MTP e INS

Última modificación 2008/02

MTP e INS

Contenido

MTP (Metodología de la programación) e INS (Ingeniería del Software)

1.- INGENIERÍA DEL SOFTWARE

1.1.- Definiciones

- **Ingeniería del software**: Establecimiento y uso de principios de ingeniería robustos encaminados a obtener un software económico, fiable y eficiente sobre máquinas reales.
- **Software**: Compuesto por algoritmos, estructuras de datos y documentación. Se desarrolla y se deteriora.
- **Métodos**: Describen cómo se desarrolla técnicamente el software.
- **Herramientas**: Dan soporte a los métodos.
- **Procedimientos**: Relacionan formalmente métodos y herramientas.
- **Factores de calidad del software**:
  - Externos:

Corrección	Reusabilidad	Eficiencia
Robustez	Compatibilidad	Verificabilidad
Modificabilidad	Portabilidad	Integridad
Facilidad de uso		

  - Internos:

Modularidad	Legibilidad

1.2.- Ciclos de vida del software

- Fase de análisis: especificación de requerimientos, planificación, evaluación de costes.
- Fase de desarrollo: diseño, implementación, pruebas.
- Fase de mantenimiento: corrección, adaptación, ampliación.

Ciclo de vida clásico

(W. Royce) Secuencial. Cada paso tiene realimentación y proporciona al siguiente entrada y documentación.

1. Especificación de requerimientos.
   - Estudio del sistema
   - Requerimientos de funcionalidad
   - Requerimientos de rendimiento
   - Interfaz con usuario
   - Documentación
2. Diseño
   - Estructuras de datos
   - Arquitectura del sistema
   - Detalle funcional o procedimental
3. Codificación
4. Integración
   - Pruebas
   - Adaptación al entorno

Prototipado

En papel y ordenador. Evolutivo (Throw-away). Vertical y Horizontal.

\+ Facilita el desarrollo y la especificación de requerimientos.

\- Crea una inversión adicional y falsas expectativas en el cliente.

1.3.- Principios de ingeniería en la etapa de diseño

- Abstracción: de procedimientos o funcional de datos
- Ocultación de información

- Modularidad: pocas interfaces, estrechas y explícitas.

2.- DISEÑO ESTRUCTURADO

2.1.- Características

- Usa la definición del problema para guiar la definición de la salida.
- Particionamiento del sistema en módulos.
- Cada módulo una caja negra con

○ Entrada ○ Salida ○ Función	○ Mecánica ○ Datos internos

- Herramientas: Diagramas de estructura o Pseudo código
- Estrategias
- Criterios de evaluación de la calidad: Acoplamiento y cohesión

Diagrama de estructura

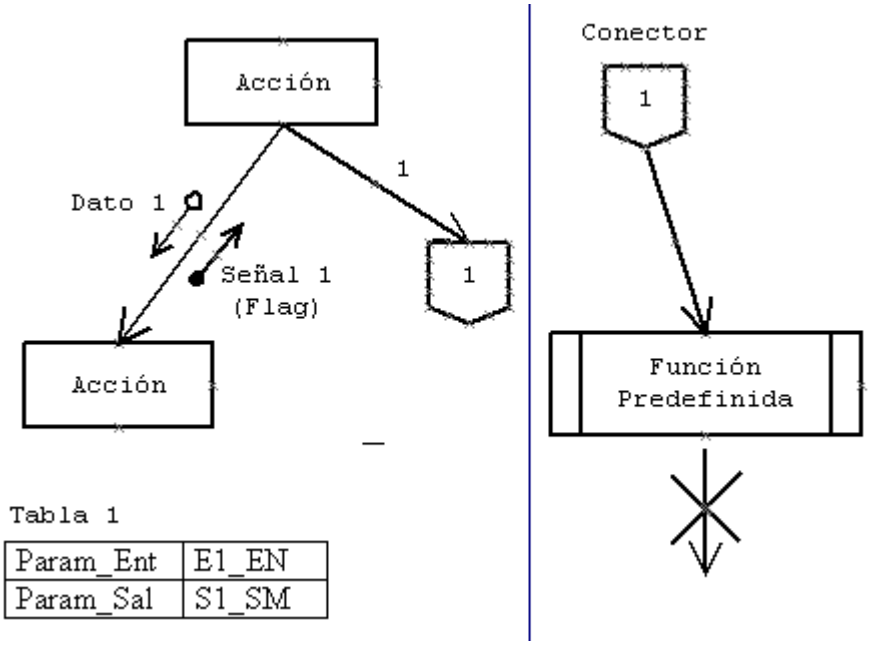


Tabla 1

Param_Ent	E1_EN
Param_Sal	S1_SM

-> Especificación por Interfaz-Función:
- Puede incluirse como comentario en el código
- Da libertad a los programadores
- Para módulos productores

módulo X
usa:
función:
devuelve:

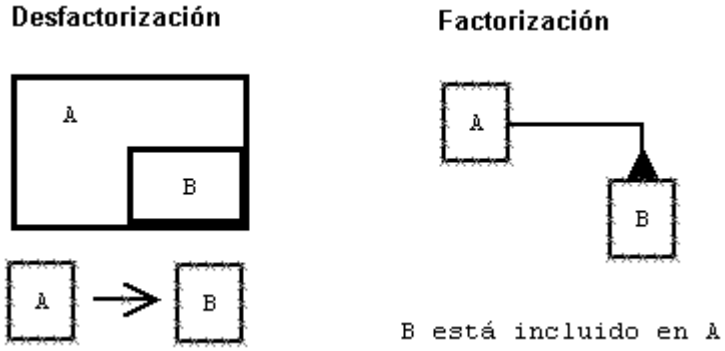
-> Especificación por pseudocódigo:
- Se utiliza en módulos críticos y directores
- Se debe incluir como comentario la especificación Interfaz-Función

