que la solución no especifique en exceso y posibilitación de la reutilización posterior del software desarrollado. **pág. 41 EXT**.

Análisis de requisitos del software

Objetivos del análisis

El objetivo final del análisis en obtener las especificaciones que se desean del sistema a desarrollar, para lo que se necesita un modelo que permite recoger todas las necesidad y exigencias que propone el cliente, además de cumplir todas las restricciones que considere el analista. Las especificaciones obtenidas dependerán del modelo obtenido, para lo que se deben descartar aquellas exigencias por parte del cliente que sean imposibles de realizar. Para lo que se emplea un modelo que debe cumplir la siguientes propiedades:

- **Completo y sin omisiones**: es una propiedad difícil de conseguir debido a la dificultad para conocer todos los detalles del sistema que se pretende especificar.
- **Conciso y sin trivialidades**: Evitar la documentación excesiva extensa, evitando repeticiones.
- **Sin ambigüedades**: En el modelo resultante nada debe quedar ambiguo, pues podría producir errores y problemas de consecuencias difíciles de prever.
- **Sin detalles sobre el diseño o la implementación**: Es importante tener en cuenta que la finalidad del análisis es determinar QUE se va a hacer y no COMO, por lo que se debe plantear bien e problema sin intentar resolverlo.
- **Fácilmente comprensible por el cliente**: Para lo que se debe mantener el nivel de programación alto, utilizando un lenguaje asequible que facilite la colaboración con el cliente, siendo positivo el empleo de representaciones gráficas.
- **Separando requisitos funcionales y no funcionales**: Es importante separar los requisitos, pues los funcionales serán aquellos que interesan realmente al cliente, siendo los no funcionales de una carácter más técnico y carecen de interés para el cliente.
- **Dividiendo y jerarquizando el modelo**: Puesto que la forma más sencillo de simplificar un problema consiste en su correcta división en algunos más sencillos, esta tarea tiene una gran importancia.
- **Fijando los criterios de validación del sistema**: Es importante que el modelo del sistema quede expresamente indicados los criterios de validación que serán aquellos que cumplan el aspecto contractual, asegurando que nuestro software cumpla las expectativas propuestas.

Tareas del análisis

Para el desarrollo correcto del análisis de requisitos se deben seguir los siguientes pasos:

1) **Estudio del sistema en su contexto**: Es importante conocer el medio en el que se va a desenvolver el sistema, para lo que se debe tener en cuenta el análisis del dominio, que no solo incide sobre la terminología a emplear, sino que permite la visión del sistema mas globalizada, facilitando la reutilización de algunas partes.
2) **Identificación de las necesidades**: Puesto que inicialmente la actitud del cliente es solicitar todas aquellas funcionalidades que piensa podrían ser útiles, aparece la labor del analista el cual decide las necesidades que se pueden cubrir, con los medio de que se disponen. Lo que implica que para la realización de esta fase es necesaria una fluida conversación entre cliente y analista. Por lo que todas aquellas necesidades que finalmente deberán estar consensuadas por todos aquellos que participan en el análisis, para lo que el analista intentará convencer de ser la mejor opción con los medios disponibles
3) **Análisis de las alternativas y estudio de viabilidad**: Puesto que existen infinidad de soluciones para un mismo problema, es tarea del analista buscar aquella solución que cubra las necesidades reales, manteniendo una viabilidad técnica y económica.
4) **Establecimiento del modelo del sistema**: Puesto que el modelo se va perfilando según avanza el estudio de las necesidades y las soluciones adoptadas, se hace necesario el empleo de todos los medios disponibles como procesadores de texto, herramientas CASE, herramientas gráficas y todo aquello que permita facilitar la comunicación entre analista, cliente y diseñador.
5) **Elaboración del documento de especificación de requisitos**: El resultado final del análisis es el documento de especificación de requisitos, siendo una de las partes mas importantes el modelo del sistema, puesto que se debe ir perfilando su elaboración en el documento desarrollado. Este documento será el que emplee el diseñador como punto de partida de su trabajo, puesto que en este documento se especifican las condiciones de validación del sistema una vez concluido.

6) **Revisión continuada del análisis**: La labor del análisis no finaliza con la redacción del documento de especificaciones, siendo necesaria la revisión y modificación a causa de problemas acaecidos durante las fases de diseño e implementación o por un cambio de los criterios del cliente.

Notaciones para la especificación

Lenguaje natural

Se emplea principalmente cuando la complejidad de los sistemas es pequeña y consiste en la utilización del lenguaje que comúnmente utilizamos para comunicarnos, lo que lo hace inviable su utilización en sistemas con una complejidad media o alta, quedando relegado para aclarar algún aspecto que no se pueda hacer mediante el resto de notaciones.

En caso de querer emplear este tipo de notación es muy importante organizar y estructurar los requisitos recogidos en la especificación, siendo la mejora manera para lograrlo, concebir cada uno de los requisitos como una cláusula de un contrato entre analista y cliente, de manera que se agrupan los requisitos según su carácter: Funciones, calidad, seguridad, fiabilidad, …; estableciendo dentro de cada grupo las cláusulas de varios niveles y subniveles, por ejemplo en el grupo de funciones: Modos de funcionamiento, formatos de entrada, formatos de salida, … Lo que permitirá eliminar imprecisiones, ambigüedades y repeticiones.

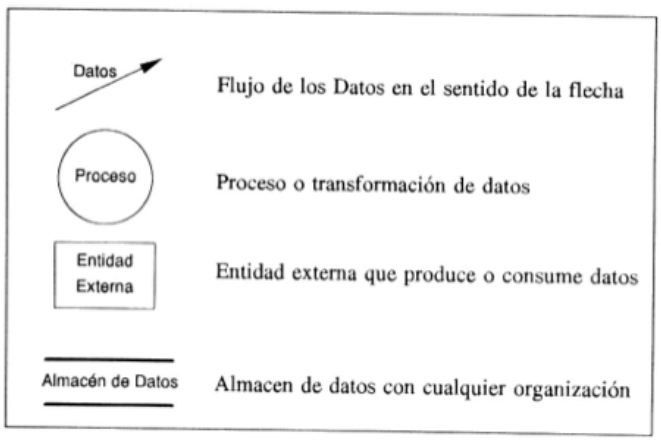
Por otro lado el lenguaje natural estructurado es una notación mas formal, la cual establece ciertas reglas para la construcción de las frases en las que se especifica una acción de secuencia, condición o iteración, tratando de emplear la misma construcción en todas las frases.

Diagramas de flujo de datos

Esta notación se asocia a la metodología de análisis de la década de los 70, de manera que estima que un sistema de software puede quedar definido por el conjunto de los datos de entrada, sus transformaciones, sus datos y sus soluciones de almacenamiento. Lo que permite su fácil aplicación a aquellos sistemas en los que se producen sucesivas fases de transformación sobre los datos de entrada.

El DFD se emplea de manera jerarquizada por niveles, denominando al primero y más general DFD de nivel 0 o DFD de contexto, el cual contiene los datos iniciales pero insuficientes, por lo que se hace necesario para explotarlo, de donde saldrá un DFD por cada proceso que existiera detallando mejor cada una de las funciones de este. Así cada vez que se crea un nuevo nivel de DFD se realiza numerando de forma correlativa los distintos procesos, de manera que en caso de explotar el proceso x, surgen los DFD x.1, x.2 y demás.

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.i.1 hasta DFD 1.i.z etc

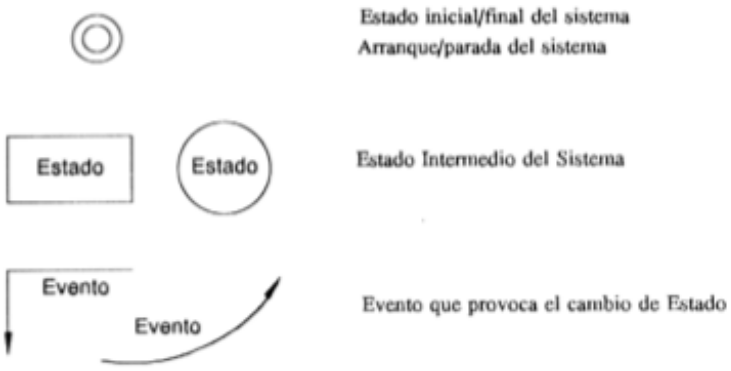


Hay que tener en cuenta que todos los flujos de entrada y salida deben coincidir antes y después de la explosión o refinamiento, siendo la principal ventaja de esta notación su simplicidad. Por lo que los DFD sirven para establecer un modelo conceptual del sistema que facilita la estructuración de su especificación, aunque

provoca que el modelo desarrollado con esta notación sea fundamentalmente estático, teniendo el problema de no poder reflejar la dinámica o secuencia en la ejecución de los procesos, siendo la única premisa carácter dinámico que se puede establecer es la utilización de un modelo abstracto de cómputo del tipo de flujo de datos.

