Tema 3

Lenguajes de programación. Representación de tipos de datos. Operadores. Instrucciones condicionales. Bucles y recursividad. Procedimientos, funciones y parámetros. Vectores y registros. Estructura de un programa.

Guion-resumen

1. **Lenguaje de Programación**
   - 1.1. Introducción
   - 1.2. Concepto de programa
   - 1.3. Características de los programas
   - 1.4. Los lenguajes de programación
2. **Elementos de un programa**
   - 2.1. Objetos
   - 2.2. Identificadores
   - 2.3. Datos
   - 2.4. Constantes
   - 2.5. Variables
   - 2.6. Operadores
   - 2.7. Expresiones
   - 2.8. Sentencias
   - 2.9. Comentarios
3. **Tipos de Datos**
   - 3.1. Datos básicos
   - 3.2. Dato derivado (puntero)
   - 3.3. Datos estructurados
4. **Operadores**
5. **Clasificación general de instrucciones**
   - 5.1. Instrucciones de definición de datos
   - 5.2. Instrucciones primitivas
   - 5.3. Instrucciones compuestas
   - 5.4. Instrucciones de control
6. **Recursividad**
7. **Procedimientos, funciones y parámetros**
   - 7.1. Procedimientos
   - 7.2. Funciones
   - 7.3. Parámetros
8. **Vectores y registros**
   - 8.1. Vectores
   - 8.2. Registros
9. **Estructura de un programa**
   - 9.1. Elementos auxiliares de programación
   - 9.2. Prueba de programas
   - 9.3. Verificación y validación
   - 9.4. Otras pruebas

1. Lenguaje de Programación

1.1. Introducción

Los ordenadores son máquinas que disponen de gran rapidez para efectuar operaciones, poseen precisión y memoria pero son carentes de inteligencia natural. Son máquinas preparadas para realizar el proceso que se les indique mediante un programa (un conjunto de instrucciones). Estas instrucciones que forman el programa deben traducirse de un lenguaje comprensible por los humanos a un lenguaje comprensible para la máquina. El componente principal del **ordenador**, el único que realiza el trabajo de cálculo (computación), es la Unidad Central de Proceso (CPU). **Es la CPU la que se encarga de ejecutar los programas.**

1.2. Concepto de programa

Un programa es una serie o secuencia de instrucciones que el ordenador debe ejecutar para realizar la tarea prevista por el programador. Cuando nos planteamos un problema complejo y queremos resolverlo con la utilización del ordenador, necesitamos descomponerlo en una serie de tareas simples que se irán repitiendo a lo largo de un proceso hasta la resolución del problema; el ordenador ha realizado una tarea compleja a partir de instrucciones simples. Dicho conjunto de tareas simples es el programa y su elaboración es lo que entendemos por programación.

Cuando nos disponemos a programar una de las primeras decisiones que hemos de tomar es la elección del **lenguaje** a emplear, **es decir, la forma en la que el programador tiene que escribir las operaciones a realizar por el ordenador.**

Veamos a continuación algunas definiciones útiles para seguir avanzando:

— Se denomina **instrucción** al conjunto de reglas o normas dadas para la realización o empleo de algo. En informática, instrucción es la información que indica a un ordenador una acción elemental que ha de realizar. Para poder realizar algún proceso de utilidad debe indicarse al ordenador un conjunto organizado de instrucciones.

— Un **algoritmo** es un conjunto ordenado de operaciones necesarias para resolver un problema.

— Un **programa** es un conjunto ordenado de instrucciones, perfectamente legibles por el ordenador, que le permiten realizar un trabajo o resolver un problema. El programa es la descripción de un algoritmo en un lenguaje inteligible por la máquina.

Los programas contienen frecuentemente conjuntos de instrucciones que pueden intervenir varias veces en la ejecución del mismo o que realizan una tarea específica. Es posible agrupar dichas instrucciones de modo que formen una unidad independiente del programa a la cual se haga referencia en aquellos puntos del programa donde sea necesario. En esto consiste una **subrutina**. En la traducción del programa a lenguaje máquina se establecerá el mecanismo para que al encontrar la referencia a ese conjunto de instrucciones, la subrutina, aquellas se ejecuten como si estuvie-

ran en el propio programa. Las subrutinas pueden ser llamadas por un solo programa pero también puede hacerse referencia a las mismas desde programas distintos.

1.3. Características de los programas

La determinación de la calidad de los programas estará en función de las ventajas de su utilización; para ello existen unas características que determinan a priori si el programa tendrá vida larga y productiva:

— **Legible**. Todo programa debe ser de fácil comprensión no solo por los futuros usuarios, sino por cualquier programador.

— **Flexible**. Capaz de adaptarse con facilidad a los cambios que puedan producirse en el planteamiento inicial.

— **Portable**. Facilidad para compilarse o interpretarse en distintas máquinas y sistemas operativos, también un factor a tener en cuenta sería su facilidad para ser traducido a otros lenguajes de programación.

— **Fiable**. El programa debe ser capaz de recuperar el control cuando su utilización no sea la adecuada.

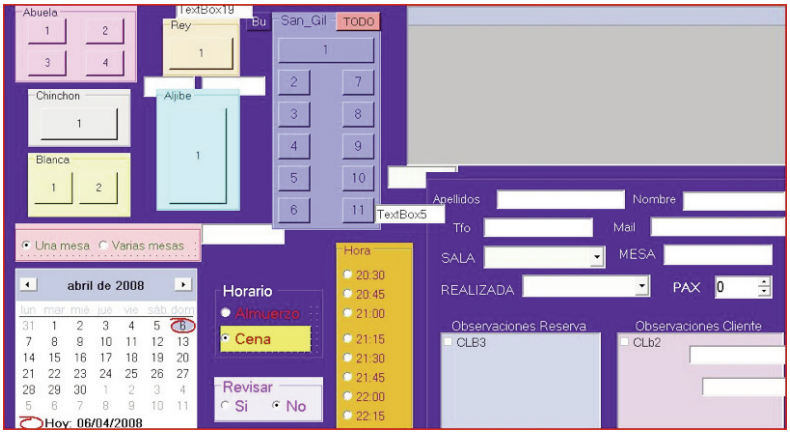
— **Eficaz**. El programa ha de utilizar eficazmente los recursos de que disponga, tanto del sistema operativo como del equipo en el que trabaje.

Características de los programas:	
Legibilidad	(comprendido por diferentes programadores)
Flexibilidad	(adaptarse a los cambios)
Portabilidad	(compilable diferentes máquinas / S.O / lenguajes)
Fiabilidad	(recuperación del control por errores)
Eficiencia	(aprovechamiento recursos S.O.)