Acoplamiento: (Datos vagabundos)
- Normal: El módulo A invoca al módulo B. B termina y devuelve el control a A. Toda la información que comparten es

pasada en la llamada.
Por datos: Todos los datos son elementales
Por estampado: Intercambian datos compuestos
Por control: Algún dato es una señal (flag) de control o descriptiva
- Global: Los módulos usan areas globales de datos.
- Contenido: Un módulo [modifica o lee datos internos | cambia o salta al código interno] del otro módulo

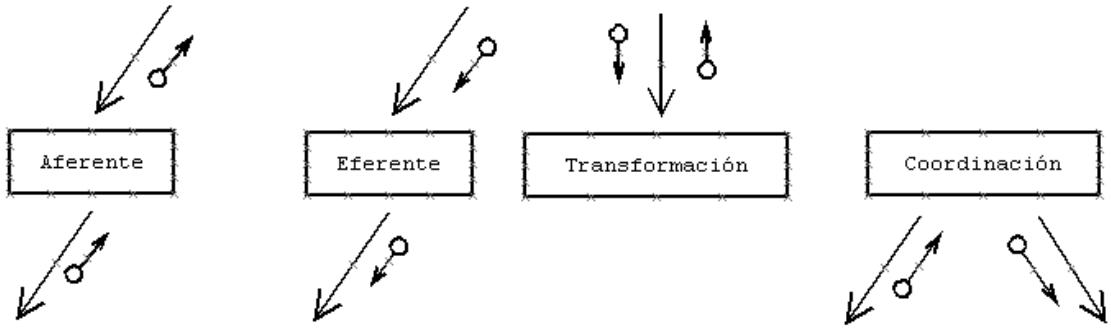
Cohesión:
- Funcional: El módulo ejecuta una sola función
- Datos: Divisibles en funcionales
- Secuencial: Ejecuta varias funciones en serie (la salida de una es la entrada de la siguiente)
- Comunicacional: Ejecuta varias funciones en paralelo
- Control
- Procedural: Funciones en serie no relacionadas más que temporalmente
- Temporal: Se relacionan en el tiempo, pero no importa el orden
- Lógica: De varias funciones relacionadas lógicamente solo realiza una (determinada por una señal -flag-).
- Casual: realiza varias funciones que no tienen relación entre ellas.

Factorización / Desfactorización



Clarificación del sistema
+ Factorización
+ Minimización del código duplicado
+ Separación del trabajo de la dirección
- Disgregación de decisiones (la parte de reconocimiento y la de ejecución en distintos módulos)

Forma del sistema



Sistemas dirigidos por entrada física
La parte aferente no produce una transformación de los datos y los módulos superiores deben tratar con el formato físico de la entrada.
- mantenimiento costoso y acoplamiento alto
- portabilidad baja

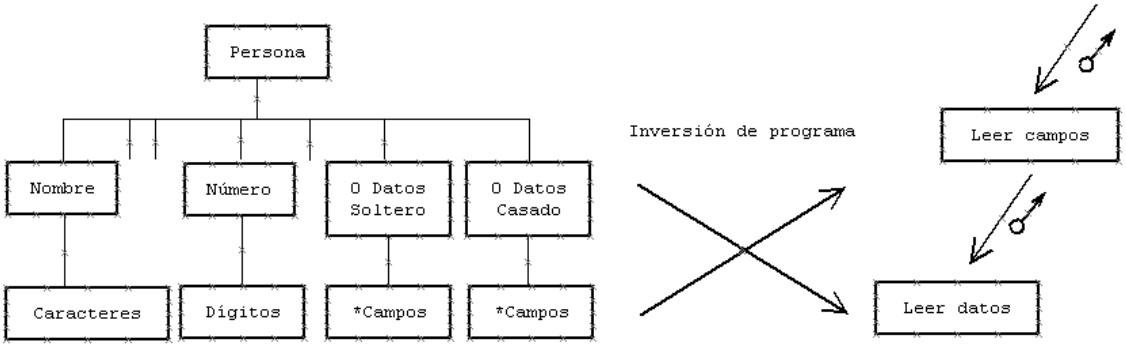
Sistemas balanceados
En la parte aferente habrá módulos que depuren los datos (≡ eferente). Los módulos superiores manejan una estructura lógica de los datos.
+ mantenimiento y acoplamiento bajos
+ portabilidad alta

Información de los errores
- El módulo que detecta el error debe informar
- Cada módulo tiene su mensaje de error o
- Existe un único módulo de error:
    + Evitamos duplicación de mensajes
    + Formato único, consistente
    + Mantenimiento más fácil
    + Reutilizable
    - Peor legibilidad del código
    - Dato "artificial" (nº de error)

