

TEMA 5 CODIFICACIÓN Y PRUEBAS

CODIFICACIÓN DEL DISEÑO

La fase de codificación es el núcleo principal de cualquiera de los modelos de desarrollo de software, en ella se elabora el producto fundamental de todo desarrollo software: el código fuente.

Un elemento esencial en la codificación es el lenguaje de programación.

En la fase de codificación pueden intervenir un gran número de programadores y durante un tiempo muy largo. Como parte de la metodología de programación y para garantizar la adecuada homogeneidad en la codificación se deben establecer las normas y estilo de codificación.

LENGUAJES DE PROGRAMACIÓN

El conocimiento de las prestaciones de los lenguajes permite aprovechar mejor sus posibilidades y también salvar sus posibles deficiencias. No existe un único lenguaje ideal para todo y a veces es necesario trabajar con varios lenguajes para las distintas partes de un mismo desarrollo.

DESARROLLO HISTÓRICO

PRIMERA GENERACIÓN

Lenguajes ensambladores. Un lenguaje ensamblador consiste fundamentalmente en asociar a cada instrucción del computador un nemotécnico que recuerde cuál es su función. Su nivel de abstracción es muy bajo.

La programación en ensamblador resulta compleja.

- Errores difíciles de detectar.
- Exige un conocimiento bastante profundo de la arquitectura del computador concreto.

Sólo se programarán en ensamblador pequeños fragmentos de programa que se podrán insertar, en forma de macros o subrutinas, dentro del programa escrito en un lenguaje de alto nivel.

SEGUNDA GENERACIÓN

Se incorporan los primeros elementos realmente abstractos. Se puede ignorar parte de la organización interna de la memoria del computador para pasar a trabajar con variables simbólicas de distintos tamaños y estructuras, según las necesidades de cada programa. Primeras estructuras de control para la definición de bucles genéricos o la selección entre varios caminos alternativos de ejecución.

- FORTRAN

Aplicaciones científicas o técnicas.

Deficiencia importante: manejo casi directo de la memoria.

- COBOL

Sistemas de procesamiento de información

- ALGOL

Primer lenguaje que da una gran importancia a la tipificación de datos.

- BASIC

Para la enseñanza de la programación. Sencillez y facilidad de aprendizaje.

TERCERA GENERACIÓN

Bases teóricas y prácticas de la programación estructurada.

La programación pierde definitivamente su consideración de “arte” destinado a demostrar las habilidades del programador para convertirse en una tarea productiva que se debe realizar de forma metódica y sin concesiones a la genialidad.

Lenguajes fuertemente tipados, que facilitan la estructuración de código y datos, con refundaciones entre la declaración y el uso de cada tipo de datos.

Se facilita la verificación en compilación de las posibles inconsistencias de un programa.

- PASCAL

Diseñado para la enseñanza de la programación estructurada.

Todo tipo de aplicaciones científico-técnicas o de sistemas. Permite la estructuración del código mediante procedimientos o funciones que se pueden definir de manera recursiva.

Tipificación de datos rígida y no se contempla en forma apropiada la compilación separada.

- MODULA-2

Se incorpora la estructura de módulo y queda separada la especificación del módulo de su realización concreta.

Se facilita enormemente la aplicación inmediata de los conceptos fundamentales de diseño: modularidad, abstracción y ocultación.

Incorpora ciertos mecanismos básicos de concurrencia.

- C

Diseñado para la codificación del sistema operativo UNIX. Ha sido utilizado ampliamente para desarrollar software de sistemas y aplicaciones científico-técnicas de todo tipo.

Posee ciertas características que le hacen muy flexible y capaz de optimizar el código tanto como si se utilizara un lenguaje ensamblador.

- ADA

Departamento de Defensa de los EEUU. Descendiente de Pascal mucho más potente y complejo.

Permite la definición de elementos genéricos y dispone de mecanismos para la programación concurrente de tareas y la sincronización y cooperación entre ellas.

Lenguajes asociados a otros paradigmas de programación o modelos abstractos de cómputo.

- **SMALLTALK**

Precursor de los lenguajes orientados a objetos.

- **C++**

Incorpora al lenguaje C los mecanismos básicos de la programación orientada a objetos.

- **EIFFEL**

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

- **LISP**

Precursor de los lenguajes funcionales. Aplicaciones de inteligencia artificial y sistemas expertos.

Pensado especialmente para la manipulación de símbolos, demostración de teoremas y resolución de problemas teóricos.

- **PROLOG**

Es el representante más importante de los lenguajes lógicos y se utiliza fundamentalmente para la construcción de sistemas expertos.

CUARTA GENERACIÓN

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

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.

No es aconsejable utilizar estas herramientas para desarrollar aplicaciones complejas debido a lo ineficiente que resulta el código que generan. Se pueden usar para la realización de prototipos.

Agrupación según su aplicación concreta:

- **BASES DE DATOS**

Permiten acceder y manipular la información de la base de datos mediante un conjunto de órdenes de petición (QUERY) relativamente sencillas.

Dotan de una gran versatilidad a la base de datos y permiten que pueda ser el propio usuario quien diseñe sus propios listados, informes, etc.

- **GENERADORES DE PROGRAMAS**

Se pueden construir elementos abstractos fundamentales en cierto campo de aplicación sin descender a los detalles concretos que se necesitan en los lenguajes de la tercera generación.

El programa generado se puede modificar o adaptar cuando la generación automática no resulta completamente satisfactoria.

Se produce un ahorro considerable de tiempo de desarrollo.

- **CÁLCULO**

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

Hojas de cálculo, herramientas de cálculo matemático, herramientas de simulación y diseño para control, etc.