1.4. Los lenguajes de programación

El **lenguaje de programación es la forma en la que el programador escribe las operaciones que el ordenador debe realizar**. Es el conjunto de notaciones, símbolos y reglas sintácticas para posibilitar la escritura de un algoritmo que posteriormente será interpretado por el hardware de un ordenador.

La CPU esta preparada para manejar unas instrucciones escritas en un tipo de lenguaje muy simple llamado **lenguaje máquina**. Cada modelo de CPU posee su propio lenguaje máquina, y puede ejecutar un programa solo si está escrito en ese lenguaje.
Para poder ejecutar programas escritos en otros lenguajes, es necesario primero trasladarlos a lenguaje máquina a través de un proceso de compilación o interpretación.



Interfaz grafica del programa

```
    CLB3.Enabled = True
    CLb2.Enabled = True
    TextBox34.Enabled = True
    fofo66()
    Button179.BackColor = System.Drawing.Color.Silver
    Button179.Enabled = False
    MC3.Visible = False
End Sub

Private Sub TextBox133_TextChanged(ByVal sender As System.Object, ByVal e As System.EventArgs) Handles TextBox133.TextChanged
    If i_reservas = 1 Then
        DATASET_CLIENTES2.Clear()
        OleDbConnection1.Close()
        OleDbConnection1.Open()
        OleDbDataAdapter20.SelectCommand.Parameters.Item(0).Value = "%" & TextBox136.Text & "%"
        OleDbDataAdapter20.SelectCommand.Parameters.Item(1).Value = "%" & TextBox134.Text & "%"
        OleDbDataAdapter20.SelectCommand.Parameters.Item(2).Value = "%" & TextBox133.Text & "%"
        OleDbDataAdapter20.Fill(DATASET_CLIENTES2, "CLIENTES")

        DataGrid7.DataSource = DATASET_CLIENTES2
        DataGrid7.DataMember = "CLIENTES"

        OleDbConnection1.Close()
    End If
    TextBox133.BackColor = System.Drawing.Color.White
    conf_reserva()
End Sub
Private Sub TextBox134_TextChanged(ByVal sender As System.Object, ByVal e As System.EventArgs) Handles TextBox134.TextChanged
    If i_reservas = 1 Then
        DATASET_CLIENTES2.Clear()
        OleDbConnection1.Close()
        OleDbConnection1.Open()
```

Código fuente del programa

Un lenguaje de programación es una notación o conjunto de símbolos y caracteres combinados conforme una sintaxis ya predefinida. En función de su parecido con el lenguaje natural, podemos hablar de lenguajes de bajo nivel y lenguajes de alto nivel. En los primeros la sintaxis está más próxima al lenguaje máquina que al lenguaje humano y en los de alto nivel es todo lo contrario. Cuando un programa es ejecutado directamente por el ordenador, es decir, está en código máquina, decimos que es un lenguaje de bajo nivel. Casi todos los programas son escritos por los programadores en lenguajes de alto nivel.

Nota. *Los lenguajes de programación, en especial C, C++, Java y la plataforma .Net serán estudiados con detalle en el tema 6.*

2. Elementos de un programa

En la elaboración de un programa utilizamos diferentes elementos que cubrirán diferentes funciones. Los diferentes elementos son:

— Objetos.

— Identificadores.

— Datos.

— Constantes.

— Variables.

— Operadores.

— Expresiones.

— Sentencias.

— Comentarios.

2.1. Objetos

Llamaremos objetos a todos los elementos que utilizaremos en la programación, susceptibles de ser manejados por las instrucciones y sentencias del programa.

Todas las cosas pueden ser objetos, siempre que puedan ser individualizables e identificables, pudiendo ser cosas reales o abstractas, pero siempre representarán un papel definido en el problema.

Los objetos tendrán tres atributos que los diferencien: el nombre que los identifique e individualice; el tipo, que determina el contenido o la clase de objeto; y su valor, que será el contenido en concreto que lo distingue de otros objetos de la misma clase.

2.2. Identificadores

Los identificadores o etiquetas son palabras escogidas por el programador para designar los distintos elementos de un programa tales como las variables, funciones, etc.

2.3. Datos

Consideramos como datos toda la información que se va a procesar en el ordenador. Serían datos los objetos anteriormente mencionados susceptibles de manipulación por las instrucciones del programa; también consideramos como datos aquellos elementos simples que utilizamos para comunicarnos, pudiendo ser de tipo:

— **Carácter**. Son las unidades utilizadas en la información:

  • Alfabéticas (a, A, b, B, ...).

  • Numéricas (0, 1, 2, ...).

  • Especiales (+, -, *, ...).

— **Numérico**. Son los valores o cantidades representados por los caracteres numéricos. Pueden ser de tipo entero o real.

— **Alfanumérico**. Son conjuntos de caracteres alfabéticos, numéricos o especiales.

— **Lógico o booleano**. Son datos lógicos que solo pueden tomar dos valores: 1/0, cierto/falso, sí/no, ...

2.4. Constantes

Las constantes son aquellos datos utilizados en los programas que permanecerán invariables en todo el proceso de ejecución del programa, no pudiéndose alterar su valor o composición ni por parte del usuario ni del ordenador.

2.5. Variables

Las variables son datos cuyo valor es modificable durante la ejecución del programa. Dicha variación solo afectará al valor de la variable, no así al lugar que ocupe en la memoria del ordenador, al tipo de información que represente ni a la identificación de la misma. Son datos cuya información va a ir cambiando a lo largo de la ejecución del programa. Pueden ser de diversos tipos:

— **Inicializadas o no inicializada**, según se les asigne o no un valor inicial de partida.

— **Globales o locales**, según tengan validez para todo el programa o solo para una parte del mismo.

2.6. Operadores

Los operadores son símbolos que representan las distintas operaciones que pueden realizarse sobre los datos. Pueden clasificarse en:

— **Aritméticos**: suma, resta, multiplicación, etc.

— **Alfanuméricos**: concatenación.

— **De asignación**: establece los valores de las variables.

— **De relación**: permiten comparaciones: igual que, menor o igual que...

— **Lógicos**: AND, OR, OR exclusiva, NOT...

2.7. Expresiones

En programación, una expresión es una combinación de operadores y datos u objetos (constantes o variables) cuyo resultado es un valor, que puede, por ejemplo, ser asignado a una variable.

Los conceptos que estamos considerando (expresiones, variables, constantes, etc.) son muy similares a los convenios matemáticos.

2.8. Sentencias

Las sentencias son las actuaciones que deseamos que ejecute el programa; se corresponderán con las operaciones estudiadas en el algoritmo, y estarán de acuerdo con la sintaxis establecida en el lenguaje concreto de programación que se esté utilizando.

Podemos clasificar las sentencias utilizadas en la programación como instrucciones de:

— **Asignación**. Fijarán el valor que deban tomar las variables en un momento determinado.

— **Entrada**. Permitirán la recogida de datos desde un dispositivo externo: teclado, ratón, etc., asignando valores a las variables.