Diagramas de transición de estados

Cada que se modifica una variable, se evalúa una condición o el término de una expresión, se produce un cambio de estado. Estos estados y las sucesivas transiciones entre ellos definen la dinámica de funcionamiento del sistema que se produce mientras se está ejecutando. Por lo que el diagrama de transición de estados es la notación específica para describir el comportamiento dinámico del sistema a partir de los estados elegidos como más importantes, por lo que complementa perfectamente con los DFD puesto que cubre la facción más dinámica del programa, que no se puede representar mediante DFD. la notación es la siguiente:



Descripciones funcionales o pseudocódigo

El pseudocódigo es una notación basada en un lenguaje de programación estructurado del que se excluyen todos los aspectos que son propios de cada lenguaje, como puedan ser tipos, variables o subprogramas, pudiéndose incluir descripciones en lenguaje natural a modo de comentarios, en caso de necesitar alguna aclaración. Cuando se utiliza se intenta detallar cual es la naturaleza del programa a resolver, sin intentar detallar como resolverlo, por lo que no se deben incluir ninguna forma concreta de organización de la información, ni tampoco propuestas concretas para la resolución de los problemas. Siendo lo básico:

- Selección: SI <condición> ENTONCES <Consecuencia>
- Selección por casos: CASO <variable que determina el caso> SI-ES<caso 1> HACER <consecuencia> SI-ES <caso 2> HACER <consecuencia> OTROS <Consecuencia>
- Iteración con pre-condición: MIENTRAS <condición> HACER <consecuencia> FIN-MIENTRAS
- Iteración con post-condición: HACER <consecuencia> HASTA <condición> FIN-HASTA
- Iteración de vueltas conocidas: PARA CADA <numero de veces> HACER <consecuencia>

Descripción de datos

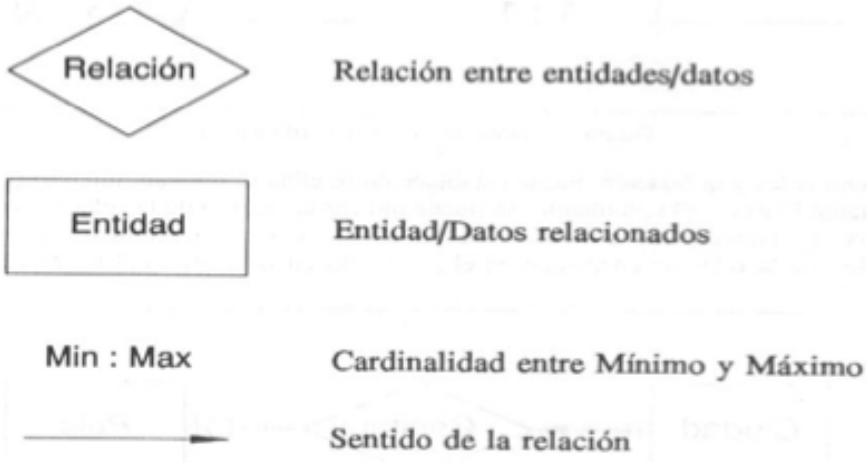
Trata de detallar la estructura interna de los datos que se van a emplear sin descender a detalles de diseño o codificación, utilizándose una metodología de análisis estructurado denominada diccionario de datos. Puesto que existen diversos formatos posibles de diccionario de datos, todos deben mantener en común el siguiente esquema:

- Nombre o nombres: Denominación con la se utilizará este dato durante el resto de la especificación.
- Utilidad: Indicará los procesos, descripciones funcionales, almacenes de datos, … Donde se utiliza el dato en concreto.
- Estructura:

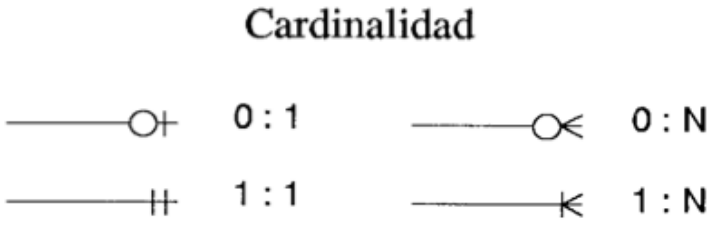
A+B	Secuencia de concatenación de los elementos A y B
[A | B]	Selección entre los distintos elementos A o bien B
$\{A\}^N$	Repetición N veces del elemento A. Se omite N si es indeterminado.
(A)	Opcionalmente se podrá incluir el elemento A
/ descripción /	Descripción en lenguaje natural como comentarios.
=	Separador entre el nombre de un elemento y su descripción

Diagramas de modelos de datos

Es una sistema que maneja una serie de datos relacionados entre sí, empleando lo que se conoce como modelo entidad-relación, que permite definir todos los datos que manejará la aplicación y la relación que existe entre ellos. Aun que el sujeto estará sujeto a modificaciones a lo largo de las fases de codificación y diseño, es muy importante pues es un punto de partida indispensable. La notación básica sería:



Un elemento fundamental es la cardinalidad de la relación que implica entre que valores de mínimo y máximo o valores de pertenencia se mueve la relación. La cardinalidad se dibuja siempre unida a la segunda entidad de la relación con un símbolo para el valor mínimo y otro para el máximo.



*** Referencia a bases de datos. Tema 6.2**

BLOQUE 2. DISEÑO DE SISTEMAS DE SOFTWARE

Índice

Tema 3. Fundamentos del Diseño de Software

1. Introducción

2. Conceptos de base

3. Notaciones para el diseño

Introducción

Definición de diseño

Es la descripción del sistema a desarrollar, tratando de definir y normalizar la estructura del sistema con el suficiente detalle como para permitir su realización física, siendo el punto de partida el documento de especificaciones de requisitos.

Es muy importante la experiencia previa, por lo que siempre que sea posible se intentará reutilizar la máxima cantidad posible de módulos o elementos ya desarrollados, inclusive, aun tratándose de un sistema completamente original se podrá utilizar el enfoque dado a algún proyecto anterior de similares características, lo que se denomina "know to know", siendo en sucesivas iteraciones cuando se perfile el enfoque más adecuado para el nuevo diseño. Hay que tener en cuenta que la fase de diseño es la más importante, ya que es la que marca el paso de QUE se quiere a COMO se hace.

Actividades fundamentales del diseño

1. **Diseño arquitectónico**: Detallar los aspectos estructurales y de organización del sistema y su posible división en subsistemas o módulos, lo que permitirá establecer las relaciones entre los subsistemas creados y definir las interfaces entre ellos.
2. **Diseño detallado**: Durante esta actividad se aborda la organización de los módulos, tratando de estudiar cual es la estructura más adecuada para cada uno de ellos. Como resultado de este, aparecen nuevos módulos que se deben incorporar al diseño global, se agrupan módulos que deban estarlo o se eliminan otros por estar vacíos de contenido, existiendo relación entre este diseño y el anterior.
3. **Diseño procedimental**: Aborda la organización de operaciones o servicios que ofrecerá cada uno de los módulos, siendo necesario detallar en pseudocódigo o PDL los aspectos más importantes de cada algoritmo en cada módulo.
4. **Diseño de datos**: Aborda la organización de la base de datos del problema, la cual se puede desarrollar en paralelo con el diseño detallado y procedimental, siendo el punto de partida el diccionario de datos y los diagramas de E-R de la especificación del sistema, siendo importante que el código obtenido sea reutilizable y fácilmente mantenible.
5. **Diseño de la interfaz de usuario**: Aborda la organización de la interfaz de usuario y cuya importancia es tal, que han aparecido herramientas especificas que facilitan mucho el diseño.

Conceptos de base

Abstracción

Cuando se diseña un nuevo software es importante identificar los elementos realmente significativos de los que consta el sistema y abstraer la utilidad específica de cada uno mas allá del sistema al que pertenecen y para el que se están diseñando, lo que permite cumplir con los principales objetivos: Conseguir elementos fácilmente reutilizables y fácilmente mantenibles.

En el diseño de los elementos software se pueden utilizar fundamentalmente 3 formas de abstracción:

- **Abstracciones funcionales**: Sirven para crear expresiones parametrizadas o acciones mediante el empleo de funciones o procedimientos, siendo necesario fijar los parámetros o argumentos que se le van a pasar, el resultado a devolver, lo que se pretende resolver y como.
- **Tipos abstractos**: Sirven para crear los nuevos tipos de datos que se necesitan para abordar el diseño del sistema. Junto al nuevo tipo de dato se deben diseñar los métodos u operaciones que se pueden realizar con él.
- **Maquinas abstractas**: Permite establecer un nivel de abstracción superior a los anteriores, definiéndose de una manera formal el comportamiento de una maquina.

Modularidad

El empleo de la modularidad promueve numerosas ventajas centradas en cuatro ámbitos:

- **División del trabajo**: Lo que permite encargar el desarrollo de cada módulo a personas diferentes, permitiendo que el trabajo sea simultáneo, aunque es importante las interfaces entre todos ellos que completamente definidas y correctamente diseñadas.
- **Claridad**: Debido a que es más sencillo comprender en su totalidad cada uno de los módulos que no el sistema al completo.
- **Reducción de costos**: Pues resulta más barato desarrollar, depurar, documentar, probar y mantener cada uno de los módulos que el sistema al completo.
- **Reutilización**: Puesto que cada módulo tiene una funcionalidad completa, este se puede emplear en otros proyectos cuya funcionalidad sea la misma.

Es muy importante recalcar que el empleo de la modularidad no debe estar ligado a la fase de codificación, ni mucho menos al lenguaje de programación empleado.

Refinamiento

El concepto de refinamiento resulta imprescindible, puesto que se parte de una idea no muy concreta que se va refinando en sucesivas aproximaciones hasta perfilar el más mínimo detalle, siendo el objetivo global de un sistema de software expresado en su especificación refinarlo en sucesivos pasos hasta que todo quede expresado en el lenguaje de programación del computador.

El uso directo del refinamiento sucesivo se puede aplicar cuando el programa es relativamente sencillo, ya que en caso contrario sería mejor aplicar las ideas de abstracción o modularidad para fragmentar la operación global en otras más sencillas y asequibles.

Estructuras de datos

La organización de la información es una tarea fundamental, puesto que el uso erróneo del tipo de contenedor puede complicar de sobremanera las acciones de búsqueda o modificación de los datos almacenados, según el tipo de datos a almacenar y los métodos a aplicar sobre estos se debe elegir entre los fundamentales soportes de datos: Registros, conjuntos, Formaciones, Listas, pilas, colas, árboles, grafos, tablas y ficheros. Logrando a través de la combinación de las estructuras básicas antes mencionadas aquella estructura final que nos permite resolver el problema.

Ocultación

Consiste en ocultar al usuario o a cualquier desarrollador que no tenga que ver directamente con el módulo concreto todo lo que pueda ser susceptible de cambio o irrelevante para su uso. Lo que permite obtener una serie de ventajas:

- **Depuración**: Resulta más sencillo localizar los errores, ya que se pueden desarrollar programas o estrategias de prueba que verifican y depuran en cada módulo según lo que hacen y no como lo hacen.
- **Mantenimiento**: Permite que el mantenimiento en un módulo no afecte a los demás.

Lo que permitirá utilizar un módulo sin conocer su estructura interna.

Genericidad

Consiste en la agrupación de aquellos elementos del sistema que utilizan estructuras semejantes o que necesitan de un tratamiento semejante, para lo que diseñamos un elemento genérico con las características comunes a todos los elementos agrupados, donde se puede diseñar cada uno como un caso particular del elemento genérico.