- **OTROS**

Pueden englobar todo tipo de herramientas que permitan una programación de cierta complejidad y para ello utilicen un lenguaje.

Herramientas para la especificación y verificación formal de programas, lenguajes de simulación, lenguajes de prototipos, etc.

PRESTACIONES DE LOS LENGUAJES

ESTRUCTURAS DE CONTROL

Conjunto de sentencias o instrucciones de los lenguajes que se encuadran dentro de la parte ejecutiva de un programa.

PROGRAMACIÓN ESTRCUCTURADA

- **SECUENCIA**

Normalmente escribiendo una tras otras las sentencias.

- **SELECCIÓN**

- **SELECCIÓN GENERAL**

Mediante un if – then – elseif- then -else

- **SELECCIÓN POR CASOS**

Mediante una sentencia case - of

- **ITERACIÓN**

- **REPETICIÓN**

repeat - until

- **BUCLE CON CONTADOR**

for – to -do

- **BUCLE INDEFINIDO**

loop y exit

Para facilitar el desarrollo por refinamientos sucesivos, todos los lenguajes permiten definir subprogramas mediante el empleo de procedimientos o funciones. Estos programas se pueden definir en forma recursiva.

MANEJO DE EXCEPCIONES

Durante la ejecución de un programa se pueden producir errores o sucesos inusuales que denominaremos genéricamente excepciones.

Los errores pueden tener distintos orígenes: errores humanos, fallos hardware, errores software.

Pueden tratarse como excepciones situaciones no erróneas pero poco frecuentes: datos de entrada vacíos, valores fuera de rango, etc.

- **ERRORES HUMANOS**

El operador puede introducir datos erróneos, aunque el programa verifique la validez de cada dato por separado cuando se introduce.

- **FALLOS HARDWARE**

El mal funcionamiento de un periférico puede dar como resultado un dato erróneo que se introduce al programa.

Errores debidos a la capacidad limitada de los recursos.

- **ERRORES SOFTWARE**

El programa puede tener algún defecto no detectado durante la fase de pruebas.

Los sistemas operativos son los encargados de detectar los fallos hardware y evitar que los errores humanos o de software puedan llegar a afectar al resto de los usuarios del computador o al mismo sistema operativo.

La medida correctora del sistema operativo es siempre abortar el programa para evitar males mayores. Este hecho no es admisible en un programa que debe funcionar siempre y en cualquier circunstancia (control de central nuclear, frenos ABS de un coche, cajero automático, etc.)

Es preferible siempre que sea posible, escribir la parte principal del código atendiendo sólo a las situaciones normales. Esto exige algún mecanismo apropiado para desviar automáticamente la ejecución hacia un fragmento de código apropiado en caso de que ocurra alguna situación anormal, esto se contempla en lenguajes que incorporan el manejo de excepciones.

CONCURRENCIA

Aspectos fundamentales a tener en cuenta para su diseño y programación:

- Tareas que se deben ejecutar concurrentemente.
- Sincronización de tareas.
- Comunicación entre tareas
- Interbloqueos (deadlock)

Todos los lenguajes concurrentes permiten declarar distintas tareas y definir la forma en que se ejecutarán concurrentemente. Existen distintas formas de abordar este aspecto:

- CORRUTINAS

No existen tareas sino corrutinas, que tienen una estructura semejante a los subprogramas, pero entre ellas se pueden transferir el control de ejecución en cualquier momento y no sólo de una forma jerarquizada según el estricto orden de llamada/retorno.

La concurrencia se limita a un avance de la ejecución de todas las corrutinas pero secuenciando ese avance por un acuerdo entre ellas.

- FORK-JOIN

Una tarea puede arrancar la ejecución concurrente de otras mediante una orden fork. La concurrencia finaliza con un join invocado por la misma tarea que ejecutó el fork y con el que ambas tareas se funden de nuevo en una única.

UNIX y lenguaje PL/1.

- COBEGIN-COEND

Todas las tareas que se deben ejecutar concurrentemente se declaran dentro de una construcción cobegin $T_1 | T_2 | \dots | T_n$ coend

La finalización de la concurrencia exige que todas las tareas hayan acabado. Es posible anidar varios cobegin-coend.

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 ninguno nuevo. Esto se usa en Pascal concurrente.

En ADA se usa esta misma propuesta, pero es posible lanzar de forma dinámica un proceso en cualquier momento.

Para lograr la sincronización y la cooperación entre tareas, se dispone de estructuras con las que resolver ambos aspectos. Estas estructuras se pueden clasificar en dos grandes grupos:

- VARIABLES COMPARTIDAS

- Semáforos.
- Regiones críticas condicionales.
- Monitores.

Necesita que todas las tareas se ejecuten en un mismo computador (multiprogramación) o en distintos computadores pero utilizando una memoria compartida (multiproceso) para que todas las tareas tengan acceso a las VARIABLES COMPARTIDAS que emplean para comunicarse.

- PASO DE MENSAJES
 - Communicating Sequential Processes CSP
 - Llamada a procedimientos remotos.
 - Rendezvous (ADA)

Se puede utilizar para la sincronización y cooperación entre tareas que se ejecutan en computadores que no tienen ninguna memoria común (procesos distribuidos) y que están conectados por una red de comunicaciones que les permite comunicarse mediante el PASO DE MENSAJES.

ESTRUCTURAS DE DATOS

Distintas formas que emplean los lenguajes para estructurar los datos que manejan.