— **Salida**. Lo utilizaremos para visualizar externamente los valores o datos del programa: pantalla, archivo, impresora, etc.

— **Condicionales**. Nos permitirán la comparación del valor de las variables, posibilitando la toma de decisiones por parte del ordenador.

— **Repetitivas**. Son aquellas que posibilitarán la repetición de un conjunto de sentencias; dicha repetición se realizará un determinado número de veces, o mientras se cumpla una condición.

— **Salto**.

2.9. Comentarios

Son líneas de texto que el compilador o el intérprete no consideran como parte del código, con lo cual no están sujetas a restricciones de sintaxis y sirven para aclarar partes de código en posteriores lecturas y, en general, para anotar cualquier cosa que el programador considere oportuno. Un programador debe tener como prioridad documentar el código fuente ya que al momento de depurar ahorrará mucho tiempo de análisis para su corrección o estudio.

Se deben documentar los programas con encabezados de texto (encabezados de comentarios) en donde se describen la función que va a realizar dicho programa, la fecha de creación, el nombre del autor y, en algunos casos, las fechas de revisión y el nombre del revisor. Se puede hacer uso de llamadas a subprogramas dentro de una misma aplicación por lo que cada subprograma

debería estar documentado, describiendo la función que realiza cada uno de estos subprogramas dentro de la aplicación.

3. Tipos de datos

Se llama **estructura de datos** al conjunto de datos con una misma denominación y que se utilizan como una sola unidad. Las estructuras de datos pueden ser internas o externas dependiendo del lugar de almacenamiento, memoria central del ordenador o dispositivo de almacenamiento externo:

— **Estructuras de datos internas**. Los arrays son estructuras de datos internas, compuestos por un número fijo de elementos del mismo tipo; son almacenados en posiciones consecutivas de memoria y pueden ser llamados como una sola unidad o conjunto compacto, pero también pueden ser llamados como variables independientes considerándolos de forma individualizada.

— **Estructuras de datos externas**, son archivos o ficheros almacenados en un dispositivo de almacenamiento externo; son necesarios como forma de almacenamiento de información de forma masiva y permanente, necesarios para el funcionamiento de un programa.

  Dichos archivos se componen de registros equivalentes a fichas con los datos que componen la información a tratar; los archivos son conjuntos de registros del mismo tipo que el programa utilizará y modificará a medida que se vaya ejecutando.

Para el diseño de un programa es importante establecer cuáles son las estructuras de los datos que se van a utilizar, con el objeto de establecer las operaciones que sobre dichos datos se pueden realizar. Para ello, debemos proporcionar al sistema información sobre los mismos. Es decir, que los datos manejados en un programa deben llevar asociados un identificador, un tipo y un valor.

- **Identificador**. Es el nombre utilizado en un programa para referenciar un dato. Existen ciertas normas generales para su empleo, siendo posible destacar las siguientes:

  a) Pueden estar constituidos por letras y dígitos y en algunos casos por el carácter subrayado (_).

  b) Deben comenzar por una letra.

  c) No deben contener espacios.

  d) El número máximo de caracteres y nombres reservados que se pueden emplear dependen del compilador utilizado.

  e) El nombre asignado debe tener relación con la información que contiene, pudiéndose emplear abreviaturas que sean significativas.

- **Tipo**. Establece el rango o intervalo de valores que puede tomar el dato. El tipo determina el espacio de memoria que se reservará para el dato. Un tipo de datos define un conjunto de valores y las operaciones sobre estos valores. Casi todos los lenguajes de programación explícitamente incluyen la notación del tipo de datos, aunque lenguajes diferentes pueden usar terminología diferente. La mayor parte de estos lenguajes permiten al programador definir tipos de datos adicionales, normalmente combinando múltiples elementos de otros tipos y definiendo las operaciones del nuevo tipo de dato.
- **Valor**. Elemento que debe pertenecer al rango o intervalo de valores según el tipo definido.

3.1. Datos básicos

— **Numéricos**. Se utilizan para contener magnitudes y pueden ser de dos tipos: enteros y reales.

— **Carácter**. Se emplean para representar un carácter dentro de un conjunto definido por el fabricante del equipo, de forma que cada uno de ellos se corresponde con un valor numérico entero sin signo, siguiendo un determinado código (EBCDIC, ASCII...).

— **Lógico**. Se emplean para representar únicamente dos valores, 1 o 0, Verdadero o Falso, etc.

3.2. Dato derivado (puntero)

Se emplea para contener la dirección de memoria de otra variable. Una variable tipo puntero debe ser definida con el mismo tipo de la variable que va a referenciar o apuntar. Este tipo de variable es de gran utilidad para realizar operaciones con estructuras y para llamadas a módulos.

3.3. Datos estructurados

Podemos encontrar varios tipos de datos estructurados:

- **Datos internos**

  Son los que residen en la memoria principal del ordenador. Por ejemplo, una tabla bidimensional de números reales.

- **Datos externos**

  Son los que residen en un soporte externo a la memoria principal, es decir, memoria auxiliar; por ejemplo un fichero guardado en disco duro.

- **Datos estáticos**

  Son aquellos cuyo tamaño queda definido en la compilación del programa y no se puede modificar durante la ejecución del mismo. Por ejemplo una tabla de alumnos.

- **Datos dinámicos**

  Son aquellos cuyo tamaño puede ser modificado durante la ejecución del programa. Por ejemplo, una lista, una pila, una cola, etc.

- **Datos lineales**

  Son los que pueden estar enlazados con un solo elemento anterior y uno solo posterior. Por ejemplo una pila.

- **Datos no lineales**

  Son los que pueden enlazarse con más de un elemento anterior y más de uno posterior. Por ejemplo, un árbol.

- **Datos compuestos**

  Son los formados por el programador a base de tipos de datos básicos y derivados, pudiendo ser internos o externos.

<table>
<tr><td rowspan="3">Datos Básicos</td><td colspan="3">Numéricos</td><td>Entero
Real</td></tr>
<tr><td colspan="3">Carácter</td><td></td></tr>
<tr><td colspan="3">Lógico</td><td></td></tr>
<tr><td>Dato derivado</td><td colspan="3"></td><td>Puntero</td></tr>
<tr><td rowspan="3">Datos estructurados</td><td rowspan="3">Internos</td><td>Estáticos</td><td>Lineales</td><td>Tabla</td></tr>
<tr><td rowspan="2">Dinámicos</td><td>Linales</td><td>Lista
Pila
Cola</td></tr>
<tr><td>No lineales</td><td>Árbol
Grafo</td></tr>
<tr><td>Externos</td><td colspan="2"></td><td>Fichero Base de datos</td></tr>
</table>

4. Operadores

Los operadores son símbolos que sirven para conectar los datos facilitando la realización de diversas clases de operaciones.