Herencia

Permite establecer una clasificación o jerarquía entre elementos del sistema partiendo de un elemento "padre" que posee una estructura y operaciones básicas, que heredan los "hijos" y al que cada uno le agrega

nuevas capacidades, las mejora o simplemente las adapta a su uso. Estos hijos pueden a la ver ser heredados por otros de manera que podemos crear un código que se fácilmente reutilizable.

Polimorfismo

El polimorfismo consiste es un concepto que aplicado a la informática implica lograr que un elemento adquiera varias formas simultáneamente. Lo que permite muchas posibilidades, entre ellas:

- El concepto de **genericidad** es una manera de lograr que un elemento genérico pueda adquirir distintas formas cuando se particulariza su utilización.
- El **polimorfismo de anulación**, donde las estructuras y operaciones del elemento "padre" se adaptan al elemento "hijo, pero algunas de estas se ven anuladas por nuevas versiones en el elemento "hijo".
- El **polimorfismo diferido**, que plantea la necesidad de la operación en el elemento "padre" pero su concreción se deja diferida para cada uno de los elementos "hijos" concrete su forma específica, lo que algunos lenguajes se denomina clase abstracta.
- El **polimorfismo de sobrecarga** que consiste en que los operadores que se utilizan para relacionar las diferentes entidades cumplan diferentes funciones o procedimientos según el tipo de entidades que se encuentren relacionando.

De lo que se entiende que el polimorfismo está ligado a las metodologías orientadas a objetos.

Concurrencia

Consiste en aprovechar la capacidad de proceso del computador ejecutando tareas de forma concurrente, ya que permite ejecutar algoritmos que no se podrían hacer de forma secuencial, lo que hace importante desarrollar un sistema de restricciones de tiempo, para lo que hay que tener en cuenta:

- **Tareas concurrentes**: Conocer las tareas que se pueden desarrollar simultáneamente para respetar las restricciones.
- **Sincronización de tareas**: Determinar claramente los puntos de sincronización entre las distintas tareas que operan de forma simultánea mediante monitores o semáforos.
- **Comunicación entre tareas**: Permite distinguir si la cooperación se basa en datos compartidos por las distintas tareas o el intercambio de mensajes, ya que en caso de emplear datos compartidos se hace necesario el empleo de métodos de sincronización y regiones críticas, lo que garantizará la exclusión mutua de las clases que intervienen en la comunicación.
- **Interbloqueo**: Es un problema producido cuando una serie de tareas se queda esperando por la finalización de otras, durante un tiempo indefinido.

Notaciones para el diseño

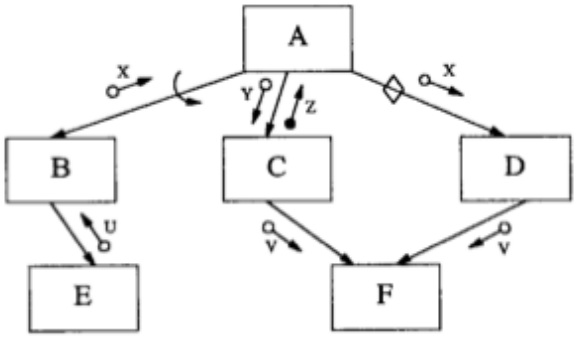
Notaciones estructurales

Sirven para cubrir un primer nivel del diseño arquitectónico, tratando con ellas de desglosar y estructurar el sistema en sus partes fundamentales. Una notación habitual para desglosar el sistema es el empleo de diagramas de bloques, donde se indican las conexiones entre los diferentes bloques, permitiendo en algunos casos representar una cierta clasificación o jerarquización. Otra notación es el empleo de cajas adosadas, donde la conexión entre dos bloques se pone de manifiesto cuando entre dos cajas existe una frontera común.

Diagramas de estructura

Estos diagramas fueron propuestos por Yourdon y Myers para describir la estructura de los sistemas de software como una jerarquía de subprogramas o módulos en general. El significado de los símbolos utilizados:

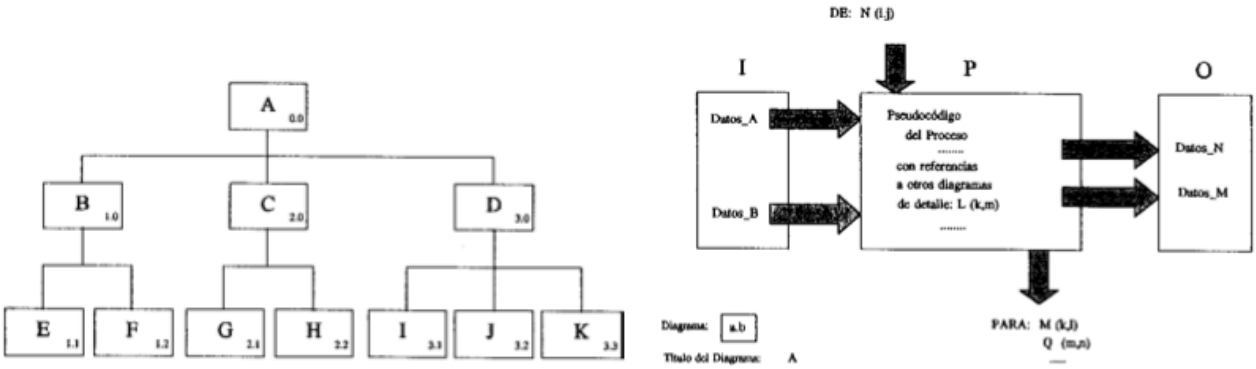
- **Rectángulo**: Representa un módulo o subprograma cuyo nombre se indica en su interior.
- **Línea**: Une dos rectángulos e indica que el módulo superior llama o utiliza al módulo inferior. A veces la línea acaba en una flecha junto al módulo inferior al que apunta.
- **Rombo**: Se sitúa sobre una línea e indica que esa llamada o utilización es opcional. Se puede omitir en caso de que en la posterior descripción del módulo superior se indica que el inferior se puede utilizar opcionalmente.
- **Arco**: Se sitúa sobre una línea e indica que esa llamada o utilización se efectúa de manera repetitiva.
- **Círculo con flecha**: Se sitúa en paralelo a una línea y representa en el envío de los datos cuyo nombre acompaña al símbolo, desde un módulo al otro. El sentido de envío lo marca la flecha, pero para indicar si los datos son de control se emplea un circulo relleno. Una información de control sirve para verificar SI/NO, o bien un estado: Correcto/Seguir repitiendo/ Error/ Desconectado…



El resultado es un diagrama en forma de árbol mostrando la jerarquización de los módulos, aunque es normal que varios módulos superiores empleen un mismo módulo inferior. EL diagrama de estructura no establece ninguna secuencia concreta de la utilización de los módulos, pues refleja una organización estática de los mismos.

Diagramas de HIPO

Los diagramas HIPO (Hierarchy-Input-Process-Output) es una notación orientada a facilitar y simplificar el diseño y desarrollo de sistemas software, destinados fundamentalmente a la gestión. la mayoría de estos sistemas se pueden diseñar como una estructura jerarquizada de subprogramas o módulos, además de que cada módulo tiene un formato que se puede adaptar a un patrón caracterizado por los datos de entrada, el tipo de proceso y los resultados de salida.



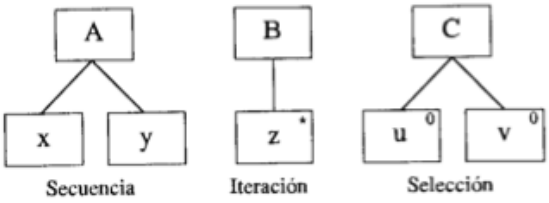
El diagrama de HIPO de contenidos se utiliza para establecer la jerarquía entre los módulos del sistema, lo que es semejante a una diagrama de estructuras simplificado, donde no se muestran los datos que se intercambian los módulos. Cada módulo tienen un nombre y una referencia al correspondiente diagrama HIPO.

El diagrama de HIPO de detalle consta de tres zonas bien diferenciadas: Entradas(I), Proceso(P), Salida (O), representando las zonas de entrada y salida a los datos que entran y salen del módulo, mientras que 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 listan a continuación de la partícula PARA, en la parte superior y a continuación de la partícula DE, donde se indica el diagrama de detalle superior.

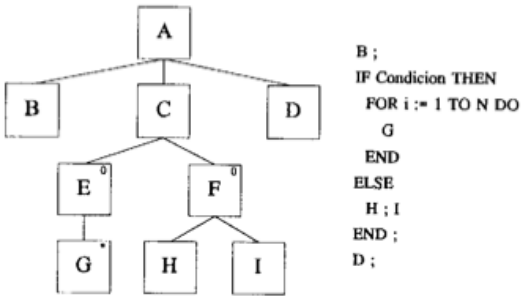
Diagramas de Jackson

Esta notación forma parte de la metodología de Jackson para diseñar sistemas de software a partir de las estructuras de sus datos de entrada y salida. El proceso de diseño es bastante sistemático y se lleva a cabo en 3 pasos:



1- Especificación de las estructuras de datos de entrada y salida.
2- Obtención de una estructura del programa capaz de transformar las estructuras de datos de entrada en las de salida. Lo que implica una conversión de las estructuras de datos en las correspondientes estructuras de programa que las manejan, siendo las equivalencias típicas:
   - TUPLA-SECUENCIA: Colección de elementos de tipos diferentes, combinados en un orden fijo.
   - UNIÓN-SELECCIÓN: Selección de un elemento entre varios posibles, de tipos diferentes.
   - FORMACIÓN-ITERACIÓN: Colección de elementos del mismo tipo.
3- Expansión de la estructura del programa para lograr el diseño detallado del sistema. Para realizar este paso normalmente se utiliza pseudocódigo.

Un ejemplo de su uso sería:



Si los elementos del diagrama representan datos, la estructura equivalente de diccionario de datos sería: A=B+C+D; C=(E|F); E={G}; F=H+I.