DATOS SIMPLES

- Datos de tipo entero

Todos los lenguajes. Se debe tener en cuenta su rango. Algunos tienen dos o más rangos de enteros: normal, corto, largo, etc.
- Datos de tipo real.

Todos los lenguajes. Tener en cuenta la precisión. A veces se dispone de reales con precisión simple y con doble precisión. En algunos lenguajes más dedicados al cálculo científico se pueden manejar directamente datos complejos.
- Datos de tipo carácter

Todos los lenguajes. Habitualmente se emplea el código ASCII, es conveniente comprobarlo.
- Datos de tipo ristra de caracteres (string)

La mayoría de los lenguajes. No existe una forma única de organización y operación con estos datos.
- Datos simples por enumeración.

Cuando un lenguaje no tiene esta posibilidad se pueden emplear directamente los enteros y es responsabilidad del programador hacer una asociación y manipulación correctas.

Dato enumerado del que disponen la mayoría de los lenguajes es el tipo lógico o booleano.
- Datos de tipo subrango

Permiten acotar el rango de un tipo de dato ordinal para crear un nuevo tipo de dato. Resulta más sencilla la depuración y el mantenimiento de los programas.

DATOS COMPUESTOS

Los datos compuestos siempre se definen como combinaciones de otros tipos de datos simples y compuestos ya definidos.

Prácticamente todos los lenguajes tienen la posibilidad de definir y manejar formaciones (vectores o matrices).

La definición de datos compuestos de elementos heterogéneos (esquema tupla) se realiza en la mayoría de los lenguajes mediante la estructura registro (record).

En lenguajes en los que no existen los registros hay que recurrir a un array o formación para agrupar varios datos en una sola estructura.

CONSTANTES

En los lenguajes modernos se pueden declarar constantes simbólicas, con nombre.

COMPROBACIÓN DE TIPOS

Se pueden distinguir cinco niveles.

- 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 qué representa cada dato.

- NIVEL 1 (Tipado automático)

Es el compilador el encargado de decidir cuál es el tipo más adecuado para cada dato que utiliza el programador. El compilador también se encarga de convertir al tipo adecuado los operandos de una expresión cuando estos son incompatibles entre sí o cuando lo son con el operador utilizado en la expresión.

- NIVEL 2 (Tipado débil)

Se realiza una conversión automática de tipos pero solamente entre datos que poseen ciertas similitudes.

En expresiones con operandos de distintos tipos las conversiones siempre se hacen hacia el tipo de mayor rango o precisión.

- NIVEL 3 (Tipado semirrígido)

Todos los datos que se quieran usar deben ser declarados previamente con sus correspondientes tipos.

Nunca es posible realizar operaciones entre datos de tipos incompatibles. Los procedimientos y funciones, se comprueba el número de los argumentos y el tipo de cada uno de ellos para que coincida la declaración con su utilización. Existen algunas vías de escape que permiten evitar todo lo anterior.

- NIVEL 4 (Tipado fuerte)

No existe ninguna escapatoria posible y el programador está obligado a hacer explícita cualquier conversión de tipo. Las comprobaciones se realizan en compilación, carga y ejecución.

ABSTRACCIONES Y OBJETOS

ABSTRACCIONES FUNCIONALES

Según el lenguaje, la denominación puede variar: subprogramas, subrutinas, procedimientos, funciones, etc.

Normalmente permanece oculta la codificación de la operación para quien hace uso de la misma. Dependiendo de las reglas de visibilidad de cada lenguaje.

TIPOS ABSTRACTOS DE DATOS

Para la codificación de un tipo abstracto de datos se deben agrupar en una entidad única el contenido o atributos de los datos de la abstracción y las operaciones definidas para el manejo del contenido.

Debe existir un mecanismo de ocultación que impida el acceso al contenido por una vía distinta a las que ofrecen las operaciones definidas.

OBJETOS

Solamente con los denominados lenguajes orientados a objetos resulta factible la codificación de diseños orientados a objetos con herencia simple o múltiple y polimorfismo.

MODULARIDAD

El concepto de modularidad está ligado a la división del trabajo y al desarrollo en equipo de un proyecto software.

La primera cualidad que se exige es la compilación separada, que permite preparar y compilar separadamente el código de cada módulo.

Cuando es posible comprobar en tiempo de compilación que el uso de un elemento es consistente con su definición, diremos que la compilación separada es segura.

CRITERIOS DE SELECCIÓN DEL LENGUAJE

Criterios de selección que deben ser analizados:

- **IMPOSICIÓN DEL CLIENTE**

En algunos casos es el cliente el que fija el lenguaje que se debe utilizar.

- **TIPO DE APLICACIÓN**

Con las prestaciones de cualquier lenguaje de última generación se pueden realizar aplicaciones para los más diversos campos.

Sólo en aplicaciones de tiempo real muy crítico o para hardware muy especial, sin compiladores de lenguajes de alto nivel, estaría justificado el empleo de lenguajes ensambladores.

- **DISPONIBILIDAD Y ENTORNO**

No existen compiladores de todos los lenguajes para todos los computadores. Se debe comprobar qué compiladores existen para el computador elegido.

Es interesante realizar un estudio comparativo de los compiladores disponibles en cuanto a la memoria y tiempo de ejecución del código.

Un factor importante a tener en cuenta en la selección es el entorno que acompaña a cada compilador. El desarrollo será más sencillo cuanto más potentes sean las herramientas disponibles.

Se deben considerar las facilidades de manejo de estas herramientas y lo amigable que resulta su utilización.

- **EXPERIENCIA PREVIA**

Siempre que sea posible es más rentable aprovechar la experiencia previa. La formación del personal es una inversión muy importante.