Pueden ser:

OPERADOR	SÍMBOLO	SIGNIFICADO
Paréntesis	( )	Paréntesis
Aritméticos	** , ^ * / div , \ % , mod + -	Potencia Producto División División entera Módulo (resto de la división entera) Signo positivo o suma Signo negativo o resta
Alfanuméricos	+ -	Concatenación Concatenación eliminando espacios
Relacionales	= = , = ! = , < > < < = > > =	Igual a Distinto a Menor que Menor o igual que Mayor que Mayor o igual que
Lógicos	! , NOT, no &&, AND, y 11 , OR, o	Negación Conjunción Disyunción

- **Orden de prioridad de los operadores**

Dentro de las expresiones hay que respetar un orden de prioridad entre operadores que depende del lenguaje utilizado, pero de forma general se puede establecer el siguiente orden:

1. Paréntesis (comenzando por los más internos).
2. Signo.
3. Potencia.
4. Producto, división y módulo (con la misma prioridad).
5. Suma y resta (con la misma prioridad).
6. Concatenación.
7. Relacionales.
8. Negación.
9. Conjunción.
10. Disyunción.

5. Clasificación general de instrucciones

Una instrucción puede ser considerada como **un hecho o suceso de duración limitada** que genera unos cambios previstos en la ejecución de un programa, por lo que debe ser una acción previamente estudiada y definida.

5.1. Instrucciones de definición de datos

Son aquellas instrucciones utilizadas para informar al procesador del espacio que debe reservar en memoria, con la finalidad de almacenar un dato mediante el uso de variables simples o estructuras de datos más complejas como, por ejemplo, tablas.

La definición consiste en indicar un nombre a través del cual haremos referencia al dato y un tipo a través del cual informaremos al procesador de las características y espacio que deberá reservar en memoria.

Ejemplo.- int x;

5.2. Instrucciones primitivas

Se consideran como tal las instrucciones de asignación y las instrucciones de entrada/salida.

- **Instrucciones de entrada**

  Son aquellas instrucciones encargadas de recoger el dato de un periférico o dispositivo de entrada (por ejemplo el teclado, un ratón, un escáner, etc.) y seguidamente almacenarlo en memoria en una variable previamente definida, para la cual se ha reservado suficiente espacio en memoria.

  Ejemplo.- scanf ("%d", &x);

  En el supuesto de leer varios valores consecutivos, con la intención de almacenarlos en variables diferentes, lo indicaremos situando uno a continuación del otro separados por comas.

- **Instrucciones de asignación**

  Son aquellas instrucciones cuyo cometido es almacenar un dato o valor simple obtenido como resultado de evaluar una expresión, en una variable previamente definida y declarada.

  Ejemplo.- x=9;

- **Instrucciones de salida**

  Son aquellas instrucciones encargadas de recoger datos procedentes de variables o los resultados obtenidos de expresiones evaluadas y depositarlos en un periférico o dispositivo de salida (por ejemplo, la pantalla, una impresora, un plóter, etc.).

  Ejemplo.- printf ("El resultado es: %d", x);

En el supuesto de escribir varios valores o resultados de forma consecutiva, lo indicaremos situando uno a continuación del otro y separados por comas, de la misma forma que ocurría con las instrucciones de entrada.

5.3. Instrucciones compuestas

Son aquellas instrucciones que no pueden ser ejecutadas directamente por el procesador, y están constituidas por un bloque de acciones agrupadas en subrutinas, subprogramas, funciones o módulos.

5.4. Instrucciones de control

Son utilizadas para controlar la secuencia de ejecución de un programa, así como determinados bloques de instrucciones.

Se clasifican en:

5.4.1. Instrucciones de Salto

Son aquellas que alteran o rompen la secuencia normal de ejecución de un programa, permitiendo la posibilidad de retornar el control de ejecución al punto de llamada. Pueden ser:

- **Condicionales**

Son aquellas que alteran la secuencia de ejecución solo cuando una condición especificada sea cierta.

```
INICIO
   DATOS:
      VARIABLES
         Long              Numérico Real
         AreaCuadrado      Numérico Real

   ALGORITMO:
         Leer Long
         Si Long <= 0
               GOTO Error
         FinSi
         AreaCuadrado = Long * Long
         Escribir "El valor del Área es:", AreaCuadrado
         ...
   Error:
         Escribir "El valor introducido no es correcto."
FIN
```

- **Incondicionales**

Son aquellas que alteran la secuencia normal de ejecución siempre, ya que no van acompañadas de ninguna condición para que se produzcan.

```
INICIO
   DATOS:
      VARIABLES
         Long              Numérico Real
         AreaCuadrado      Numérico Real

   ALGORITMO:
      Inicio:
      Leer Long
      AreaCuadrado = Long * Long
      Escribir "El valor del Área es:", AreaCuadrado
            ...
            GOTO Inicio
FIN
```

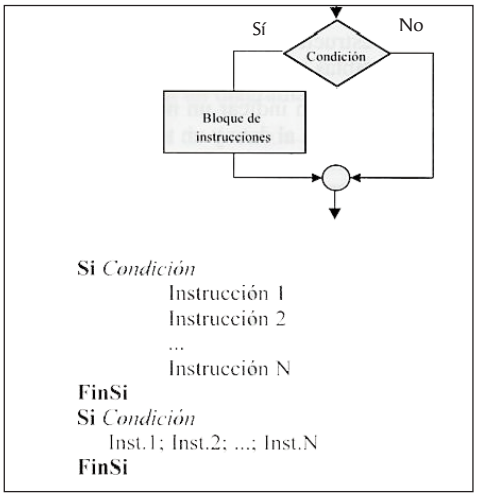
Es necesario indicar, que si bien muchos lenguajes de programación las mantienen entre sus instrucciones por mera compatibilidad, no es deontológicamente correcto utilizarlas porque rompen toda lógica de seguimiento del programa cuando hablamos de programación, tanto estructurada como orientada a objetos. En la actualidad estas instrucciones son utilizadas en el lenguaje ensamblador que utiliza las instrucciones JMP, que es el acrónimo de salto.

5.4.2. Instrucciones condicionales

Son aquellas que controlan la ejecución o la no ejecución de una o más instrucciones en función de que se cumpla o no una condición previamente establecida.