Notaciones estáticas

Estas notaciones sirven para describir características estáticas del sistema, sin tener en cuenta su posible evolución a lo largo del funcionamiento del sistema. Como resultado del diseño se tendrá una organización de la información con un nivel de detalle mucho mayor. Las notaciones que se pueden emplear para describir el resultado de este trabajo son las mismas que se empelan para realizar la especificación:

- **Diccionario de datos**: Detalla la estructura interna de los datos que maneja el sistema, por lo que se parte del diccionario de datos incluido en el documento SRD y mediante los refinamientos necesarios y se ampliará y completará hasta conseguir un nivel de detalle suficiente que permita pasar a la fase de codificación.
- **Diagramas de entidad-relación**: Permite definir el modelo de datos, las relaciones entre los datos y en general la organización de la información. Para la fase de diseño se tomará como punto de partida el diagrama propuesto e el documento SRD, que se completará y ampliará con las nuevas entidades y relaciones entre las mismas, que aparezca en la fase de diseño del sistema.

Notaciones dinámicas

Estas notaciones permiten describir el comportamiento del sistema durante su funcionamiento. Puesto que la especificación es precisamente una descripción de su funcionamiento externo, al diseñar la dinámica se detalla su comportamiento externo y se añade la descripción de un comportamiento interno capaz de garantizar que se cumplen los requisitos establecidos, así las notaciones que se emplean para el diseño serán las mismas utilizadas para la especificación:

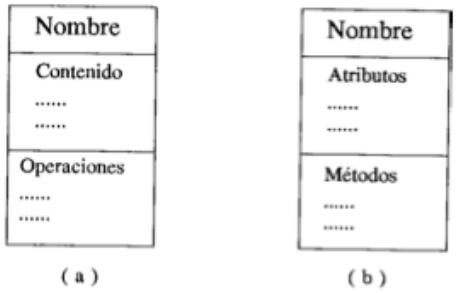
- **Diagrama de flujo de datos**: Las características de esta notación se hacen durante la especificación de software, pero a diferencias de estas son mucho mas exhaustivas, ya que los diagramas de flujo de datos deben describir como se hacen internamente las cosas y no solo el que.
- **Diagrama de transición de estados**: Al igual que la anterior está descrita durante la fase de especificación, pero durante la fase de diseño pueden aparecer nuevos diagramas de estado que reflejen las transiciones entre los estados internos. A pesar de ellos es preferible no modificar o ampliar los diagramas que aparecen el SRD encargados de reflejar el funcionamiento externo del sistema.
- **Lenguaje de descripción de programas (PDL)**: Esta notación se emplea tanto para realizar la especificación funcional del sistema, como para elaborar el diseño del mismo, marcando la diferencia entre ambas el nivel de detalle empleado. Tanto al especificar como al diseñar se utilizan las mismas estructuras básicas, pero para descender el nivel de detalle requiere en la fase de diseño del empleo de ciertas estructuras en un lenguaje de alto nivel como Ada-PDL.

Notaciones híbridas

Tratan de cubrir simultáneamente aspectos estructurales, estáticos y dinámicos, utilizando metodologías de diseño basadas en abstracciones y diseño orientado a objetos.

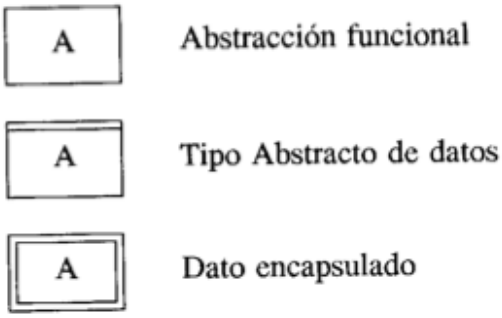
Diagramas de abstracciones

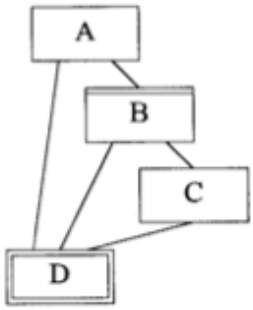
Permiten describir la estructura de un software como un compuesto de datos abstractos, donde en origen se contemplaban dos tipos de abstracciones: Las funciones y los tipos abstractos de datos, a lo que se le han añadido los datos encapsulados. Existe una cierta analogía entre la metodología de orientación a objetos y los diagramas de abstracciones que se muestra a continuación:



En la abstracción se distingue tres partes o elementos:

- **Nombre**: Identificador de la abstracción.
- **Contenido**: Elemento estático de la abstracción y en el que se define la organización de los datos que constituyen la abstracción.
- **Operaciones**: Elemento dinámico de la abstracción y en él se agrupan todas las operaciones definidas para manejar el contenido de la abstracción.
  Inicialmente la única forma de abstracción disponible en los lenguajes, y por tanto con la que únicamente se podía abordar el diseño, era la definición de subprogramas: Funciones (expresiones parametrizadas) o procedimientos (acciones parametrizadas). Un subprograma constituye una operación abstracta que denominaremos **abstracción funcional**, ya que esta no tiene la parte de contenido. En un diseño bien organizado se puede agrupar en una misma entidad la estructura del tipo de datos con las correspondientes operaciones necesarias para su manejo, denominándose a esta forma de abstracción **tipo abstracto de datos**, ya que dispone de una parte de contenido y una de operaciones. Cuando solo se necesita una variable de un determinado tipo abstracto su declaración se puede encapsular dentro de la misma abstracción, de manera que todas las operaciones se refieran a esa variable sin necesidad de indicarlo de manera explícita. Esta forma de encapsulado se denomina dato encapsulado, tiene contenido y operaciones, pero no permite declarar ma variables del mismo tipo.





Los módulos son abstracciones en sus distintas formas posibles y la relación entre estas es jerárquica e implica que la abstracción superior utiliza a la inferior. En el ejemplo la abstracción funcional A utiliza el dato encapsulado en D y el tipo abstracto B, mientras el dato encapsulado D es utilizado por B y la abstracción funcional C, a la vez que la abstracción funcional C es utilizada por B.

Diagramas de objetos

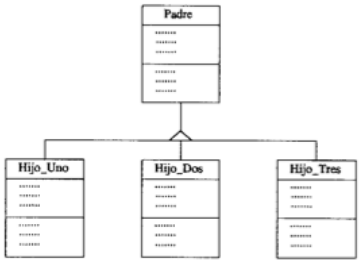
Mientras que las abstracciones se pueden considerar una propuesta de los expertos en programación, los objetos son una propuesta de los expertos en inteligencia artificial, ya que la estructura de on objetos es exactamente igual a la de una abstracción con dos diferencias fundamentales:

- No existe nada equivalente a los datos encapsulados ni a las abstracciones funcionales, si realizamos objetos en su forma estricta.
- Solo entre objetos se considera una herencia.

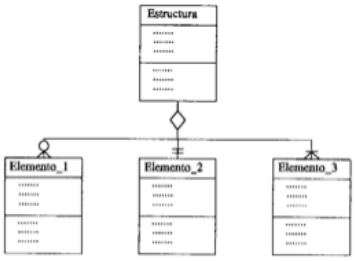
Abstracciones	Objetos
Tipo abstracto de datos	Clase de objeto
Abstracción funcional	No hay equivalencia
Dato encapsulado	No hay equivalencia
Dato encapsulado (Variable o constante)	Objeto (Ejemplar de la clase)
Contenido	Atributos
Operaciones	Métodos
Llamada a una operación	Mensaje al objeto

Debido a las propiedades particulares de los objetos se pueden establecer entre ellos dos tipos de relaciones:

1. **Clasificación, Especialización o Herencia**: Esta relación entre objetos permite diseñar un sistema aplicando el concepto de herencia. No contemplada entre las abstracciones, pudiendo denominar la herencia como especialización o clasificación. En la imagen los triángulos indican que los objetos inferiores heredan del superior atributos y operaciones.



2. **Composición**: La relación de composición permite describir un objeto mediante los elementos que lo forman. También es valido entre abstracciones. En la figura el rombo indica que el objeto superior se encuentra compuesto por los inferiores. En la relación de composición solo hay que indicar la cardinalidad en un sentido, ya que cada objeto "hijo" solo puede formar parte de un objeto "padre".



Tema 4. Técnicas Generales de Diseño de Software

1. Descomposición modular
2. Técnicas de diseño funcional descendiente
3. Técnicas de diseño basadas en abstracciones
4. Técnicas de diseño orientadas a objetos
5. Técnicas de diseño de datos
6. Diseño de bases de datos relacionales
7. Diseño de bases de datos de objetos

Descomposición modular

Definición

Todas las técnicas de diseño describen la necesidad de la descomposición modular del sistema como actividad fundamental del diseño, siendo necesario cubrir tres aspectos: Identificación de módulos, descripción de módulos y descripción de las relaciones entre ellos. La diferencia fundamental entre las diferentes técnicas de diseño hace referencia a los métodos que se emplean para llevar a cabo cada una de las actividades. Partiendo de que un módulo es un fragmento de software relativamente independiente de los demás, se pueden elaborar diferentes tipos de módulos entre ellos:

- **Código fuente**: Contienen el texto fuente escrito en el lenguaje de programación elegido, siendo el módulo considerado como tal con mayor frecuencia y en el que se hace referencia desde las diferentes técnicas de diseño.
- **Tabla de datos**: Se utiliza para tabular ciertos datos de inicialización, experimentales o simplemente difíciles de obtener.
- **Configuración**: Puesto que un sistema se puede concebir para trabajar en entornos diversos según las necesidades de cada cliente, interesa agrupar en un mismo módulo toda aquella información que permite configurar el entorno de trabajo.
- **Otros**: Un módulo sirve para agrupar ciertos elementos del sistema relacionados entre sí y que se pueden tratar de forma separada del resto.

El formato concreto de cada módulo depende de la técnica, metodología o herramienta, por lo que un mismo sistema se puede descomponer en módulos de muchas formas distintas, siendo el objetivo fundamental del diseño conseguir un sistema mantenible y solo en casos excepcionales se sacrificará este objetivo para lograr una mayor velocidad de proceso o menor tamaño de código.

Independencia funcional

Puesto que los módulos deben realizar tareas independientes es importa preservar la independencia funcional, pero puesto que la relación con los otros módulos del sistema es indispensable, es importante intentar que las relaciones con el resto de módulos sean mínimas, para lo que se deben tener en cuanta dos criterios: Acoplamiento y cohesión.