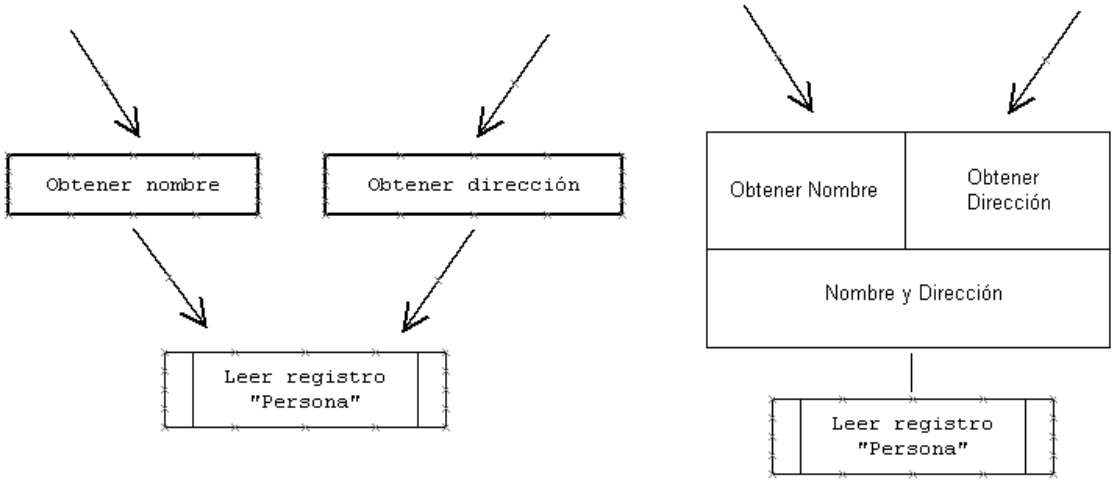
Memoria previa: Datos estáticos

2.2.- Estructuras de datos

Choque de estructuras (no compatibles)



Cúmulos de información



- Evita acoplamiento por estampado de datos
- Evita acoplamiento global
- Mejora accesos a BD

Módulos de iniciación y terminación
Lo más tarde | Lo más pronto posible para evitar flujos de datos vagabundos.

Abanicos de entrada y salida
- Cohesión alta (cada interfaz tiene el mismo nº y tipo de parámetros)
- Ha de ser de 7±2 excepto los centros de transacciones o estructuras de datos grandes.

3.- EL ANÁLISIS ESTRUCTURADO

3.1.- Un poco de historia

La ingeniería del software, según la definió Bauer en 1969 es "el establecimiento y uso de principios de ingeniería robustos orientados a obtener software económico que sea fiable y funcione de manera eficiente sobre máquinas reales".

El ciclo de vida clásico del software apareció en los años 70 como precursor del resto de paradigmas que se han propuesto posteriormente, como el ciclo de vida con prototipación o el ciclo de vida en espiral.
Este ciclo de vida considera la producción del software como un proceso secuencial dividido en 5 etapas: análisis, diseño, codificación, pruebas y mantenimiento. Estas etapas se pueden abordar utilizando técnicas estructuradas.

En la etapa de análisis se emplea el análisis estructurado, una metodología que permite describir un sistema como una red de procesos que transforman datos. Este proceso se utiliza como entrada para la siguiente fase, el diseño, que a su vez produce una salida (descomposición modular del sistema) que se utilizará como documentación para que los programadores puedan realizar la codificación del sistema.

El propósito del análisis de requisitos, primer paso técnico del proceso de ingeniería del software, es obtener una descripción lógica del sistema. Los desafíos principales son: para el analista, entender con precisión lo que el usuario quiere y para el usuario, entender con precisión lo que el producto software le ofrece. Así la clave de éxito en esta etapa es lograr establecer una buena comunicación entre el usuario y el analista.

El análisis estructurado es un método para realizar el análisis de requisitos popularizado por DeMarco en los 70 y con variaciones realizadas por Yourdon, Gane y Sarson en la misma década, por Ward y Mellor para aplicaciones en tiempo real y por Hatley y Pirbhai a final de los 80.

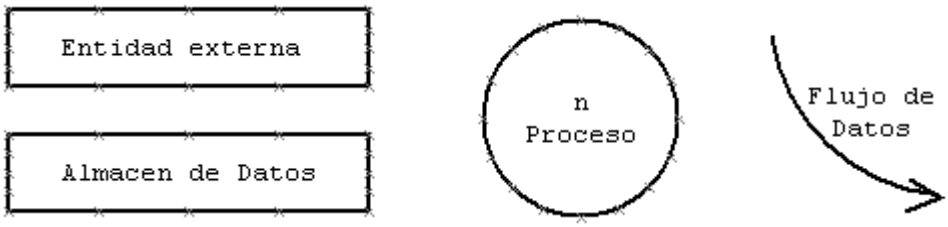
3.2.- El análisis estructurado

Las herramientas utilizadas en el análisis estructurado son:

- Diagramas de Flujo de Datos (DFD)
- Diccionario de Datos (DD)
- Especificación de Procesos (EP)
- Diagrama de Transición de Estados (DTE)
- Diagramas Entidad-Relación (DE-R)

La suma de los tres primeros (DFD, DD y EP) se conoce como diagrama de estructura.

El Diagrama de Flujo de Datos (DFD)



Es una herramienta gráfica que representa el flujo de la información y las transformaciones que se aplican a los datos al moverse desde la entrada a la salida. Sus características son:

- Es comprensible por los usuarios
- Especifica QUÉ hace el sistema.
- Se puede aplicar en diferentes niveles de detalle.
- Muestra el flujo de la información.