- Alternativa simple

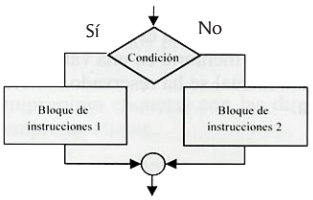
```
if (condición) {
  instrucciones
}
```




```
Si Condición
        Instrucción 1
        Instrucción 2
        ...
        Instrucción N
FinSi
Si Condición
   Inst.1; Inst.2; ...; Inst.N
FinSi
```

- **Alternativa doble**

```
if(condición){
  instrucciones
}else{
  instrucciones
}
```



Si *Condición*
Instrucción 1A
Instrucción 1B
...
Instrucción 1N
Sino
Instrucción 2A
Instrucción 2B
...
Instrucción 2N
FinSi

```
if(condición){
   instrucciones
}
else if(condición){
   instrucciones
}
else if(condición){
   instrucciones
}
...
else {
   instrucciones
}
```

```
switch(x){
        case 1:
                instrucciones
                break;
        case 2:
                instrucciones
                break;
        ...
        case n:
                instrucciones
                break;
        default:
                break;
}
```

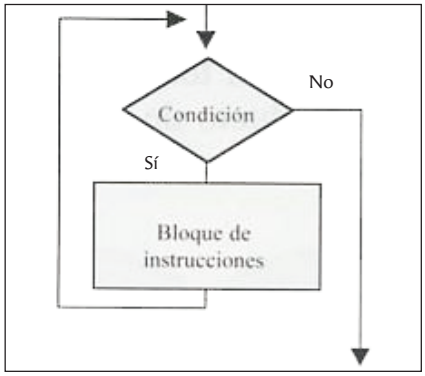
- **Alternativa múltiple**

Es similar a la doble pero con más de dos posibilidades de salida del flujo del programa.

- **Alternativa anidada (condicionales anidados)**

```
if(condición){
  instrucciones
    if(condición){
          instrucciones
          if(condición){
                instrucciones
          }
    }
}
```

5.4.3. Instrucciones repetitivas (bucles)

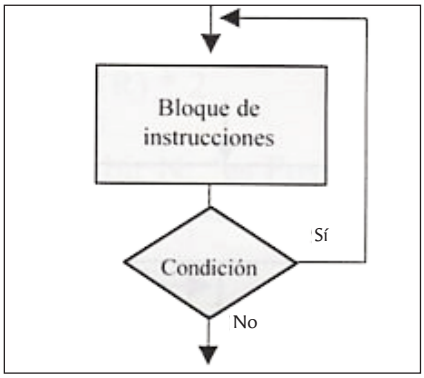


Son aquellas que permiten variar o alterar la secuencia de un programa haciendo posible que un grupo de acciones se ejecute más de una vez de forma consecutiva. Reciben el nombre de bucles.

- **Estructura Mientras**

```
while(condicion) {
    ...
    instrucciones
    ....
}
```

Permite ejecutar un bloque de instrucciones entre 0 y n veces.



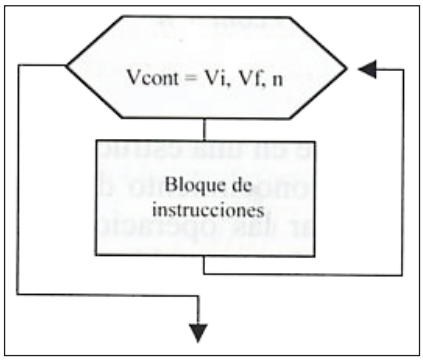
- **Estructura Repetir-Mientras**

```
do{
    ...
    instrucciones
    ...
} while(condición);
```

Permite ejecutar un bloque de instrucciones entre 1 y n veces.

- **Estructura Para**

```
for (inicialización; condición _
final; incremento o decremento){
    ...
    instrucciones
    ...
}
```



Permite ejecutar un bloque de instrucciones una cantidad de veces fijada de antemano.

6. Recursividad

La recursividad es una técnica muy potente de programación que puede utilizarse en sustitución de las iteraciones (instrucciones repetitivas) o bucles en ciertos problemas. Esta técnica consiste, básicamente, en **permitir que un subprograma se llame a sí mismo.**

El uso de la recursividad es adecuado en problemas a resolver que tengan una clara definición recursiva (como, por ejemplo, el factorial de un número) y permite desarrollar programas más simples y elegantes. A pesar de ello la recursividad es menos eficiente que los bucles.

Diremos entonces que un método es recursivo si entre sus instrucciones tiene una llamada a sí mismo, en la cual mantiene una pila con los valores procesados.

```
función f (parámetros) {
    ...
    instrucciones;
    ...
    f(parámetros);
}
```

Ejemplo del Cálculo del factorial de 5 escrito en Java:

```
public class Factoriales {
  static int factorial(int numero){
    if ( numero <= 1 ) {
      return 1;
    } else {
      return numero*factorial(numero-1);
    }
  }
  public static void main(String args[]){
    System.out.println(factorial(5));
  }
}
```

7. Procedimientos, funciones y parámetros

El **diseño descendente** o *top-down* consiste en una serie de descomposiciones sucesivas del programa inicial, que describen el refinamiento progresi-

vo del repertorio de instrucciones que configuran un programa. Un programa diseñado con *top-down* quedará claramente constituido por dos partes claramente diferenciadas:

- **Programa principal:** describe la solución completa del problema y consta principalmente de llamadas a subprogramas. También puede contener otras instrucciones.
- **Subprogramas:** se encuentran agrupados en diferente lugar que el programa principal. Su estructura básica coincide con la de un programa, con algunas diferencias en su encabezamiento y finalización. La función de un subprograma es resolver de modo independiente una parte del problema. Un subprograma solo se ejecuta cuando es llamado por el programa principal o por otro subprograma.

Esta división en módulos se debe terminar cuando en los módulos se definan tareas específicas a realizar. La parte de interconexión entre las subrutinas no debe tener problemas si el paso de datos (parámetros) entre ellos se especifica claramente.

Estos módulos pueden estar ya creados por otros programadores y almacenados en librerías, con lo cual nosotros solo tendríamos que invocarlos para poder usarlos.

Como consecuencia de esta técnica de diseño surgen conceptos como:

— Parámetros.

— Procedimientos.

— Funciones.

— Recursividad.

7.1. Procedimientos

Un procedimiento es un subprograma que realiza una tarea específica y que puede ser definido mediante cero, uno o "n" parámetros. Tanto la entrada como la devolución de resultados desde el procedimiento al programa llamador se realizará a través de los parámetros. El nombre de un procedimiento no está asociado a ninguno de los resultados que obtiene.

Esta compuesto por un grupo de sentencias a las que se asigna un nombre (identificador) y constituye una unidad de programa. La tarea asignada al procedimiento se ejecuta siempre que se encuentra el nombre del procedimiento.