Acoplamiento

El grado de acoplamiento entre módulos es una medida de interrelación que existe entre ellos, afectando al tipo de conexión y la complejidad de la interfase. Dependiendo del nivel de acoplamiento entre los módulos se definen tres niveles, a los que se asocia cada tipo de acoplamiento, los cuales son:

- **Acoplamiento por contenido** (Fuerte): Se produce cuando desde un módulo se pueden cambiar los datos locales o incluso el código de otro módulo, por lo que no existe una separación real. Este tipo de acoplamiento solo se puede lograr con un lenguaje de bajo nivel y resulta prácticamente imposible de entender y depurar.
- **Acoplamiento común** (Fuerte): En este acoplamiento se emplea una zona común de datos a la que tienen acceso varios o todos los módulos, lo que implica que en la práctica cada módulo puedo estructurar y manejar la zona común con total libertad, lo que implica que el empleo de este acoplamiento exige que todos los módulos estén de acuerdo en la estructura de la zona común. Este tipo de acoplamiento se produce con el uso de COMMON o FORTRAN y responde al uso de una memoria que es escasa, siendo la depuración y mantenimiento muy difícil.
- **Acoplamiento externo** (Fuerte): Ocurre cuando la zona común es un dispositivo externo al que están ligados los módulos, por lo que la estructura de la zona común la impone el formato de los datos que maneja el dispositivo. Aunque este es inevitable es recomendable ligar a dispositivos externos la menor cantidad de módulos.
- **Acoplamiento de control** (Moderado): Es cuando se emplea una señal o dato de control generada en un módulo para controlar la línea de ejecución que se debe seguir en el módulo que lo recibe.
- **Acoplamiento por etiqueta** (Débil): Los módulos se intercambian no solo datos sino además la estructura completa de la que forman parte.
- **Acoplamiento de datos** (Débil): En este caso, el cual es el más ventajoso, los módulos solo se intercambian datos entre ellos, de manera que el acoplamiento es el mínimo posible.

Cohesión

La cohesión es el criterio que define en que medida los componentes de un módulo están relacionados o son afines al objetivo fijado para dicho módulo. Por otro lado se debe tener en cuenta no crear un número excesivo de módulos para no aumentar exageradamente la complejidad del sistema, lo que puede obligar a que determinados elementos sueltos se incorporen a un determinado módulo sin tener mucha afinidad con él, lo que provocará una disminución en la cohesión. Clasificados por el grado de cohesión existen:

- **Cohesión concidencial** (Baja): Es la peor posibles y se produce cuando cualquier relación entre los componentes del módulo es una coincidencia, por lo que no guardan relación entre ellos.
- **Cohesión lógica** (Baja): Se produce cuando se agrupan una serie de elementos que realizan funciones similares desde le punto de vista del usuario.
- **Cohesión temporal** (Baja): Se produce cuando se agrupan elementos que se ejecutan en un mismo momento, como ocurre con inicialización o finalización del sistema en que se deben "parar" o arrancar" determinados dispositivos.
- **Cohesión de comunicación** (Media): Se produce cuando todos los elementos del módulo operan con el mismo conjunto de datos de entrada o producen el mismo conjunto de datos de salida.
- **Cohesión secuencial** (Media): Se produce cuando los elementos del módulo trabajan de forma secuencial, de manera que la salida un elemento es la entrada del siguiente.
- **Cohesión funcional** (Baja): Se logra cuando cada elemento está encargado de la realización de una función concreta y específica. Con las técnicas de diseño funcional descendiente es el nivel máximo de cohesión que se puede lograr en un módulo, por lo que estas técnicas buscan este tipo de cohesión.
- **Cohesión abstraccional** (Alta): Esta se logra cuando se diseño un módulo como tipo abstracto de datos con la técnica basada en abstracciones o como una clase de objeto con la técnica orientada a objetos. En ambos casos se asocia un cierto contenido u organización de datos con las correspondientes operaciones que posibilitan su manejo.

Comprensibilidad

Puesto que los cambios continúan durante la fase de mantenimiento hasta que se sustituye por un sistema nuevo, y puesto que los cambios realizados son emprendidos por personas diferentes a las que participaron en el diseño y la implementación, es importante que la comprensión de los módulos de forma aislada no sea mas compleja que su realización. por lo que a parte de l acoplamiento y la cohesión es importante cuidar los siguientes factores:

- **Identificación**: Elección correcta de la nomenclatura del módulo y de sus componentes, que debe reflejar de manera sencilla el objetivo de la entidad.
- **Documentación**: Es importante la documentación de cada módulo, especialmente de aquellos aspectos de diseño o implementación que por su naturaleza no pueden quedar reflejados de otra manera.
- **Simplicidad**: Las soluciones sencillas son las mejores, pues un algoritmo complicado es difícil de entender, depurar y modificar en caso de ser necesario, pudiendo a veces sacrificarse la simplicidad en caso de contar con tiempo o memoria escasos.

Adaptabilidad

Dado que la descomposición modular está muy fijada por el objetivo concreto del diseño, lo que dificulta la adaptabilidad del diseño a otras necesidades, con lo que una vez cubiertas la independencia funcional y la comprensibilidad, se hace necesario cuidar otros factores que faciliten su adaptabilidad:

- **Previsión**: Puesto que es complicado prever la evolución de un determinado sistema, los módulos que se prevén puedan cambiarse se deben agrupar con un acoplamiento débil respecto a los demás, puesto que una modificación de la descomposición modular resulta complicada y costosa.
- **Accesibilidad**: Es importante que resulte sencillo el acceso a los documentos de especificación, diseño e implementación, lo que requiere una organización minuciosa, que en muchos casos solo se puede llevar a cabo mediante una herramienta CASE.
- **Consistencia**: Cuando se modifican los programas fuente se deben modificar todos los documentos implicados, lo que se puede hacer automáticamente gracias al empleo de herramientas para el control de versiones y configuración. puesto que en entornos orientados a objetos no existen estas herramientas se hace necesario emplear una disciplina férrea dentro de la biblioteca.

Técnicas de diseño funcional descendiente

Desarrollo por refinamiento progresivo

Esta técnica corresponde a la aplicación de la programación estructurada mediante el uso de estructuras de control claras y sencillas, con un único inicial y un único punto final de ejecución, en particular son la secuencia, la selección y la iteración. La construcción de programas basada en el concepto de refinamiento consiste en plantear el programa como una operación global e irla descomponiendo en función de otras operaciones más sencillas. Cada paso consistirá en refinar o detallar la operación considerada en ese momento, quedándonos en esta etapa en los primeros niveles de refinamiento.

Programación estructurada de Jackson

Esta técnica sigue las ideas de la programación estructurada en cuanto a las estructuras (secuencia, selección e iteración) y el método de refinamiento sucesivos, modificando las recomendaciones para ir construyendo la estructura del programa que debe ser similar a las estructuras de los datos de entrada y salida, por lo que es especialmente recomendada para las aplicaciones de procesamiento de datos. La técnica se basa en los siguientes pasos:

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

Diseño estructurado

Está técnica de diseño es el complemento del análisis estructurado, ya que ambas técnicas parten de los diagramas de flujo de datos (DFD) , llegando mediante esta técnica a los diagramas de estructura. La dificultad radica en que no basta con asignar módulos a procesos o grupos de procesos relacionados entre sí, sino que se debe establecer una jerarquía o estructura de control entre los diferentes módulos, que no está implícita en el modelo funcional descrito mediante los DFD. Para lo que se emplea un módulo de coordinación para construir la jerarquía, en el que se hacen dos análisis del flujo de datos global:

- **Análisis de flujo de transformación**: Consiste en identificar el flujo de información global desde los elementos de entrada del sistema, a los elementos de salida, separándose los procesos en tres regiones, flujo de entrada, de transformación y de salida. 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 de 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 centro de transacción del que salen las líneas de flujo y las regiones correspondientes a cada una de esas líneas o transacciones.

Técnicas de diseño basadas en abstracciones

Descomposición modular basada en abstracciones

Esta técnica considerada 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, mientras que aplicada al diseño, consiste en dedicar módulos separados a la realización de cada tipo abstracto de datos y cada función importante. Esta técnica de diseño puede emplearse tanto de forma ascendente como descendente.

- **Forma descendente**: se considera una ampliación de la técnica de refinamiento progresivo, en que realizar un refinamiento se plantea como alternativa, además de su descomposición, el que la operación a refinar se defina separadamente como abstracción funcional o como un tipo abstracto de datos.
- **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, más adecuados para el campo de aplicación que estamos diseñand. En muchos casos se pueden aplicar simultáneamente ambas maneras.

Método de Abbot

En este método se sugiere una forma metódica de conseguir una descomposición modular basada en abstracciones a partir de las descripciones o especificaciones del sistema hechas en lenguaje natural, por lo que esta técnica puede aplicarse también, y con mayor precisión, a las descripciones más formales que emplean notaciones precisas en vez de lenguaje natural. Para lo que se siguen los siguientes pasos:

- **Identificar** en el texto de descripción los tipos de datos como sustantivos genéricos, los atributos como sustantivos en general y las acciones como verbos o como nombres de acciones. Pudiendo algunos adjetivos sugerir valores de atributos.
- **Hacer dos listas** una con los nombres y otra con los verbos
- **Reorganizar** las dos listas extrayendo los posibles datos y asociándoles sus atributos y acciones, además de eliminar sinónimos y añadir los elementos implícitos en la descripción.
- Para **obtener el diseño** asignamos un módulo a cada abstracción de datos o grupo de abstracciones relacionadas entre sí, indicando para cada tipo abstracto de datos cuales son sus atributos y operaciones.

El módulo puede corresponder a un dato encapsulado si solo se maneja un dato de ese tipo en le programa. Este método se puede usar tanto en abstracciones como en diseño orientado a objetos.