- **REUSABILIDAD**

Es importante por la posibilidad de utilizar software ya realizado como por el hecho de dejar disponibles para otros proyectos partes del software desarrollado.

Conviene disponer de herramientas dentro del entorno del lenguaje para la organización de dichas librerías en las que se facilite la búsqueda y el almacenamiento de los módulos reutilizables.

- **TRANSPORTABILIDAD**

Es interesante que el lenguaje utilizado sea transportable. La transportabilidad está ligada a que exista un estándar del lenguaje que se pueda adoptar en todos los compiladores.

- **USO DE VARIOS LENGUAJES**

No es aconsejable mezclar varios lenguajes en un mismo proyecto, pero hay ocasiones en las que las distintas partes de un mismo proyecto resultan mucho más sencillas de codificar si se utilizan diferentes lenguajes.

Para tomar esta decisión se debe hacer un estudio de la compatibilidad entre los compiladores.

ASPECTOS METODOLÓGICOS

NORMAS Y ESTILO DE CODIFICACIÓN

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

Es importante fijar un estilo concreto y que todo el equipo lo adopte y respete.

Para fijar un estilo de codificación se deberán concretar al menos los siguientes puntos:

- Formato y contenido de las cabeceras de cada módulo:
 - Identificación del módulo.
 - Descripción del módulo
 - Autor y fecha
 - Revisiones y fechas
 - ...

- Formato y contenido de cada tipo de comentario.
 - Sección
 - Orden
 - Al margen
 -
- Utilización de encolumnado.
 - Tabulador, nº de espacios.
 - Máximo indentado.
 - Formato selección.
 - Formato iteración
 - ...
- Elección de nombres.
 - Convenio para uso de mayúsculas y minúsculas.
 - Nombres de ficheros.
 - Identificadores de elementos del programa.

Se deben incluir todas aquellas restricciones o recomendaciones que puedan contribuir a mejorar la claridad del código y a simplificar su mantenimiento posterior.

MANEJO DE ERRORES

Conceptos básicos:

- Defecto

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

Es el hecho de que un elemento del programa no funcione correctamente, produciendo un resultado (parcial) erróneo.
- Error

Estado inadmisibles de un programa al que se llega como consecuencia de un fallo. Consiste en la salida o almacenamiento de resultados incorrectos.

Al codificar un programa se pueden adoptar distintas actitudes respecto al tratamiento de los errores, más precisamente de los fallos.

- **NO CONSIDERAR LOS ERRORES**

La más cómoda desde el punto de vista de la codificación. Se exigirá como requisito que todos los datos que se introduzcan deberán ser correctos y que el programa no deberá tener ningún defecto.

No es realista.

- **PREVENCIÓN DE ERRORES**

Detectar los fallos antes de que provoquen un error.

Programación a la defensiva, que consiste en que cada programa o subprograma esté codificado de manera que desconfíe sistemáticamente de los datos o argumentos que se le introducen y devuelva siempre:

a.- El resultado correcto, si los datos son válidos.

ó

b.- una indicación precisa del fallo, si los datos no son válidos.

La ventaja principal es evitar la propagación de errores, facilitando así el diagnóstico de los defectos.

- **RECUPERACIÓN DE ERRORES**

Cuando no se pueden detectar todos los posibles fallos, es inevitable que en el programa se produzcan errores.

Se puede hacer un tratamiento del error con el objetivo de restaurar el programa en un estado correcto y evitar que el error se propague. El tratamiento exige dos actividades diferentes y complementarias:

- **DETECCIÓN DEL ERROR**

Concretar qué situaciones se considerarán erróneas y realizar las comprobaciones adecuadas en ciertos puntos estratégicos del programa.

- **RECUPERACIÓN DEL ERROR**

Adoptar decisiones sobre cómo corregir el estado del programa para llevarlo a una situación consistente. Estas decisiones pueden afectar a otras partes del programa diferentes y alejadas de aquella en la que se produce la detección del error.

Dos esquemas generales para la recuperación de errores:

- **Recuperación hacia adelante**

Trata de identificar la naturaleza o el tipo de error, para posteriormente tomar las acciones adecuadas que corrijan el estado del programa y le permitan continuar correctamente su ejecución.

Se puede programar mediante el mecanismo de excepciones.

- Recuperación hacia atrás

Corregir el estado del programa restaurándolo a un estado correcto anterior a la aparición del error, con independencia de la naturaleza del error.

Se necesita guardar periódicamente el último estado correcto del programa. Se usa habitualmente en sistemas basados en transacciones.

Todos los programas que realizan una previsión o recuperación de errores se denominan tolerantes a fallos.

ASPECTOS DE EFICIENCIA

La eficiencia se puede analizar 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 que se puede obtener de forma automática empleando un compilador que disponga de posibilidades de compresión de memoria.

Cuando el volumen de información a manejar sea excesivamente grande para la memoria disponible, durante la fase de diseño, se deben estudiar las distintas alternativas y optar por el algoritmo que optimice más la utilización de la memoria.

- Eficiencia en tiempo

Tiene mayor importancia en la codificación de sistemas de tiempo real con plazos muy críticos. En ocasiones se logra una mayor eficiencia de tiempo disminuyendo la eficiencia en memoria.

La primera vía para conseguir un ahorro de tiempo importante es realizar en la fase de diseño detallado un estudio exhaustivo de las posibles alternativas del problema y adoptar el algoritmo más rápido.