La declaración indica las instrucciones a ejecutar. Su sintaxis es:

```
procedimiento nombreproc (lista de parametros)
declaraciones locales
inicio
        cuerpo del procedimiento (instrucciones)
fin.
```

Un procedimiento es llamado en un programa o dentro de otro procedimiento directamente por su nombre. Supongamos que queremos calcular el valor medio de tres datos. Definimos un procedimiento "lolo":

```
Void lolo(par1, par2, par3){ //inicio procedimiento
aux= (par1+ par2+ par3)/3
printf (" El resultado es: ", aux);
}fin procedimiento
```

Este supuesto programa nos imprime la media de tres elementos. Nos permite su utilización sistemática tantas veces como queramos sin necesidad de escribir las instrucciones tantas veces como veces queremos utilizarla. Podría ser utilizado en cualquier lugar de nuestro programa haciendo una llamada del siguiente tipo: lolo(7,6,2) y nos imprimirá en pantalla: 5. Pero en este caso el modulo que ha llamado a este procedimiento no recibe ningún valor de retorno.

7.2. Funciones

Una función es un subprograma que realiza una tarea específica y devuelve un resultado en el propio nombre de la función. Podría ser definida como un conjunto de instrucciones que permiten procesar las variables para obtener un resultado. Recibe como parámetros (argumentos) datos y devuelve un único resultado. Esta característica le diferencia esencialmente de un procedimiento. Su formato es el siguiente:

```
tipo_develto funcion nombrefuncion (p1,p2,...) :
  declaraciones locales
inicio
  cuerpo de la función
  nombrefuncion // valor a devolver
fin
```

Una función es llamada por medio de su nombre, en una sentencia de asignación o en una sentencia de salida. Supongamos que queremos calcular el valor medio de tres datos. Definimos una función "lolo":

```
int lolo(par1, par2, par3){ //inicio función
aux= (par1+ par2+ par3)/3
return(aux)
}fin función
```

Esta función suma podría ser utilizada en cualquier lugar de nuestro programa haciendo una llamada del siguiente tipo: lolo(7,6,2)

El modulo llamante recibiría como valor de retorno: 5

7.3. Parámetros

Un subprograma puede realizar, al igual que un programa, operaciones de E/S. Sin embargo, es frecuente que sus datos de entrada y sus resultados provengan y sean enviados del y al programa o subprograma llamante, respectivamente.

Para ello, se utilizan los parámetros. Cada vez que se realiza una llamada a un subprograma, los datos de entrada le son pasados por medio de una variable y, de forma análoga, cuando termina la ejecución del subprograma, los resultados se devuelven mediante esas mismas variables o mediante otras.

Existen dos **tipos de parámetros**:

- **Parámetros formales:** variables locales de un subprograma, utilizados tanto en la emisión de datos como en la recepción.
- **Parámetros actuales:** variables y datos enviados en cada llamada de subprograma, por el programa o subprograma llamante.

Los parámetros formales son siempre fijos para cada subprograma, mientras que los parámetros actuales pueden ser modificados en cada llamada. En cualquier caso, tiene que haber una correspondencia entre los parámetros formales y los actuales.

El proceso de emisión y recepción de datos y resultados mediante variables de enlace se denomina **paso de parámetros**.

Se pueden pasar parámetros de dos formas diferentes:

— **Por valor**: en este caso, el parámetro actual no puede ser modificado por su programa, el cual copia su valor en el parámetro formal correspondiente para poder utilizarlo.

— **Por referencia:** cuando un parámetro actual se pasa por referencia se proporciona la dirección o referencia de la variable, con lo que el subprograma llamante la utiliza como propia, modificándola si es necesario. La utilización de esta forma de pasar parámetros supone un ahorro de memoria, debido a que la variable local correspondiente no existe físicamente, sino que es asociada a la global en cada llamada. Es evidente que también supone el riesgo de modificar de forma indeseada una variable global.

8. Vectores y registros

8.1. Vectores

Un array o vector es una estructura de datos constituida por un número fijo de elementos, todos ellos del mismo tipo y ubicados en direcciones de memoria físicamente contiguas.

Las estructuras de datos cuyos elementos son del mismo tipo, con las mismas características, y que se referencian bajo un nombre o identificador común, reciben el nombre de vectores. Generalmente se definen los **vectores** como tablas de dimensión uno (unidimensionales).

Al igual que las tablas unidimensionales, una tabla bidimensional es un conjunto de elementos del mismo tipo y características, que se referencian bajo un mismo nombre. Este tipo de estructuras también reciben el nombre de **matrices**.

También reciben el nombre de **poliedros** aquellas tablas de tres o más dimensiones que al igual que las tablas unidimensionales y bidimensionales están constituidas por elementos del mismo tipo y características.

8.2. Registros

Como ya hemos definido en otros capítulos un fichero es un conjunto de información relacionada entre sí y estructurada en unidades más pequeñas. A estas unidades mas pequeñas en las cuales podemos descomponer un fichero se les denomina "registros" que forman un bloque que puede ser manipulado de forma unitaria. Se pueden clasificar en:

- **Registro lógico: las estructuras de datos homogéneas referentes a una misma entidad o cosa**, dividida a su vez en elementos más pequeños denominados campos que pueden ser del mismo o diferente tipo. **El registro es considerado en sí mismo como una unidad de tratamiento dentro del fichero**.

- **Registro físico:** también llamado **bloque, es la cantidad de información que el sistema puede transferir como unidad**, en una sola operación de E/S, entre la memoria principal del ordenador y los periféricos o dispositivos de almacenamiento. El tamaño del bloque o registro físico dependerá de las características del ordenador.

- **Registro bloqueado**: un registro físico puede constar de un número variable de registros lógicos. Por tanto, suponiendo que utilizáramos como soporte de almacenamiento el disco, se podrían transferir varios registros lógicos de la memoria al disco y del disco a la memoria en una sola operación de E/S. Este fenómeno recibe el nombre de bloqueo y el registro físico así formado se llama bloque. **Se conoce como factor de bloqueo al número de registros lógicos contenidos en un bloque o registro físico**.

- **Registro expandido:** es justamente el concepto contrario de registro bloqueado, es decir, cuando el registro lógico ocupa varios bloques se le da la denominación de registro expandido.

Los ficheros son estructuras de datos jerarquizadas.

9. Estructura de un programa

Todo programa está constituido por un conjunto de órdenes o instrucciones capaces de manipular un conjunto de datos. Estas órdenes o instrucciones pueden ser divididas en tres grandes bloques claramente diferenciados, correspondientes cada uno de ellos a una parte del diseño de un programa:

- **Entrada de datos**

  En este bloque se engloban todas aquellas instrucciones que toman datos de un dispositivo o periférico externo, depositándolos posteriormente en memoria central o principal para poder ser procesados.

- **Proceso o algoritmo**

  Engloba todas aquellas instrucciones encargadas de procesar la información o aquellos datos pendientes de elaborar y que previamente habían sido depositados en memoria principal para su posterior tratamiento. Finalmente, todos los resultados obtenidos en el tratamiento de dicha información son depositados nuevamente en memoria principal, quedando de esta manera disponibles. En definitiva, se puede considerar como una especie de caja negra capaz de albergar unos datos de entrada, realizar unos cálculos previamente definidos y proporcionar unos resultados adecuados. De cara al usuario del programa, cómo se realicen las operaciones de cálculo o se procesen los datos de entrada no es importante, siempre y cuando los resultados obtenidos sean los correctos.