Técnicas de diseño orientadas a objetos

Diseño orientado a objetos

El diseño orientado a objetos es similar al diseño basado en abstracciones, solo que añadiendo la herencia y el polimorfismo. Cada módulo contendrá la descripción de una clase objetos o de varias relacionadas entre sí. Además del diagrama modular. nos apoyamos en diagramas ampliados del modelo de datos, como los diagramas de estructura. La técnica general de diseño sigue los siguientes pasos:

1. **Estudiar y comprender el problema**: Aunque debe realizarse durante la fase de análisis de requisitos, puede ser necesario repetirlo en parte porque la especificación no sea suficientemente precisa o simplemente porque el diseño va a ser realizado por personas ajena a las que hicieron la especificación.
2. **Desarrollar una posible solución**: Aunque es posible que se haya hecho durante la fase de análisis, es probable que se tenga que hacer en la fase de diseño, para lo que convendrá considerar varias alternativas y elegir la que se considere más apropiada. La solución debe expresarse con suficientes detalles como para que en su descripción aparezcan mencionados los elementos que formarán parte del diseño.
3. **Formalizar** la estrategia en términos de clases y objetos y sus relaciones. Lo que se hace a través de las siguientes etapas:
    - Identificar las clases y objetos:
    - Identificar las operaciones:
    - Aplicar herencia:
    - Describir las operaciones:
    - Establecer la estructura modular:

Técnicas de diseño de datos

Definición

Puesto que la mayoría de las aplicaciones requieren almacenar datos de manera permanente, se suele hacer apoyando esa aplicación en una base de datos subyacente. La organización de la base de datos puede realizarse desde varios puntos de vista, siendo la forma clásica una clasificación tres niveles: externo, conceptual e interno. En cada nivel se establecen esquemas de organización de los datos desde un punto de vista concreto, correspondiendo los niveles a:

- **El nivel externo**: corresponde a la visión de usuario. La organización de los datos se realiza siguiendo esquemas significativos en el campo de la aplicación.
- **El nivel conceptual**: Establece una organización lógica de los datos, con independencia del sentido físico que tengan en el campo de aplicación. La organización se resume en un diagrama de modelo de datos.
- **El nivel físico**: Organiza los datos según esquemas admisibles en el sistema de gestión de bases de datos y/o lenguaje de programación elegido para el desarrollo. Si se utiliza una base de datos relacional los esquemas físicos serán esquemas de tablas.

El paso del nivel externo al conceptual se hace durante la etapa de diseño de requisitos y el paso de nivel conceptual a interno, se hace durante la fase de diseño.

Diseño de bases de datos relacionales

Diseño de bases de datos de objetos

BLOQUE III. CODIFICACIÓN Y PRUEBAS

Índice

Tema 5. Codificación y Pruebas

1. Desarrollo Histórico
2. Prestaciones de los lenguajes
3. Criterios de selección del lenguaje
4. Aspectos metodológicos
5. Técnicas de prueba de unidades
6. Estrategias de integración
7. Pruebas del sistema

Desarrollo Histórico

Primera generación

Son los lenguajes ensambladores, los cuales consisten en asociar a cada instrucción del computador un nemotécnico que recuerde cual es su función, por lo que su nivel de abstracción es muy bajo. La programación con este tipo de lenguajes resulta compleja porque los errores son difíciles de detectar y subsanar, a la vez que exige un conocimiento profundo de cada computador concreto. Debido a esto solo se emplea para la programación de pequeños fragmentos que después se incorporan en forma de macros o subrutinas, dentro de un programa en lenguaje de alto nivel.

Segunda generación

Se incorporan los primeros elementos realmente abstractos, como la inclusión de los tipos de datos, permitiéndose en parte ignorar la organización interna de la memoria para pasar a trabajar con variables simbólicas, además se incorporan las primeras estructuras de control para la definición de bucles o selectores genéricos. Algunos de los más representativos:

- **FORTRAN**: Tiene un origen científico, empleándose en sistemas de gestión de bases de datos y sistemas en tiempo real, por ejemplo. Su gran deficiencia es el control directo de memoria.
- **COBOL**: Se emplea para el desarrollo de los sistemas de procesamiento de la información.
- **ALGOL**: Es el primer lenguaje que da importancia a la tipificación de los datos, careciendo de una importante difusión.
- **BASIC**: Fue desarrollado para la enseñanza de la programación gracias a su sencillez y facilidad de aprendizaje.

Tercera generación

Aparecen asociados a las bases prácticas y teóricas de la programación estructurada, donde se prima por primera vez la productividad, lo que provoca la generación de lenguajes fuertemente tipados, que faciliten la estructuración de código y datos, con redundancia entre la declaración y el uso de cada tipo de dato, lo que facilita la verificación en compilación de las posibles inconsistencias de un programa. Algunos de los lenguajes mas importantes son:

- **PASCAL**: Fue desarrollado para la enseñanza de programación estructurada, pero su sencillez permitieron su uso en aplicaciones científico-técnicas, pero si tipificación de datos es bastante rígida y la compilación de forma separada es deficiente.
- **MODULA-2**: Se incorporan la estructura de módulo y queda separada la especificación del módulo de su realización, lo que facilita la aplicación inmediata de los conceptos fundamentales de diseño, además incorpora ciertos mecanismos básicos de concurrencia.
- **C**: Diseñado en un principio para la codificación del sistema operativo UNIX, se empleado para el desarrollo de todo tipo de aplicaciones, gracias a que posee características que lo hacen muy flexible y capaz de optimizar código tanto como si se empleara lenguaje ensamblador.
- **ADA**: Descendiente de PASCAL mas potente y complejo, ya que permite la definición de elementos genéricos y dispone de mecanismos para la programación concurrente de tareas y la sincronización y cooperación entre ellas.

Apareciendo de forma paralela otros lenguajes orientados a otros paradigmas como:

- **SMALLTALK**: Este lenguaje es el precursor de los lenguajes orientados a objetos.
- **C++**:Es un lenguaje que incorpora al lenguaje C los mecanismos básicos de la programación orientada a objetos, mediante la ocultación, las clases, la herencia y el polimorfismo. Lo que permite aprovechar la amplia difusión del lenguaje C para la introducción del nuevo paradigma.
- **EIFFEL**: Es el primer lenguaje que introduce la definición de clases genéricas, herencia múltiple y polimorfismo.
- **PROLOG**: Es el más importante representante de los lenguajes lógicos y se utiliza fundamentalmente en la construcción de lenguajes expertos.

Cuarta generación

Ofrecen al programador un mayor nivel de abstracción, siendo completamente independientes del computador que los ejecuta. Estos lenguajes no son de propósito general y en realidad se pueden considerar herramientas específicas para la resolución de determinados problemas en los campos más diversos, aunque no es aconsejable la realización de aplicaciones complejas debido a lo ineficiente del código que generan, aunque son óptimas para la realización de prototipos. Se pueden agrupar según su utilización en:

- **Bases de datos**: permiten acceder y manipular la información de la base de datos mediante un conjunto de órdenes de petición relativamente sencillas, lo que permite dotar a la base de datos de una gran versatilidad y permiten que sea el propio usuario quien diseñe sus propios listados, informes, etc.
- **Generadores de programas**: Permiten construir ciertos elementos abstractos fundamentales en cierto campo de la aplicación sin descender a los detalles concretos que se necesitan en los lenguajes de tercera generación, aunque el programa generado se puede modificar o adaptar cuando la generación automática no resulte completamente satisfactoria, lo que produce un ahorro considerable de tiempo.
- **Cálculo**: Con estos se puede desarrollar casi cualquier problema dentro de un campo, lo que incluye: Hojas de cálculo, herramientas de cálculo matemático, herramientas de simulación y diseño de control.
- **Otros**: Estos pueden englobar todo tipo de herramientas que permitan una programación de cierta complejidad, para lo que utilizan un lenguaje. Algunos ejemplos son las herramientas para la especificación y verificación de programas, los lenguajes de simulación o los de prototipos.

Prestaciones de los lenguajes

Estructuras de control

Programación estructurada

La programación estructurada implica el uso de las siguientes fórmulas:

- **Secuencia**: Indica la ejecución de las sentencias de manera continua.
- **Selección**: La selección se puede hacer de varias formas, siendo la más sencilla la cumplimentación de una afirmación, pudiendo indicar una serie de sentencias al caso afirmativo o negativo if-then o else-then, o una selección por casos denominada case-of.
- **Iteración**: Consiste en la repetición de una sentencia siendo su condicionante una afirmación repeat-until, un contador for-to-do, o un bucle de tipo indefinido loop-exit.

Manejo de excepciones

El manejo de excepciones consiste en la definición dentro del programa de la manera en la que se debe reaccionar en caso de ocurrir una serie de posibles errores o sucesos inesperados, que definimos como excepciones. Los errores se clasifican según su procedencia por:

- **Errores humanos**:
- **Fallos Hardware**:
- **Errores Software**:

Los errores de hardware son tarea del sistema operativo, de manera que este el que debe manejar este tipo de errores, evitando que puedan llegar a afectar al resto de los usuarios del computador o al propio sistema operativo, pero puesto que la medida correctora del sistema operativo en caso de fallo de software es siempre abortar la ejecución del programa, en caso de que el programa deba funcionar de manera continua y sin interrupciones no es admisible.

Es preferible escribir la parte principal del código atendiendo solo a las situaciones normales, lo que exige algún mecanismo apropiado para desviar automáticamente la ejecución hacia un fragmento de código apropiado en caso de ocurrir alguna situación anormal, lo que se denomina manejo de errores.

Concurrencia

Todos los lenguajes concurrentes permiten la declaración de distintas tareas y definir la forma en que se ejecuta la concurrencia, para lo que existen diferentes formas:

- **Corrutinas**: Las tareas son corrutinas que son semejantes a subprogramas, y entre ellas se pasan el control de ejecución. Se utilizan en Modula-2.
- **Fork-Join**: Una tarea puede arrancar la ejecución concurrente de otras mediante una orden fork, la cual se finaliza con un join invocado por la misma tarea que ejecutó el fork, y con el que ambas tareas se funden en una única. Se utiliza en UNIX y en el lenguaje PL/1.
- **Cobegin-Coend**: Todas las tareas que se deben ejecutar concurrentemente se declaran dentro de cobegin T1|T2|Tn coend. iniciándose todas al llegar al Cobegin y se finaliza la concurrencia cuando todas las tareas han acabado. Se utiliza en Algol68.
- **Procesos**: Cada tarea se declara como un proceso. Todos los procesos declarados se ejecutan concurrentemente desde el comienzo del programa, y no es posible iniciar uno nuevo. Se utiliza en Pascal concurrente. En Ada si se puede lanzar un proceso en cualquier momento.