La explosión de un proceso consiste en desagregar un proceso padre en un nuevo DFD de mayor detalle. Al realizar la explosión, hay que tener en cuenta las siguientes normas:

- El proceso "n" da lugar a los hijos "n.1", "n.2", "n.m"
- En el nuevo DFD pueden aparecer almacenes de datos privados
- Todos los flujos que entraban o salían del proceso de datos deben entrar y salir del conjunto de procesos hijos, y viceversa (balanceo de datos).
- La única excepción corresponde al tratamiento de errores.

La cantidad de niveles depende de la complejidad del sistema. En todo caso, los primeros niveles son:

- Diagrama de contexto (Nivel 0)
  Es un resumen genérico del sistema. Contiene un único proceso y las entidades externas.
- DFD 0 (Nivel 1)
  Modelo con toda la funcionalidad del sistema.

El Diccionario de Datos (DD)

Contiene la descripción detallada de cada dato del sistema. Para cada flujo o almacén de datos que aparezca en el DFD del sistema, el DD tendrá una entrada que lo describa. En el DD se especificará cada estructura de datos hasta el nivel más elemental.

Cada dato deberá tener una definición que incluya:

- Comentario
- Composición (si es un dato no elemental)
- Valores posibles (si es un dato elemental)

La notación utilizada para describir los datos es la siguiente:

= está compuesto de

+ concatenación de datos
() dato opcional
{} repetición
[] selección de una de las alternativas
| separador de alternativas
** comentario
@ campo clave para un almacén de datos

@(Campo clave)*Nombre* = *dato* + (*dato opcional*) + {*repetición carácter permitido*} + [*opción 1* | *opción 2*] * Comentario

La Especificación de Procesos (EP)

Es un conjunto de descriptores de la lógica interna de los procesos de último nivel en los DFDs. Éstos definen qué debe hacerse para transformar las entradas en salidas.
Para especificar un proceso pueden utilizarse herramientas tales como un lenguaje estructurado (LDP o pseudocódigo), diagramas de flujo, esquemas interfaz-función, diagramas de Nassi-Schneiderman, árboles de decisión o tablas de decisión.

- Lenguaje estructurado: Nombre proceso + Número + Definición + Entrada + Salida + Resumen de la lógica
- Árbol de decisión: Nº de acciones pequeño
- Tabla de decisiones: Nº de acciones grande

El Diagrama de Transformación de Estados (DTE)

Un DTE representa el comportamiento de un sistema mostrando los estados y sucesos que hacen que el sistema cambie de estado. Además el DTE indica qué acciones se llevan a cabo como consecuencia de un suceso determinado.

Los Diagramas Entidad-relación (DE-R)

Cuando la información que fluye entre los procesos es relativamente simple la notación básica de flujos y almacenes de datos es suficiente. Sin embargo, generalmente es necesario representar relaciones entre complejas colecciones de datos. Así se propone realizar la modelización de los datos. Para ello la herramienta más popular es el DE-R que permite describir los grupos de datos y sus relaciones. Un DE-R puede ser fácilmente traducido a un modelo relacional e implementado usando un Sistema Gestor de Bases de Datos.
El modelo E-R original propuesto por Chen ha sido extendido por Teorey, Fry y Yang incorporando aspectos de herencia y agregación. Este modelo representa la información mediante entidades, atributos y relaciones.

La relación entre el modelo E-R y los DFDs

Toda modelización de un sistema de información puede ser abordada desde dos perspectivas no excluyentes entre sí: la modelización de los datos y la modelización de los procesos. Los DFDs son utilizados para modelar jerárquicamente los procedimientos del sistema, en cambio el modelo E-R se utiliza principalmente para modelizar preferentemente los datos. Las metodologías Orientadas a Objetos (OO) destacan por modelar ambos aspectos conjuntamente.
En todo caso el uso preferente de una modelización no hace innecesaria el uso de la otra ya que la modelización simultánea de los procesos y los datos permite realizar una verificación muy importante antes de finalizar el Análisis de Requisitos:

a) toda la información contenida en los almacenes de datos del modelo de procesos debe estar contenida en el modelo de datos.
b) toda la información contenida en el modelo de datos debe ser procesada por algún proceso en el modelo de procesos.

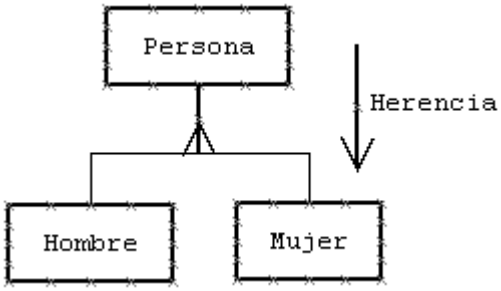
4.- LA PROGRAMACION ORIENTADA A OBJETOS (OO)

Nota: Los ejemplos de código están realizados en Object Pascal.

4.1.- Conceptos básicos

Clase: Define un tipo de objeto que tiene atributos -que definen el estado- y métodos -que definen el comportamiento-. Cada instancia de clase es un objeto.