- **Salida de datos o resultados**

  Este bloque está formado por todas aquellas instrucciones que toman los resultados depositados en memoria principal una vez procesados los datos de entrada, enviándolos seguidamente a un dispositivo o periférico externo.

9.1. Elementos auxiliares de programación

Los elementos auxiliares de programación **son variables o conjuntos de variables creadas por el programador con el fin de facilitar la resolución del problema**; no forman parte de los datos originales del problema. Entre los elementos auxiliares de programación podemos considerar los descritos a continuación.

9.1.1. Contadores

Contadores son variables que se incrementan o decrementan en una cantidad constante cuando el flujo de control del programa pasa por una determinada posición; en dicho punto encontramos una sentencia del tipo:

```
variable = variable ± incremento
```

Un contador suele estar asociado a un bucle que determina el número de iteraciones, y por tanto de incrementos, que se tienen que producir para

alcanzar la condición de salida del bucle y continuar; dicho contador no necesariamente es un número de veces, sino un valor determinado a alcanzar por la variable.

9.1.2. Acumuladores

Acumuladores son variables que cambian su valor dependiendo de una función matemática determinada en el programa; esta variable se diferencia de la variable tipo contador en que la variación no es fija, sino que adopta el valor determinado por una ecuación. La sentencia que lo determina es del tipo:

```
variable = variable expresión (operación aritmética)
```

9.1.3. Interruptores

Un interruptor o switch es una variable que solo puede tomar uno entre dos valores posibles: 1/0, si/no, on/off, verdadero/falso...; este tipo de variable permite la toma de decisión en el programa, de forma que cambie de valor cada vez que el flujo del programa pase por un punto determinado, mediante la siguiente sentencia:

```
switch = 1

...

switch = switch * -1
```

9.2. Prueba de programas

El objetivo específico de la fase de pruebas es encontrar la mayor cantidad posible de errores. La prueba ideal de un sistema sería exponerlo a todas las situaciones posibles. Así garantizaríamos su respuesta ante cualquier caso que se le presente en la ejecución real. Esto es imposible desde todos los puntos de vista. Probar un programa es someterle a todas las posible variaciones de los datos de entrada, tanto si son válidos como si no lo son. Probar es buscarle los fallos a un programa. La fase de pruebas absorbe una buena porción de los costes de desarrollo de software.

9.2.1. Prueba de unidades

La prueba de unidades se plantea a pequeña escala, y consiste en ir probando uno a uno los diferentes módulos que constituyen una aplicación.

Los criterios más habituales son los denominados de caja negra y de caja blanca.

Se dice que una prueba es de caja negra cuando prescinde de los detalles del código y se limita a lo que se ve desde el exterior. Intenta descubrir casos y circunstancias en los que el módulo no hace lo que se espera de él.

Por oposición al término "caja negra" se suele denominar "caja blanca" al caso contrario, es decir, cuando lo que se mira con lupa es el código que

está escrito y que se intenta que falle. Quizás sea más propia la denominación de "pruebas de caja transparente".

A) Caja blanca o pruebas estructurales o pruebas de caja transparente

Las pruebas de caja blanca consisten en probar el código fuente que forma la aplicación. Con la cobertura intentamos formalizar el código probándolo todo de principio a fin. Las pruebas de caja blanca nos sirven para aseverarnos de que un programa hace de modo adecuado los diferentes pasos para los cuales ha sido creado, pero no nos aseveran que haga bien lo que deseamos que haga el programa en su globalidad, la finalidad para la cual ha sido creado el programa.

Tenemos varios tipos de coberturas:

— Cobertura de segmentos o cobertura de sentencias.

  Por segmento se entiende una secuencia de sentencias sin puntos de decisión. Como el ordenador está obligado a ejecutarlas una tras otra, es lo mismo decir que se han ejecutado todas las sentencias o todos los segmentos. El número de sentencias de un programa es finito. Se puede diseñar un plan de pruebas que vaya ejercitando más y más sentencias, hasta que hayamos probado todas.

— Cobertura de ramas.

  Se trata de una ampliación de la cobertura de segmentos consistente en recorrer todas las posibles salidas de los puntos de decisión.

— Cobertura de condición/decisión.

  Se trata de nuevo de una ampliación de la cobertura de ramas consistente en probar todos las posibles condiciones aunque estas estén formadas por expresiones complejas, como es el caso de una expresión dentro de un “if” que contenga un “||” (o).

— Cobertura de bucles.

  Se trata de probar las sentencias de control de iteración, que suelen acarrear gran cantidad de errores. Pueden llevar consigo expresiones booleanas complejas.

En la práctica conviene acercarse al 100% de los segmentos del programa, logrando una buena cobertura de ramas, esto supone, juegos para probar entre el 60 y el 80% del código y, dependiendo de la criticidad de la aplicación (sanitarias, centrales nucleares, aplicaciones militares), superar el 90% de segmentos.

La ejecución de pruebas de caja blanca puede llevarse a cabo con un depurador (que permite le ejecución paso a paso o el uso de puntos de interrupción en los lugares que el programador desee). Esta tarea es muy tediosa, pero puede ser automatizada a través de la generación de los diversos valores que toman los distintos elementos del programa como son, por ejemplo, las ventanas de inspección.

B) Caja negra, pruebas de caja opaca, pruebas funcionales, pruebas de entrada/salida o pruebas inducidas por los datos

Un programa puede estar perfecto en todos sus términos, y sin embargo no servir a la función que se pretende. Las pruebas de caja negra se centran en lo que se espera de un módulo, es decir, intentan encontrar casos en los que el módulo no se atiene a su especificación. Por ello se denominan pruebas funcionales, y el probador se limita a suministrarle datos como entrada y estudiar la salida, sin preocuparse de lo que pueda estar haciendo el módulo por dentro.

Las pruebas de caja negra se apoyan en la especificación de requisitos del módulo. De hecho, se habla de "cobertura de especificación" para dar una medida del número de requisitos que se han probado.

El problema con las pruebas de caja negra no suele estar en el número de funciones proporcionadas por el módulo (que siempre es un número muy limitado en diseños razonables); sino en los datos que se le pasan a estas funciones. El conjunto de datos posibles suele ser muy amplio (por ejemplo, un entero).