Para lograr la sincronización y cooperación entre las tareas disponemos de:

- **Variables compartidas**: Ejecutándose en el mismo computador (multiprogramación), o en distintos Computadores pero utilizando una memoria compartida (multiproceso), son: Semáforos, Regiones criticas condicionales, Monitores.
- **Paso de mensajes**: Ejecutándose en Computadores que no tienen ninguna memoria común) procesos distribuidos, que están en una red lo que posibilita el paso de mensajes. Communicating Sequential Processes (CSP), Llamadas a procedimientos remotos, Rendezvous de Ada.

Estructuras de datos

- **Datos simples**: Entre los datos de tipo simple se encuentran los enteros, que son valores positivos o negativos sin decimales, los datos reales, que agregan a los anteriores los números con decimales. Por otro lado están los datos de tipo carácter y los de tipo ristra de caracteres o string que varían en su forma y manipulación según el lenguaje, además están los datos de tipo enumeración, que no existen en todos los lenguajes y que permiten definir un grupo de elementos asociados a una variable y finalmente los datos booleanos especialmente orientados a este tipo de álgebra y los cuales pueden tomar solo dos valores true o false.
- **Datos compuestos**: Este tipo de datos se definen como una combinación de otros datos simples o complejos pero ya definidos, prácticamente todos los lenguajes tienen métodos para definir y manejar este tipo de datos, utilizando para la definición de estos la estructura de registro (record). En caso de no disponer en el lenguaje de ese tipo de estructura específica será necesario el uso de un array o formación para agrupar varios datos en una estructura.
- **Constantes**: En los lenguajes modernos se pueden declarar constantes con nombre de forma simbólica para facilitar la identificación y el uso de estas.

Comprobación de tipos

- **Nivel 0** (Sin tipos): Lenguajes en los que no se pueden declarar nuevos tipos de datos y todos los datos que se utilizan deben pertenecer a sus tipos predefinidos. EL compilador no realiza ninguna comprobación de tipos. Es responsabilidad exclusiva del programador distinguir que representa cada dato.
- **Nivel 1** (Tipado automático): Es el compilador el encargado de decidir cual es el tipo más adecuado para cada dato que se utiliza, lo que implica la conversión de los operandos cuando estos son incompatibles entre sí o cuando los son con el operador utilizado en la expresión.
- **Nivel 2** (Tipado débil): Se realiza la conversión automática entre aquellos datos que guardan ciertas similitudes, de manera que en aquellas expresiones con operandos de distintos tipos, la conversión se hace hacia el tipo de mayor rango o precisión.
- **Nivel 3** (Tipado semirrígido): Todos los datos se deben declarar con sus correspondientes tipos, siendo imposible operar entre datos incompatibles. En los procedimientos y funciones se realiza una comprobación de la cantidad y tipo de datos para coincidir con la declaración, para lo que existen algunas vías de escape, que permiten evitar todo lo anterior.
- **Nivel 4** (Tipado fuerte): No existe ningún cambio posible, de manera que cualquier conversión debe hacerla el programador de forma explícita. Las comprobaciones se realizan en compilación, carga, etc.

Abstracciones y objetos

- **Abstracciones funcionales**: Puesto que estas han sido empleadas como una parte fundamental del diseño cada lenguaje dispone de una construcción para su definición variando como subprogramas, subrutinas, procedimientos, funciones, ... En todos es necesario definir el nombre y la interfaz de la abstracción, se tiene que codificar la operación que realiza y definir un mecanismo para su utilización.

  Normalmente en todos los lenguajes esta oculta la codificación de la operación, para quien hace uso de la misma. Pascal, Modula-2, Ada tienen visibilidad por bloques, esto es, pueden consultar o modificar datos de bloques externos. En Fortran hay total opacidad de dentro hacia fuera e viceversa.

- **Tipos abstractos de datos**: Se deben agrupar tanto el contenido o atributos y las operaciones definidas para el manejo del contenido, además de un mecanismo de ocultación que impida el acceso por una vía diferente a las operaciones permitidas.

- **Objetos**: Puesto que las diferencias entre objetos y abstracciones son mayores a la hora de la programación, los conceptos de polimorfismo y herencia aparecen ligados a los objetos lo que hace necesario que los lenguajes dispongan de mecanismos que les dispongan unas construcciones específicas. Modula-2 no dispone de mecanismos de herencia ni polimorfismo, así es complicado usarlo en diseño orientado a objetos. Ada solo dispone de polimorfismo de sobrecarga , pero no de anulación ni diferido, con lo que también es complicado. Los que si son aptos son: Smalltalk, Object Pascal, Eiffel, Objetive - C, C++.

Modularidad

El concepto de modularidad está ligado a la división del trabajo y al desarrollo en equipo de un proyecto de software, siempre la primera cualidad que se exige la compilación separada, que permite preparar y compilar de manera aislada el código de cada módulo. Podemos decir que la compilación es segura si en tiempo de compilación es posible comprobar que el uso de un elemento es consistente con su definición.

Esto cambia según el lenguaje de manera que Fortran y Pascal disponen de compilación no segura, Modula-2 y Ada tienen compilación segura, mediante DEFINITION MODULE e IMPLEMENTATION MODULE y package y package body. En C ANSI la compilación no será del todo segura pues hay posibilidad de error si no coincide la interfaz del modulo con la copia usada en el programa

Criterios de selección del lenguaje

- **Imposición del cliente**: En algunos casos es el cliente el que fija el lenguaje que se debe utilizar.
- **Tipo de aplicación**: Debido a las prestaciones de cualquier lenguaje de última generación se pueden realizar aplicaciones para los mas diversos campos, estando justificado el uso de lenguajes de ensamblador en aplicaciones de tiempo real muy crítico o para hardware especial.
- **Disponibilidad y entorno**: Se debe comprobar la compatibilidad de los lenguajes con los computadores elegidos, siendo importante un estudio de los compiladores en cuanto a la memoria y tiempo de ejecución del código. Finalmente resaltar que el desarrollo será más sencillo cuanto más potentes sean las herramientas empleadas, siendo considerables la facilidad de manejo y lo amigable de la herramienta.
- **Experiencia previa**: Siempre que sea posible es importante aprovechar la experiencia previa, pues la formación del personal es muy importante.
- **Reusabilidad**: Es importante por la posibilidad de utilizar software ya realizado como dejar disponible para otros proyectos partes del software desarrollado. Conviene disponer de herramientas dentro del lenguaje para organización de las librerías en las que se facilita la búsqueda y almacenamiento de los módulos reutilizables.
- **Transportabilidad:** Esta ligada a que exista un estándar del lenguaje que se pueda adoptar en todos los compiladores.
- **Uso de varios lenguajes**: No es aconsejable, pero hay veces en que es mas sencillo de codificar en diferentes lenguajes.

Aspectos metodológicos

Normas y estilo de codificación

Es fundamental fijar las normas que deben respetar todos los programadores, con el objetivo de lograr un resultado homogéneo, para lo que es importante fijar un estilo concreto en el que se debe concretar una serie de puntos:

- **Formato y contenido de las cabeceras de cada módulo**: Identificador del módulo, descripción del módulo, autor y fechas, revisiones y fechas, …
- **Formato y contenido de cada tipo de comentario**: Sección, orden, al margen, …
- **Utilización de encolumnado**: Tabulado (nº de espacios), máximo indentado, formato selección, formato iteración, …
- **Elección de los nombres**: Convenio para uso de mayúsculas y minúsculas, nombre de ficheros, identificadores de elementos del programa, …

Se deben incluir todas las restricciones o recomendaciones que puedan contribuir a la mejora de la claridad del código o a la simplificación del posterior mantenimiento.

Manejo de errores

Dentro de los conceptos básicos encontramos:

- **Defecto**: Puede permanecer oculto durante un tiempo indeterminado, en caso de que los elementos defectuosos no intervengan en la ejecución del programa.
- **Fallo**: Es el hecho de que un elemento del programa o funcione correctamente o produzca resultados erróneos, en caso de participar en la ejecución del programa producirá un fallo.
- **Error**: Se define como un estado inadmisible de un programa al que se llega como consecuencia de un fallo. Normalmente consiste en la salida o almacenamiento de resultados incorrectos.

Durante la codificación de un programa se pueden adoptar distintas actitudes respecto a los errores:

- **No considerar los errores**: Es lo más cómodo desde el punto de vista de la codificación, para lo que exige que todos los datos introducidos sean correctos y que el programador no tenga defectos, lo que no es realista.
- **Prevención de los errores**: Consiste en la detección de los fallos antes de que produzcan un error, para lo que cada programa y subprograma debe codificarse de manera que desconfíe de manera sistemática de los datos que se le introducen devolviendo siempre, el resultado correcto en caso de una datos válidos o una indicación precisa de fallo en caso de que estos no sean válidos. La ventaja principal radica en evitar la propagación de errores, facilitando el diagnóstico de posibles defectos.
- **Recuperación de los errores**: En caso de no poder detectar todos los posibles fallos, es inevitable que se produzcan errores, los cuales se pueden tratar con el objetivo de restaurar el programa en un estado correcto y evitar que el error se propague, para lo que se emplean dos actividades:
  - Detección del error: Concretar que tipo de situaciones son las que se consideran erróneas, para las que se realizan comprobaciones en situaciones estratégicas del programa.
  - Recuperación del error: Adoptar decisiones sobre como corregir el estado del programa para llevarlo a una situación consistente, lo que puede afectar a diferentes partes del programa, inclusive aquellas alejadas de aquella en la que se produjo el error.

Para la recuperación de errores existen dos esquemas generales:

- **Recuperación hacia delante**: Trata de identificar el tipo de error, de manera que pueda tomar las acciones adecuadas que permitan corregir el estado del programa y continuar de manera correcta la ejecución, lo que se puede llevar a cabo mediante el mecanismo de control de excepciones.
- **Recuperación hacia atrás**: Consiste en la corrección del estado del programa mediante la restauración a un estado correcto anterior a la aparición del error, con independencia de su tipo, para lo que se necesita guardar periódicamente el último estado del programa, por lo que se usa habitualmente en programas basados en transacciones.

Por lo que los programas que realizan prevención y recuperación de errores se denominan tolerantes a fallos.