Herencia y Polimorfismo



- La herencia es el mecanismo de reutilización de código
- El polimorfismo es la capacidad de adoptar distintas formas durante la ejecución

- A partir de la clase "Persona", por herencia, generamos las clases "Hombre" y "Mujer", que pueden añadir y/o modificar atributos de la clase "Persona".
- Puedo definir una variable "Persona" sin saber si será "Hombre" o "Mujer" y más tarde adaptar su uso mediante polimorfismo universal:

if (ActiveMDIChild IS TFHijoTexto) then (ActiveMDIChild AS TFHijotexto).Text:=…

- La operación "+" sirve para sumar tipos numéricos iguales, pero mediante polimorfismo ad-hoc puede usarse para concatenar cadenas (sobrecarga) o para sumar tipos distintos (coerción).

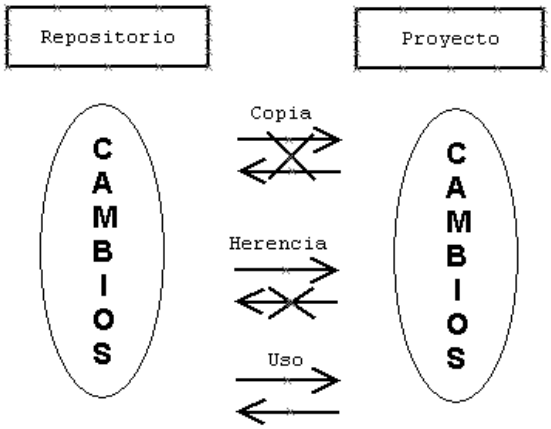
Tipo de enlace

- Estático: El tipo se asocia con los identificadores explícitamente.
- Dinámico: El tipo se asocia en tiempo de ejecución con los valores de las variables.

Entornos de programación visuales y OO

Herramientas RAD: Sistema para el desarrollo rápido de aplicaciones software -tanto OO como no OO-.

Delphi:



- Entorno visual (≠ programación visual)
- Programación O.O. (Lenguaje Object Pascal)
- Múltiples capacidades: BD, Cliente/Servidor, Componentes prefabricados, Internet, Servidores de automatización.

Aplicaciones SDI/MDI
SDI -> FormStyle:= fsNormal
MDI -> FormStyle := fsMDIForm (padre único -formulario "Main")
- Propiedad ActiveMDIChild
- Propiedades MDIChildren y MDIChildCount

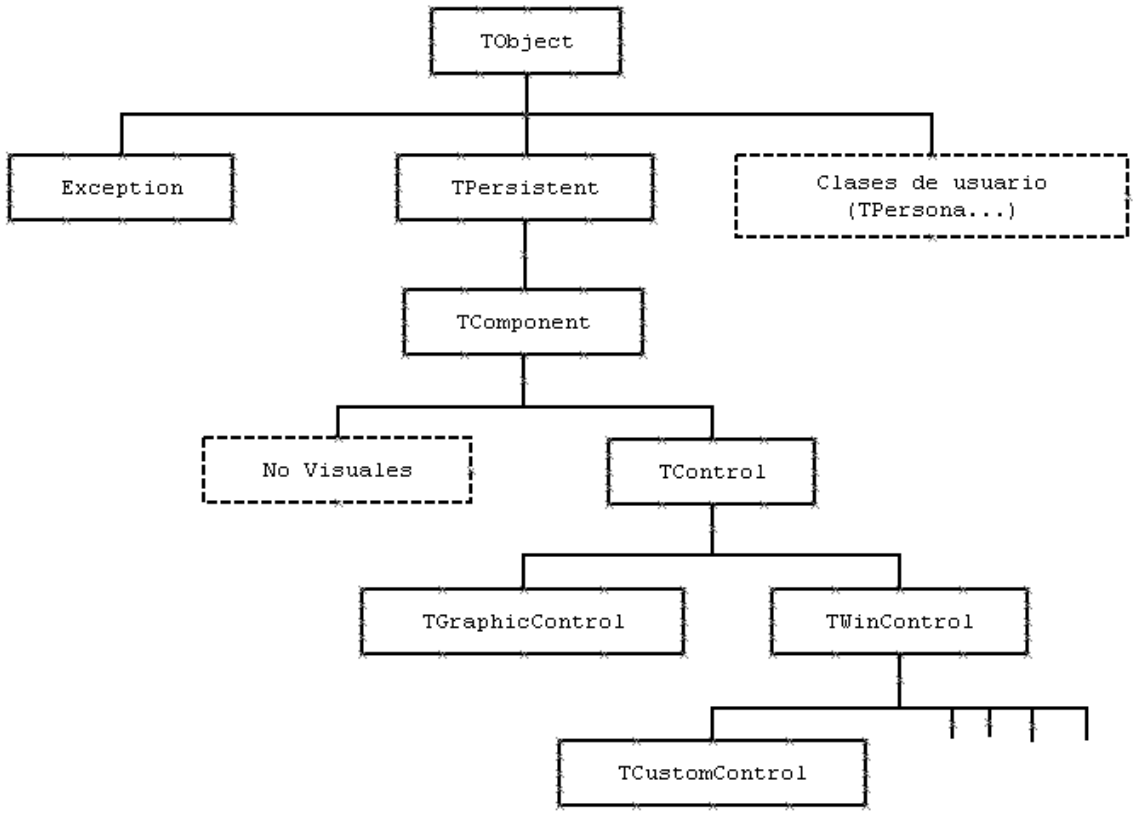
FormStyle := fsMDIChild (n hijos -de tipos diversos)

Parent vs Owner
NuevoForm := TForm1.Create(Owner);
NuevoForm.Parent(Self);
Un botón puede tener como *owner* el formulario y como *parent* un panel del mismo.
Hasta que no se asigna un *parent* no se dibuja el control.