Formas simples de obtener un ahorro de tiempo significativo:

- Tabular los cálculos complejos
- Expansión en línea: si se usan macros en lugar de subrutinas se ahorra el tiempo necesario para la transferencia de control y el paso de argumentos.
- Desplegado de bucles: en la evaluación de la condición de un bucle se emplea un tiempo que se puede ahorrar repitiendo el código de forma sucesiva.
- Simplificar expresiones aritméticas y lógicas.
- Sacar fuera de los bucles todo lo que sea necesario repetir.
- Usar estructuras de datos de acceso rápido.
- Evitar operaciones en coma flotante y usar coma fija.
- Evitar conversiones innecesarias de tipos de datos.

TRANSPORTABILIDAD DEL SOFTWARE

Factores esenciales de la transportabilidad:

- Utilización de estándares

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

La falta de estándares dificulta la transportabilidad.

Evitar aquellos elementos no consolidados por completo y que puedan estar sujetos a futuros cambios o revisiones.

- Aislar las peculiaridades

Destinar un módulo específico para cada una de ellas. El transporte se resolverá recodificando y adaptando solamente estos módulos específicos al nuevo computador.

Las peculiaridades fundamentales de los computadores suelen estar vinculadas a los elementos siguientes:

- ARQUITECTURA DEL COMPUTADOR

Determina la longitud de palabra y de esto se derivan la representación interna de los valores enteros y reales.

- SISTEMA OPERATIVO

Es habitual necesitar operaciones más complejas y particularizadas que las predefinidas en los lenguajes. Se deben concretar y especificar claramente cada una de ellas.

Las nuevas operaciones se agruparán en módulos, de entrada/salida, manejo de ficheros, librerías matemáticas, etc.. propios de cada aplicación. Para cada módulo y operación se definirá una interfaz única y precisa en toda la aplicación. El resto de la aplicación utilizará estos módulos de forma independiente del sistema operativo. En la implementación de estos módulos se podrán utilizar las operaciones disponibles en el lenguaje y el sistema operativo.

Para transportar la aplicación a otro sistema operativo sólo será necesario realizar una nueva implementación de estos módulos a partir del nuevo sistema operativo y sus operaciones más o menos sofisticadas.

TÉCNICAS DE PRUEBA DE UNIDADES

Es necesario someter al programa a diversas pruebas destinadas a detectar los errores o verificar su funcionamiento correcto.

Para software crítico el costo de las pruebas puede ser la partida más importante del costo de todo el desarrollo.

Para evitar el caos de una prueba global única, se deben hacer pruebas a cada unidad o módulo según se avanza en la codificación del proyecto. Se facilitará enormemente las posteriores pruebas de integración entre módulos y las pruebas del sistema total.

OBJETIVOS DE LAS PRUEBAS DE SOFTWARE

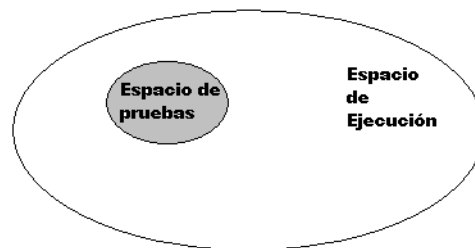
El principal objetivo de las pruebas es conseguir que el programa funcione incorrectamente y que se descubran sus defectos. Esto exige elaborar minuciosamente un juego de casos de prueba destinados a someter al programa a un máximo número posible de situaciones diferentes.

Para elaborar casos de prueba se debe tener en cuenta:

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

Probar completamente cada módulo es inabordable y además no resulta rentable ni práctico.

Dominio de las pruebas



Con las pruebas sólo se explora una parte de todas las posibilidades del programa. Se debe alcanzar un compromiso para que con el menor esfuerzo posible se puedan detectar el máximo número de defectos y sobre todo aquellos que puedan provocar las más graves consecuencias.

Para garantizar unos resultados fiables es esencial que todo el proceso de prueba se realice de la manera más automática posible, lo cual exige crear un entorno de prueba que asegure unas condiciones predefinidas y estables para las sucesivas pasadas, después de corregir los errores detectados en cada pasada anterior.

Las pruebas de unidades se realizan en un entorno de ejecución controlado, diferente del entorno de ejecución del programa final en explotación. El entorno debe proporcionar un informe con los resultados de las pruebas y un registro de todos los errores detectados con discrepancia respecto al valor esperado.

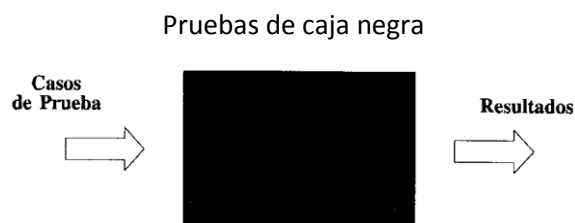
Las técnicas de prueba de unidades responden a dos estrategias fundamentales:

PRUEBAS DE CAJA NEGRA

La estrategia de caja negra ignora por completo la estructura interna del programa y se basa exclusivamente en la comprobación de la especificación entrada salida del software. Trata de verificar todos los requisitos impuestos al programa.

Es la única estrategia que puede adoptar el cliente o cualquier persona ajena al desarrollo del programa.

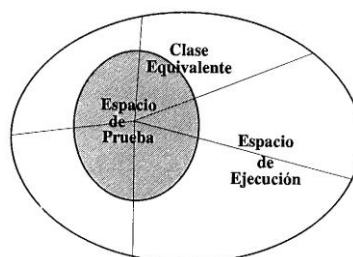
Elaborar unos buenos casos de prueba que permitan conocer el correcto funcionamiento del programa.