Aspectos de eficiencia

La eficiencia se puede estudiar desde varios puntos de vista:

- **Eficiencia en memoria**: El bajo costo de la memoria hace que cuando se precisa cierto ahorro resulte suficiente con el uso de un compilador que disponga de posibilidades de comprensión de memoria. En caso de que el volumen de información a manejar sea demasiado grande para la memoria disponible, se debe realizar durante la fase de diseño estudios en los que se enumeren las distintas alternativas, optando por el algoritmo que optimice más la utilización de memoria.
- **Eficiencia en tiempo**: Tiene mayor importancia en la codificación de sistemas de tiempo real con plazos muy críticos, siendo a veces necesario disminuir la eficiencia de memoria para mejorar la eficiencia de tiempo. La primera vía para conseguir un ahorro importante es la realización durante la fase de diseño un estudio detallado y exhaustivo de las posibles alternativas al problema, permitiendo adoptar un algoritmo más rápido.

Las principales formas de obtener un ahorro de tiempo considerable son: Tabular los cálculos complejos, expansión en línea, desplegado de bucles, simplificación de expresiones aritméticas y lógicas, sacar fuera de los bucles aquello que no sea necesario repetir, usar estructuras de datos de acceso rápido, evitar operaciones con coma flotante y evitar conversiones innecesarias entre tipos de datos.

Transportabilidad del software

Los factores esenciales de la Transportabilidad son:

- **Uso de estándares**: Un producto desarrollado exclusivamente sobre los elementos estándar es teóricamente transportable sin ningún cambio, al menos entre plataformas que cuentan con el soporte para dicho estándar, lo que implica que la falta de estos dificulta la transportabilidad, por lo que se hace importante evitar aquellos elementos que no estén consolidados y que puedan estar sujetos a futuros cambios o revisiones.
- **Aislamiento de peculiaridades**: Es importante destinar un módulo a cada una de ellas, por lo que el transporte se ejecutará recodificando y adaptando estos módulos al nuevo computador.

Las principales peculiaridades se suelen encontrar ligadas a dos elementos:

- **Arquitectura del computador**: Determina la longitud de palabra, de lo que se deriva la representación interna de los valores enteros y reales.
- **Sistema operativo**: Puesto que es habitual el uso de funciones más particulares y complejas que las predefinidas en los lenguajes, se deben concretar y especificar cada una de ellas. Las nuevas operaciones se agrupan en módulos según su operación, para lo que se define una interfaz única y precisa en toda la aplicación. El resto de la aplicación utilizará estos módulos de forma independiente del sistema operativo, pudiendo utilizarse en la implementación de estos las operaciones disponibles en el lenguaje y el sistema operativo. Por lo que para transportar la aplicación a otro sistema operativo solo será necesario realizar una nueva implementación de estos módulos.

Técnicas de prueba de unidades

Objetivos de las pruebas de software

El principal objetivo de las pruebas es que el programa funcione incorrectamente, descubriéndose sus defectos, lo que exige analizar un juego de casos de prueba que permitan someter al programa al máximo número posible de situaciones diferentes. Para lo que se debe tener en cuenta:

- Una buena prueba es aquella que encuentra errores y no los encubre.
- Para conocer un error es necesario conocer cual sería el resultado correcto.
- Es bueno que no participen en la prueba el codificador o el diseñador.
- Siempre hay errores y en caso de no aparecer se hace necesario diseñar pruebas mejores.
- Al corregir un error se pueden producir otros nuevos.
- Es imposible demostrar la ausencia de defectos mediante pruebas.

Puesto que con las pruebas solo se consigue probar alguna de las posibilidades se hace necesario que con el menor esfuerzo posible se puedan detectar el máximo número de defectos, sobre todo aquellos que puedan provocar errores graves. Para poder garantizar unos resultados viables se hace necesario garantizar que el proceso de prueba se hace de la manera más automatizada posible, lo que implica un entorno de prueba que asegure unas condiciones predefinidas y estables para las sucesivas pasadas, después de los errores corregidos en cada tanda.

Las pruebas de unidades se realizan en un entorno de ejecución controlado, diferente del entorno final de explotación, de manera que se proporcione un informe con los resultados de las pruebas y un registro de todos los errores detectados con discrepancia respecto del valor esperado.

Pruebas de caja negra

La estrategia de caja negra ignora la estructura interna del programa y se basa únicamente en la comprobación de la especificación de entrada-salida, siendo la única estrategia que puede llevar a cabo cualquier persona ajena al desarrollo. Existen ciertos métodos basados fundamentalmente en la experiencia, los cuales se pueden utilizar de manera conjunta y complementaria, siendo algunos de los más extendidos:

- **Partición en clases de equivalencia**: Consiste en dividir el espacio de ejecución del programa en varios subespacios, agrupando en cada uno de ellos a aquellos números que sería:
  - Determinar las clases equivalentes apropiadas.
  - Establecer pruebas para clase de equivalencia, proponiendo casos o datos de prueba válidos o inválidos para cada una de las clases definidas anteriormente.

  Puesto que eligiendo bien las clases se reduce mucho el número de casos que se necesitan para descubrir un efecto, se pueden emplear las siguientes directrices:
  - Rango de valores: 0<valor<120, tomaríamos -5, 15 y 130.
  - Valor específico:
  - Conjunto de valores:

  Puesto que un caso de prueba válido para una clase de prueba puede ser inválido para otra, tal como un caso de prueba puede ser válido para varias clases, es importante el refinamiento de las pruebas, para lo que hay que seguir los siguientes pasos:
  1. Definir las clases equivalentes.
  2. Definir una prueba que cubra tantos casos válidos como sea posible de cualquier clase.
  3. Marcar las clases cubiertas y repetir el paso anterior hasta cubrir todos los casos válidos de todas las clases.
  4. Definir una prueba específica para cada caso válido.

- **Análisis de valores límite**: Hace un especial hincapié en las zonas del espacio de ejecución que están próximas al borde, los errores en los limites pueden ser graves al solicitar recursos que no se habían preparado. Las directrices para elaborar esta prueba son:
  - Entradas: Probar con los valores límite y justo fuera de los límites.
  - Salidas: Probar con los mismos valores límite y justo fuera de los límites.
  - Memoria: Probar con tamaños nulos, límite y superior al límite de todas las estructuras de información.
  - Recursos: Probar con ningún recurso, límite de recursos y superior al límite.
  - Otros: Pensar en otras situaciones límite y elaborar las pruebas.

- **Comparación de versiones**: Se hacen diferentes versiones del mismo programa hechas por diferentes programadores y se someten al mismo juego de pruebas, se comparan y evalúan los resultados, cuando produzcan los mismos resultados podemos elegir cualquier versión. Esto no es infalible pues un error en la especificación lo arrastrarían todas las versiones.

- **Empleo de la intuición**: Se debe dedicar cierto tiempo a preparar pruebas que planteen situaciones especiales y que puedan provocar algún error. Por lo que las personas ajenas al desarrollo del módulo suelen aportar un punto de vista mucho más distante y fresco de los que participan en él.

Pruebas de caja transparente

En este caso se conoce y se tiene en cuenta la estructura interna del módulo, de manera que se intenta que el programa transite todos los posible caminos de ejecución, poniendo en juego todos los elementos del código. Por esto es importante que las pruebas de caja negra y transparente sean complementarias y nunca excluyentes, puesto que una prueba únicamente de caja negra dejaría muchos caminos inexplorados. Los principales métodos que se emplean son:

- Cubrimiento lógico: Se determina un conjunto de caminos básicos que recorran todas las posibles líneas de flujo del diagrama que vendrá determinado por el número de predicados simples, siguiendo la siguiente fórmula: Nº máximo de caminos=Nº predicados+1. Para lo que tenemos diferentes niveles de encubrimiento:
  - Nivel 1: Se tratan de elaborar casos de pruebas para se ejecuten al menos una vez todos los caminos básicos. Cada uno de ellos por separado.
  - Nivel 2: Se tratan de elaborar casos de prueba para que se ejecuten todas las combinaciones de caminos básicos por parejas.
  - Nivel 3: Se tratan de elaborar casos de prueba para que se ejecuten un número significativo de las combinaciones posibles de caminos.
    Puesto que cubrir todas las combinaciones serían inabordable, como mínimo las pruebas deben garantizar el nivel 1. Finalmente hay que tener en cuenta que nunca se podrá detectar la falta de un fragmento de código.

- Pruebas de bucles: Para lo que se deben tener en cuenta las siguientes posibilidades.
  - Bucle con número no acotado de repeticiones: Para lo que se ejecuta el mismo bucle 0, 1 2, una cantidad moderada de veces y por último una vez elevada de veces.
  - Bucle con número máximo (M) de repeticiones: Para lo que se ejecuta el bucle 0, 1, 2, un número moderado de veces, se ejecuta M-1 veces, M veces exactas y finalmente M+1 veces.
  - Bucles anidados: Donde el número de pruebas crece geométricamente con el nivel de anidamiento, utilizando con la intención de reducir este incremento la siguiente técnica.
    1. Ejecutar todos los bucles externos en su número mínimo de veces para probar el bucle más interno con el algoritmo de bucle que corresponda.
    2. Para el siguiente nivel de anidamiento, ejecutar los bucles externos en su número mínimo de veces y los internos un número típico de veces, para probar el bucle del nivel con el algoritmo de bucle que corresponda.
    3. Repetir el paso 2 hasta completar todos los niveles.

- Bucles concatenados: Si son independientes se probaran por separado con los criterios anteriores.

Estimación de los errores no detectados

Resulta imposible demostrar que un módulo no tiene defectos, por que conviene obtener alguna estimación estadística de las erratas que pueden permanecer todavía sin ser detectadas. Se usa la siguiente estrategia:

1. Anotar el número de errores que se producen inicialmente con el juego de casos de prueba: E_I (Ej.: E_I=56)
2. Corregir el módulo hasta que no tenga ningún error con el mismo juego de casos de prueba.
3. Introducir aleatoriamente en el módulo un número razonable de errores en los puntos más diversos: E_A (Ej.: E_A=100.
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: E_D (Ej.: E_D=95)
5. Para el juego de casos de prueba considerado, suponiendo que se mantiene la misma proporción estadística, el porcentaje de errores sin detectar será el mismo para los errores iníciales que para los errores deliberados. E_E=(E_A-E_D)*(E_I/E_D)=(100-95)*(56/95) ≈3 errores

Estrategias de integración

Integración BIG BANG

Integración descendente

Integración ascendente

Pruebas del sistema

Objetivo de las pruebas

Pruebas Alfa y Beta