4.2.- VCL (Visual Component Lybrary)

- Componentes No visuales
  - De sistema
  - Menús
  - Diálogos
    - Componentes prediseñados (función *Execute*)
    - Funciones de Biblioteca (*Unit Dialogs*)
    - P.e.: *ShowMessage*, *MessageDlg*, *InputBox*, *InputQuery*

- Todos los componentes pueden ser "*Owner*" (Propiedad "*Components*")
- Solo *TWinControl* e hijos pueden ser "*Parent*".
- *TWinControl*
  - Reciben foco, pueden ser "*Parent*"
  - Tienen manejador de ventana (*Handle*, *WindowHandle*)
  - P.e.: *TButton*, *TEdit*, *TPanel*.
- *TGraphicControl*
  - No reciben foco, no pueden ser "*Parent*"
  - Propiedad "*Canvas*" / Método "*Paint*"
  - P.e.: *TLabel*, *TImage*, *TBevel*



Creación de componentes

Un componente es una especificación de interfaz.
Es un tipo descendiente de:

- *TComponent* (si es No visual)
- *TGraphicControl* (si es un control gráfico)
- *TCustomControl* (si es un control visual original)
- Algún hijo de *TWinControl* (si es un control visual derivado)

La creación se hace de modo no-visual:
1.- Creación de un módulo (*Unit*)
2.- Derivar el tipo de uno existente
3.- Añadir/Modificar propiedades, métodos y/o eventos
4.- Registrar el componente
5.- Creación de la ayuda del nuevo componente
6.- Creación de un paquete para instalar

Propiedades vs Atributos

Una propiedad encapsula un atributo y aparece en tiempo de diseño (en el inspector de objetos) si la publicamos.
- Podemos publicar propiedades de solo lectura
- Podemos usar métodos para su lectura/escritura y en ellos generar eventos (por ejemplo).

4.3.- Excepciones

Una excepción es un objeto. La gestión de excepciones debe:
1.- Localizar bloques de código que pueden generar excepciones

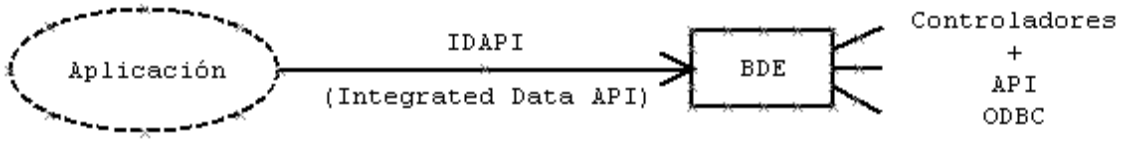
2.- Aislar el bloque localizado
3.- Definir respuestas para
- Tratar excepciones (try...except...end;)
- Proteger recursos (try..finally...end;)
- Personalizar el gestor de excepciones
- Los bloques de gestión de excepciones se pueden anidar

- Generar una excepción:
raise Excepcion.Create('Mensaje excepción');
raise EAbort.Create('Mensaje ignorado'); //≡Abort

- Propagar una excepción:
```
try ...
except ...
        raise; // No destruir el objeto excepción. Se eleva al bloque siguiente.
end;
```

```
try
  try
    ...
   finally
     ... // No destruye el objeto excepción
  end;
except
  ... // destruye el objeto excepción (excepto si indicamos 'raise')
end;
```

4.4.- BDE (Borland Database Engine)
- Capa entre la aplicación y los datos de la BD
- Se ocupa de entenderse con los SGBD de modo homogéneo (API consistente)
- Proporciona características no soportadas por los SGBD
- Puede ofrecer servicios compartidos entre varios SGBD



4.5.- Modelos Cliente / Servidor
Costes de hardware, software y personal.

CLIENTE				SERVIDOR
(Front-end)				(Back-end)
envía	-	mensajes (peticiones)	-	recibe
recibe	-	respuestas	-	envía

Modelos de comunicación
- Mensajes (colas/buzones)
        Asíncrono
- RPC

Modelos tecnológicos
- BD-SQL
- Monitores TP (*Transaction process*)
- *Groupware* (Datos estructurados)

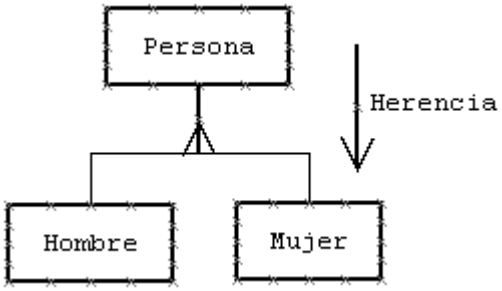
Síncrono - Objetos distribuidos

Clasificación de los sistemas según la carga en cliente y servidor de: Gestión de datos, Funciones de aplicación, Presentación.

Niveles en la aplicación cliente:
1.- IGU (Interfaz gráfica de usuario)
2.- Interacción con usuarios: Sucesos, Funciones específicas
3.- Lógica de la aplicación.