Métodos en la elaboración de casos de prueba:

- **PARTICIÓN EN CLASES DE EQUIVALENCIA**

Dividir el espacio de ejecución del programa en varios subespacios. Cada subespacio o clase equivalente agrupa a todos aquellos datos de entrada al módulo que resultan equivalentes desde el punto de vista de la prueba de caja negra.



Pasos a seguir:

1. Determinar las clases equivalentes apropiadas.
2. Establecer pruebas para cada clase de equivalencia: proponer casos o datos de pruebas válidos e inválidos para cada una de las clases definidas en el paso anterior.

Si las clases se eligen adecuadamente, se reduce bastante el número de casos que se necesitan para descubrir un defecto.

Según cómo se caractericen las clases de equivalencia se pueden emplear las siguientes directrices.

A. Rango de valores:

Ejemplo: $0 \leq \text{edad} < 120$ años

Casos válidos:	1 dentro del rango	33 años
Casos inválidos:	1 mayor y 1 menor	-5 y 132 años

B. Valor específico:

Ejemplo: Clave = OCULTA

Casos válidos:	1 igual	OCULTA
Casos inválidos:	1 distinto	Otra578

C. Conjunto de valores:

Ejemplo: Operaciones = compra, venta, cambio

Casos válidos: 1 por elemento compra, venta, cambio

Casos inválidos: 1 fuera del conjunto regalo

Un caso de prueba válido para una clase puede ser también un caso de prueba inválido para otra y asimismo puede ocurrir que un caso de prueba bien elegido sea válido para varias clases

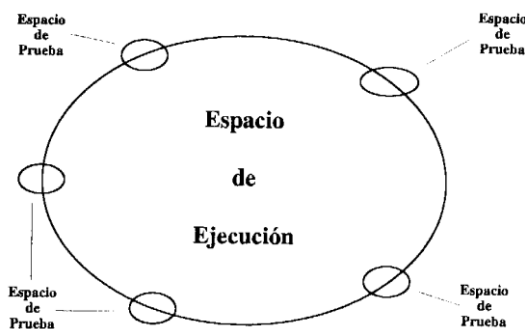
Refinamiento de los pasos indicados:

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 inválido

○ ANÁLISIS DE VALORES LÍMITE

Es frecuente que los errores tengan cierta tendencia a aparecer al operar en las fronteras o valores límite de los datos normales.

El método de análisis límite (boundary analysis) hace hincapié en las zonas del espacio de ejecución que están próximas al borde.



Este método es complementario del anterior.

Se deben de poner casos de prueba válidos e inválidos.

Los errores en los límites pueden tener consecuencias más graves de lo normal debido a que pueden provocar la necesidad de unos recursos extra que no estaban previstos.

Directrices para elaborar casos de prueba:

1. Entradas: Probar con los mismos valores límite y justo fuera de límites. Ejemplo, si la precisión en los cálculos = 5 cifras, probar 5 y 6
2. Salidas: Probar con los mismos valores límite y justo fuera de límites. Ejemplo: Para listados con N° líneas/página = 70, probar 70 y 71
3. Memoria: Probar con tamaños nulos, límite y superior al límite de todas las estructuras de información. Ejemplo, para pilas, colas, tablas, etc., probar los casos vacío, lleno y sobrelleno (un elemento más de lleno)
4. Recursos: Probar con ningún recurso, límite de recursos y superior al límite. Ejemplo, si máximo N° de terminales = 30, probar 0, 30 y 31
5. Otros: Pensar en otras situaciones límite y establecer las pruebas

○ COMPARACIÓN DE VERSIONES

Cuando una unidad o módulo es especialmente crítico se puede hacer un desarrollo multi-versión (N-version) encargando la codificación de diferentes versiones a distintos programadores. Todos utilizarán las mismas especificaciones de partida y deberían obtener módulos completamente intercambiables.

Se elaborará un caso de pruebas mediante los métodos habituales. Tener en cuenta que no siempre se conocen de forma completamente exacta los resultados esperados. En los sistemas en tiempo real, resulta casi imposible conocer a priori cuáles serán esos resultados.

Todas las versiones se someten al mismo juego de casos de pruebas de una forma completamente automática. Los resultados se comparan entre ellos y con los esperados.

Cualquier discrepancia entre las distintas versiones se debe analizar hasta discernir si una versión es errónea y sus causas. Cuando todas las versiones produzcan los mismos resultados y éstos coincidan con los deseados se puede suponer que todas son equivalentes y correctas, y se puede utilizar cualquiera de ellas.

La redundancia que se introduce en la codificación de varias versiones aumentan las garantías de que el módulo funciona correctamente y que cumple las especificaciones. No es un criterio infalible.

○ EMPLEO DE LA INTUICIÓN

Se debe dedicar cierto tiempo a preparar pruebas que planteen situaciones especiales y que puedan provocar algún error.

Las personas ajenas al desarrollo del módulo suelen aportar un punto de vista mucho más distante y fresco que las que participan en él.

PRUEBAS DE CAJA TRANSPARENTE

En la estrategia de caja transparente se conoce y se tiene en cuenta la estructura interna del módulo.

Se trata de conseguir que el programa transite por todos los posibles caminos de ejecución y ponga en juego todos los elementos del código.

Los casos de prueba deben conseguir que:

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

En la mayoría de los programas sólo se llegan a ejecutar un número bastante reducido de todos los caminos posibles. No resulta fácil prever a priori qué caminos son los que se ejecutarán cuáles no.

Las pruebas de caja negra y de caja transparente deben ser complementarias y nunca excluyentes. Es conveniente aplicar el método de análisis de valores límites para elaborar pruebas de caja transparente teniendo en cuenta la estructura del programa.