A la vista de los requisitos de un módulo, se sigue una técnica algebraica conocida como "clases de equivalencia". Esta técnica trata cada parámetro como un modelo algebraico donde unos datos son equivalentes a otros. Si logramos partir un rango excesivamente amplio de posibles valores reales a un conjunto reducido de clases de equivalencia, entonces es suficiente probar un caso de cada clase, pues los demás datos de la misma clase son equivalentes. Una forma de identificar estas clases de equivalencia son aquellas que respondan a:

- Por debajo/en el/por encima del rango especificado para un tipo de dato.
- Por debajo/en el/por encima de un valor concreto.
- Que se encuentre en el conjunto o fuera de él.
- Que sea verdadero o falso.
- Utilización de los mismos criterios para los datos de salida.

Lograr una buena cobertura con pruebas de caja negra es un objetivo deseable, pero no suficiente a todos los efectos. Un programa puede pasar con holgura millones de pruebas y sin embargo tener defectos internos que surgen en el momento más inoportuno.

9.2.2 Pruebas de integración

Las pruebas de integración y de aceptación son pruebas a mayor escala, que puede llegar a dimensiones industriales cuando el número de módulos es muy elevado, o la funcionalidad que se espera del programa es muy compleja.

Las pruebas de integración se centran en probar la coherencia semántica entre los diferentes módulos, tanto de semántica estática (se importan los módulos adecuados, se llama correctamente a los procedimientos proporcionados por cada módulo), como de semántica dinámica (un módulo recibe de otro lo que esperaba). Normalmente estas pruebas se van realizando por etapas, englobando progresivamente más y más módulos en cada prueba.

Las pruebas de integración se pueden empezar en cuanto tenemos unos pocos módulos, aunque no terminarán hasta disponer de la totalidad. En un diseño descendente (top-down) se empieza a probar por los módulos más generales; mientras que en un diseño ascendente se empieza a probar por los módulos de base.

El planteamiento descendente tiene la ventaja de estar siempre pensando en términos de la funcionalidad global. El planteamiento ascendente evita tener que escribirse módulos ficticios, pues vamos construyendo pirámides más y más altas con lo que vamos teniendo.

Las pruebas de integración se llevan a cabo durante la construcción del sistema, involucran a un número creciente de módulos y terminan probando el sistema como conjunto. Estas pruebas se pueden plantear desde un punto de vista estructural o funcional.

Las pruebas estructurales de integración son similares a las pruebas de caja blanca, pero trabajan a un nivel conceptual superior. En lugar de referirnos a sentencias del lenguaje, nos referiremos a llamadas entre módulos. Se trata pues de identificar todos los posibles esquemas de llamadas y ejercitarlos para lograr una buena cobertura de segmentos o de ramas.

Las pruebas funcionales de integración son similares a las pruebas de caja negra. Aquí trataremos de encontrar fallos en la respuesta de un módulo cuando su operación depende de los servicios prestados por otros módulos. Según nos vamos acercando al sistema total, estas pruebas se van basando más y más en la especificación de requisitos del usuario.

Las pruebas finales de integración cubren todo el sistema y pretenden cubrir plenamente la especificación de requisitos del usuario. Además, a estas alturas ya suele estar disponible el manual de usuario, que también se utiliza para realizar pruebas hasta lograr una cobertura aceptable.

En todas estas pruebas funcionales se siguen utilizando las técnicas de partición en clases de equivalencia y análisis de casos límite.

9.2.3. Pruebas de aceptación

Las pruebas de aceptación son las que se plantea el cliente final, que decide qué pruebas va a aplicarle al producto antes de darlo por bueno. El objetivo del que prueba es encontrar los fallos lo antes posible, antes de poner el programa en producción.

Son básicamente pruebas funcionales, sobre el sistema completo, y buscan una cobertura de la especificación de requisitos y del manual del usuario. Estas pruebas no se realizan durante el desarrollo, sino una vez pasadas todas las pruebas de integración por parte del desarrollador.

— Las pruebas alfa consisten en invitar al cliente a que venga al entorno de desarrollo a probar el sistema. Se trabaja en un entorno controlado y el cliente siempre tiene un experto a mano para ayudarle a usar el sistema y para analizar los resultados.

— Las pruebas beta vienen después de las pruebas alfa y se desarrollan en el entorno del cliente, un entorno que está fuera de control. Aquí el cliente se queda a solas con el producto y trata de encontrarle fallos de los que informa al desarrollador.

9.3. Verificación y validación

La verificación y validación tiene por objetivo:

— Detectar y corregir los defectos cuanto antes, disminuir los riesgos y las desviaciones.

— Mejorar la calidad y la fiabilidad de los productos "software".

— Mejorar la visibilidad de la gestión y valorar rápidamente los cambios propuestos.

— Detectar y corregir los defectos en el ciclo de vida del "software".

— Valorar rápidamente los cambios propuestos y sus consecuencias.

No es por ejemplo un objetivo deducir el tamaño y el tiempo de ejecución del software. Verificación y validación no son términos equivalentes que puedan usarse indistintamente cuando se habla del software.

VERIFICACIÓN	productos correctos
VALIDACIÓN	los productos correctos acordes con los requerimientos

El objetivo de la validación del software es la corrección del producto final respecto a las necesidades del usuario. La técnica más tradicional de validación son las pruebas del software.

Las principales técnicas de verificación son las revisiones y auditorias. Entre las actividades de verificación se encuentran: comprobar la adecuación de los requisitos, determinar la adecuación del diseño y aplicar los datos de prueba. Una de las actividades de la verificación seria realizar análisis de valores límites.

El objetivo de las revisiones técnicas es evaluar un producto intermedio para ver que se ajusta a las especificaciones, para comprobar que el desarrollo se está haciendo de acuerdo con los planes y que los cambios en el producto se realizan adecuadamente.

Uno de los objetivos principales de las inspecciones es comprobar si el producto satisface sus especificaciones o los atributos de calidad fijados.

El objetivo de la inspección es detectar y registrar los defectos.

Uno de los objetivos principales del "walktrhough" es la evaluación de un producto para mejorarlo. El objetivo del "walktrough" es la evaluación de un producto para buscar defectos, omisiones y contradicciones, para mejorar el producto, para evaluar conformidad con normas y considerar posibles soluciones y alternativas a los problemas encontrados.

9.4. Otras pruebas

— **Aleatorias:** basadas en que la probabilidad de descubrir un error es similar si se eligen pruebas al azar que si se utilizan criterios de cobertura.

— **Solidez** *(robustness testing)*: probando la reacción del sistema ante datos de entrada erróneos.

— **Aguante** *(stress testing)*: se trata de probar hasta dónde aguanta un programa por razones internas, como puede ser trabajar con una carga de CPU del 90%, un disco con el 90% de espacio ocupado, con memoria ocupada forzando "swapping", etc.

— **Prestaciones** *(performance testing)*: se miden parámetros de consumo en cuanto a tiempo de respuesta, memoria ocupada, espacio en disco...

— **Interoperabilidad** *(interoperability testing)*: se buscan problemas de comunicación entre nuestra aplicación y otras con las que debe trabajar.