4.6.- Componentes distribuidos (COM)

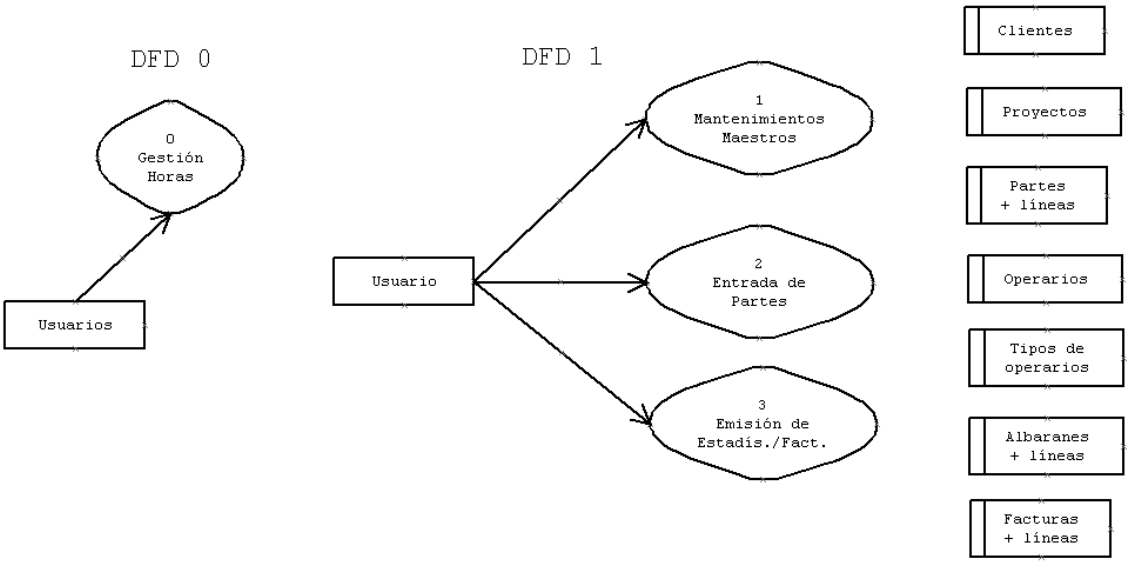
Localizables por su URL: protocolo://servidor[:puerto][/ruta][/consulta]
Clases como implementaciones de un conjunto de interfaces.
En el sistema clásico O.O.:

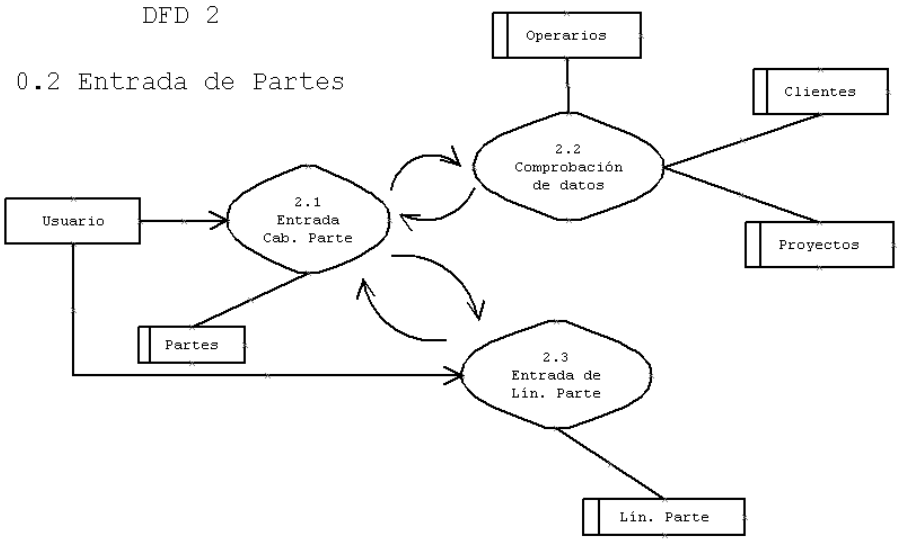


- Si tengo una instancia de "Persona" no puedo usarla como "Hombre", aunque puedo usar una instancia de "Hombre" a través de una variable tipo "TPersona" (polimorfismo universal). Sin embargo esa instancia de "Hombre" nunca podrá ser "Mujer".

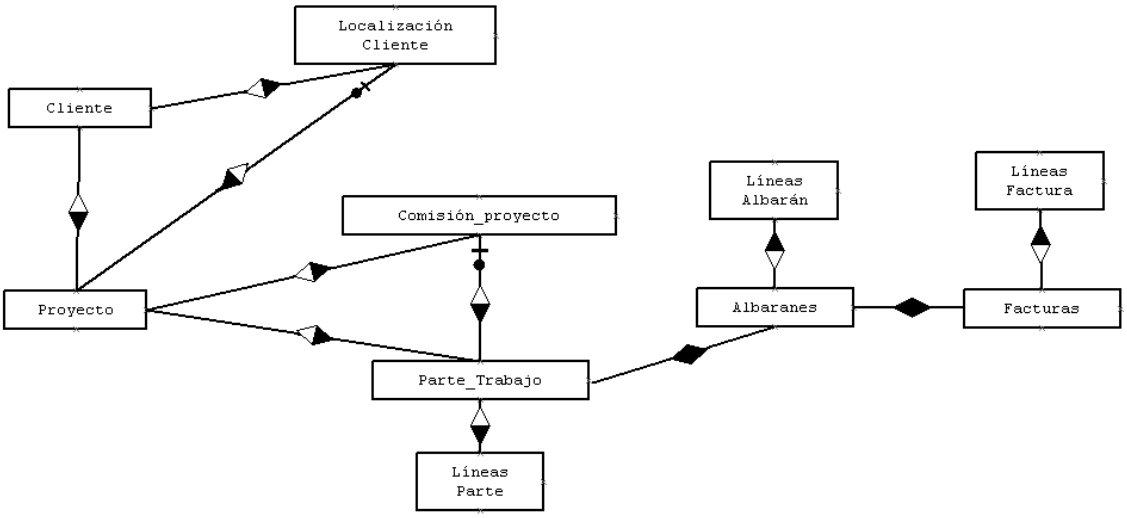
- En el sistema COM la idea es que "TPersona" tenga varias interfaces ("Futbolista", "Estudiante"...) de modo que pueda usar una instancia con cualquiera de ellas.