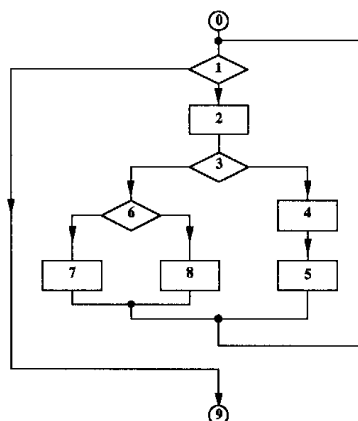
Las pruebas de caja transparente deben ser propuestas por alguien que haya participado o conozca la codificación en detalle.

Métodos utilizados:

- CUBRIMIENTO LÓGICO

Camino básico. Cualquier recorrido que siguiendo las flechas de las líneas de flujo nos permita ir desde el punto inicial al punto final del diagrama.

Diagrama de flujo con 3 predicados lógicos simples



Cada rombo del diagrama debe representar un predicado lógico simple, no puede ser una expresión lógica con algún operador OR, AND, etc.

Primero, se determina un conjunto de caminos básicos que recorran todas las líneas de flujo del diagrama alguna vez.

$$\text{Nº. máx. caminos} = \text{Nº. predicados} + 1$$

En el caso de la figura de ejemplo:

$$\text{Nº. máx. caminos} = 3 + 1 = 4$$

Y serían suficientes los siguientes cuatro caminos:

Camino 1: 0-1-9

Camino 2: 0-1-2-3-4-5-1-9

Camino 3: 0-1-2-3-6-8-1-9

Camino 4: 0-1-2-3-6-7-1-9

El método del cubrimiento lógico consiste en elaborar casos de prueba para que el programa recorra un determinado conjunto de caminos siguiendo ciertas pautas.

Se pueden establecer distintos niveles de cubrimiento:

- NIVEL I

Elaborar casos de prueba para que el programa recorra una vez todos los caminos básicos, cada uno de ellos por separado.

- NIVEL II

Elaborar casos de prueba para que se ejecuten todas las combinaciones de caminos básicos por parejas.

- NIVEL III

Elaborar casos de prueba para que se ejecuten un número significativo de las combinaciones posibles de caminos.

Cubrir todas las combinaciones posibles resulta inabordable.

Como mínimo las pruebas de cada módulo deben garantizar el nivel I. Con el cubrimiento lógico se pueden quedar sin detectar el 50% de los errores y es necesario utilizar otros métodos.

Nunca se podrá detectar la falta de un fragmento de código.

- PRUEBA DE BUCLES

Distinguiremos las siguientes situaciones:

- Bucle con número no acotado de repeticiones

Se elaborarán pruebas para:

- Ejecutar el bucle 0 veces.
- Ejecutar el bucle 1 vez.
- Ejecutar el bucle 2 veces.
- Ejecutar el bucle un número moderado de veces.
- Ejecutar el bucle un número elevado de veces.

- Bucle con número máximo (M) de repeticiones

- Ejecutar el bucle 0 veces.
- Ejecutar el bucle 1 vez.
- Ejecutar el bucle 2 veces.
- Ejecutar el bucle un número intermedio de veces.
- Ejecutar el bucle M-1 veces.
- Ejecutar el bucle M veces.
- Ejecutar el bucle M+1 veces.

- Bucles anidados

El número de pruebas crece forma geométrica con el nivel de anidamiento; para reducir este número se utilizará 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 bucles 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 probarán cada uno por separado con alguno de los criterios anteriores.

Si están relacionados se empleará un enfoque similar al indicado para los bucles anidados.

- EMPLEO DE LA INTUICIÓN

Merece la pena dedicar un cierto tiempo a elaborar pruebas que sólo por intuición podemos estimar que plantearán situaciones especiales. Para ello es necesario conocer muy en detalle la estructura del módulo y tener en cuenta alguna experiencia previa.

ESTIMACIÓN DE 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 iniciales que para los errores deliberados.

$$E_E = (E_A - E_D) * (E_I / E_D) = (100 - 95) * (56 / 95) \approx 3 \text{ errores}$$

Dependiendo de lo crítico que resulte el software y del resultado obtenido con esta estrategia, se debe estudiar la conveniencia de elaborar nuevos casos de prueba.

ESTRATEGIAS DE INTEGRACIÓN

En la fase de integración también aparecen nuevos errores debidos a las más diversas causas:

- Desacuerdos en la interfaz.
- Interacción indebida entre módulos.
- Imprecisiones acumuladas.
- ...

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

Estrategias básicas de integración:

- **INTEGRACIÓN BIG BANG**

Realizar la integración de todas las unidades en un único paso. La cantidad de errores que aparecen de golpe puede hacer casi imposible la identificación de los defectos que los causan. Auténtico caos. Sólo para sistemas muy pequeños.

La ventaja fundamental es que se evita la realización de software de andamiaje.

- **INTEGRACIÓN DESCENDENTE (Top-Down)**

Se parte del módulo principal que se prueba con módulos de andamiaje o sustitutos (stubs) de los otros módulos usados directamente.

Los módulos sustitutos se van reemplazando, uno por uno, por los verdaderos y se realizan las pruebas de integración correspondientes.

La forma de realizar la sustitución puede ser por niveles, ramas o una mezcla.