ANEXO I - Ejemplo de DFD





ANEXO II - Ejemplo de DE-R



ANEXO III - Algunos componentes importantes en Borland Delphi

TForm - Algunos eventos importantes por orden en que se generan:

OnCreate OnShow (se crean los controles y componentes) OnResize OnActivate OnPaint	OnCloseQuery OnClose OnDeactivate OnHide OnDestroy

	Propiedades	Eventos	Métodos
TApplication	Active CurrentHelpFile Title Icon	OnException OnHint OnIdle	Minimize Maximize
TScreen	Height Width Cursor	OnActiveFormChange	

TSession: Para acceder al BDE y configurarlo. Gestiona alias (Temporales y Permanentes)

TDataBase

Algunos eventos importantes por orden en que se generan:

- BeforeConnect
- OnLogin
- AfterConnect
- BeforeDisconnect
- AfterDisconnect

- Crea un alias temporal o usa uno existente
- Gestiona conexiones
- Gestiona claves de usuario
- Gestiona transacciones - (**A**tomic, **C**onsistent, **I**solated, **D**urable)
    1. Estado de la transacción.
       StartTransaction ---------- Commit
       |---------- Rollback
       (InTransaction)
    2. Control de concurrencia
        - Pesimista (Local) - Bloquea el registro en uso
        - Optimista (Update Mode). De mayor a menor restricción:
            - UpWhereAll
            - UpWhereChanged
            - UpWhereKeyOnly
    3. Aislamiento (TransIsolation). De mayor a menor aislamiento:
        - tiRepeatableRead
        - tiReadCommitted
        - tiDirtyRead

TDataSet

- Cached Updates (-> TransIsolation == tiDirtyRead)
- UpdatesPending
- ApplyUpdates
- CommitUpdates

```
try
  ApplyUpdates;
  BD.Commit;
  except
    BD.Rollback;
end;
CommitUpdates;
```

ANEXO IV - Componente BeepButton que puede hacer "Beep" al pulsarlo (click)

```
type
      TBeepButton = class(TButton)  // Derivamos de TButton
      private
            FBeepEnabled: Boolean; // Determina si debe sonar el beep
      protected
      public
constructor Create(Aowner:TComponent); override; // Sobreescribimos
                                    // el constructor
            procedure Click; override;  // Sobreescribimos el método click
      published
            property BeepEnabled: Boolean read FBeepEnabled write FBeepEnabled;
end;

procedure Register;

implementation
constructor TBeepButton.Create(Aowner:TComponent);
begin
      inherited Create(Aowner); // Heredamos el método Create de TButton
      FBeepEnabled := True;
end;

procedure TBeepButton.Click;
begin
      if FBeepEnabled then Beep; // Hacemos beep si procede
      inherited Click; // Heredamos el método click de TButton
end;

procedure Register;
begin
      RegisterComponents('Ejemplos',[TBeepButton]);
end;

end.
```

ANEXO V - Componente AlarmTime: Alarma-despertador

```
type
      AlarmTime = class(TComponent);
      private
            FTimer: TTimer; // Utilizo dentro de mi componente, un componente
TTimer
            FOnAlarm: TNotifyEvent;
            FAlarmEnabled: Boolean;
            FAlarmTime: TDateTime;
            procedure SetAlarmTime(Alarm:TDateTime); // En el original generamos
                                                     //  OnChangeAlarmTime
      protected
            procedure SelfTimer(Sender: TObject);
      public
constructor Create(Aowner:TComponent); override; // Sobreescribimos
                                    // el constructor
            destructor Destroy; override; // Sobreescribimos el destructor
            property AlarmTime:TDateTime read TAlarmTime write FAlarmEnabled;
      published
            property AlarmEnabled:Boolean read FAlarmEnabled write
FAlarmEnabled;
            property OnAlarm:TNotifyEvent read FOnAlarm write FOnAlarm;
end;


procedure Register;

implementation
constructor TAlarmTime.Create(Aowner:TComponent);
begin
      inherited Create(Aowner); // Heredamos el método Create de TComponent
      FTimer := TTimer.Create(Self); // Creamos el TTimer
      FTimer.Enabled := True;
      FTimer.OnTimer := SelfTimer;
      FTimer.Interval := 1000;

      FOnAlarm := nil; // El programador asignará un método como respuesta al
evento
      FAlarmEnabled := False;
end;

(…)

procedure TAlarmTime.SelfTimer(Sender:Tobject);
begin
      // Si es la hora, la alarma está activa y se ha asignado un método a la
alarma...
      if (Time = FAlarmTime) AND (FAlarmEnabled) AND (Assigned(FOnAlarm))
then FOnAlarm(Self); // Ejecutamos el método asociado al evento.
end;

(…)
```