La codificación de los sustitutos es un trabajo adicional que conviene simplificar al máximo. Distintas opciones:

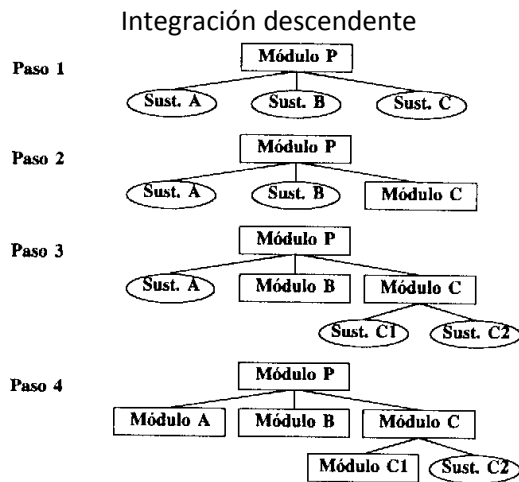
- No hacer nada y que sólo sirva para comprobar la interfaz.
- Imprimir un mensaje de seguimiento con información de la llamada.
- Suministrar un resultado fijo.
- Suministrar un resultado aleatorio.
- Suministrar un resultado tabulado u obtenido con un algoritmo simplificado.

Ventajas:

- Se ven desde el principio las posibilidades de la aplicación.
- Permite mostrar muy pronto al cliente un prototipo sencillo

Inconvenientes:

- La integración estrictamente descendente limita en cierta forma el trabajo en paralelo
- Al conducir la integración de los nuevos módulos desde otros módulos ya integrados y definitivos se tienen bastantes limitaciones para hacer pruebas especiales o dirigidas a un objetivo específico. Para lograr esto es necesario desarrollar nuevos módulos o hacer una integración híbrida ascendente-descendente.

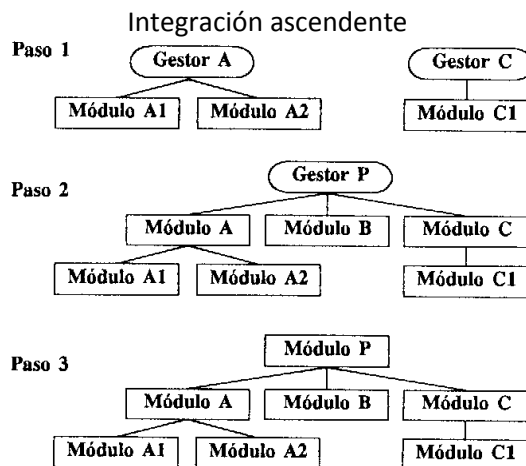


- **INTEGRACIÓN ASCENDENTE** (Bottom-Up)

Se empieza a codificar por separado y en paralelo todos los módulos de nivel más bajo. Para probarlos se escriben módulos gestores o conductores (drivers) que los hacen funcionar independientemente o en combinaciones sencillas.

Los gestores se van sustituyendo uno a uno por los módulos de mayor nivel según se van codificando, al tiempo que se van desarrollando nuevos gestores si hace falta.

El orden de sustitución puede ser cualquiera salvo para los últimos pasos en que se necesita que todos los módulos inferiores estén disponibles.



En algunos casos se puede prescindir de los gestores, su interés radica en su capacidad para probar de forma explícita situaciones especiales o infrecuentes.

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

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

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

La mejor solución es adoptar una integración ascendente con los módulos de nivel más bajo y una integración descendente con los de nivel más alto. Integración sandwich.

PRUEBAS DEL SISTEMA

Probar el sistema completo para ver si verdaderamente cumple las especificaciones en un entorno real de trabajo.

Se suele aplicar una estrategia de caja negra al sistema en su conjunto.

OBJETIVO DE LAS PRUEBAS

Según el objetivo perseguido:

- **PRUEBAS DE RECUPERACIÓN**
Comprobar la capacidad del sistema para recuperarse ante fallos. Además de provocar el fallo, se debe comprobar si el sistema lo detecta, lo corrige cuando así está especificado y si el tiempo de recuperación es menor del también especificado.
- **PRUEBAS DE SEGURIDAD**
Comprobar los mecanismos de protección contra un acceso o manipulación no autorizada.
- **PRUEBAS DE RESISTENCIA**
Comprobar el comportamiento del sistema ante situaciones excepcionales. Deberán forzar el sistema por encima de su capacidad y prestaciones normales para verificar cuál es su comportamiento y cómo se va degradando en estos casos.
- **PRUEBAS DE SENSIBILIDAD**
Comprobar el tratamiento que da el sistema a ciertas singularidades relacionadas con los algoritmos matemáticos utilizados. Buscar combinaciones de datos que puedan provocar alguna operación incorrecta o poco precisa.
- **PRUEBAS DE RENDIMIENTO**
Comprobar las prestaciones del sistema que son críticas en tiempo.
Fundamentales para los sistemas de tiempo real.
Para medir los tiempos se suelen usar equipos de instrumentación externos (emuladores, analizadores lógicos, osciloscopios, etc.)

PRUEBAS ALFA Y BETA

Para comprobar que un producto software es realmente útil para sus usuarios es conveniente que estos intervengan también en las pruebas finales del sistema.

Se pueden poner de manifiesto nuevas deficiencias no caracterizadas hasta entonces.

- **Pruebas alfa**

Primeras pruebas que se realizan en un entorno controlado donde el usuario tiene el apoyo de alguna persona del equipo de desarrollo y a su vez esta misma persona puede seguir muy de cerca la evolución de las pruebas.

- **Pruebas beta**

Uno o varios usuarios trabajan con el sistema en su entorno normal, sin apoyo de nadie, y anotando cualquier problema que se presente.

Es importante que sea el usuario el encargado de transmitir al equipo de desarrollo cuál ha sido el procedimiento de operación que le llevó al error.