

## TEMA 1

### **Orientación a objetos, una técnica para mejorar la calidad del software**

Facultad de Informática  
Universidad de Murcia

## **Índice**

1.- Calidad del software

2.- Modelo de objetos

- Abstracción
- Encapsulación
- Modularidad
- Herencia
- Polimorfismo

3.- Reutilización

4.- Diseño Estructurado vs. Diseño OO

5.- Tipos abstractos de datos

# 1.- Calidad del Software

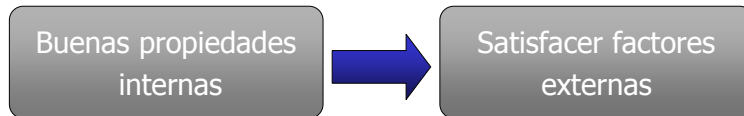
- **Factores Externos**

Pueden ser detectados por los usuarios  
Calidad externa es la que realmente preocupa

- **Factores Internos**

Sólo los perciben los diseñadores e implementadores  
Medio de conseguir la calidad externa

## OBJETIVO



La POO es un conjunto de técnicas para obtener **calidad interna** como medio para obtener **calidad externa** (Reutilización y Extensibilidad)

3

# Factores de calidad del Software

- **Factores Externos**

- |                         |                    |                             |
|-------------------------|--------------------|-----------------------------|
| - Corrección            | - Eficiencia       | - Economía                  |
| - Robustez              | - Portabilidad     | - Integridad                |
| - <b>Extensibilidad</b> | - Facilidad de uso | - Facilidad de reparación   |
| - <b>Reutilización</b>  | - Funcionalidad    | - Facilidad de verificación |
| - Compatibilidad        | - Oportunidad      |                             |

- **Factores Internos**

- Modularidad
- Legibilidad

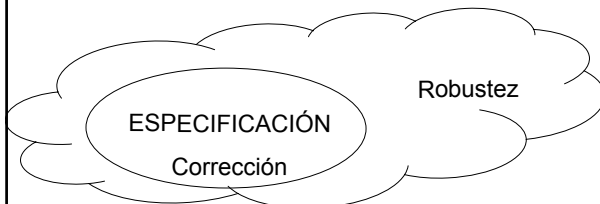
## Corrección

Es la capacidad de los productos software de realizar con exactitud su tarea, tal y como es definida en la especificación.

- Definir los requisitos de manera precisa

## Robustez

Es la capacidad de los productos software de reaccionar adecuadamente ante situaciones excepcionales



Tienen que ver con el comportamiento (casos previstos o no)  
5

## Extensibilidad

Es la facilidad de adaptación de los productos software a los cambios en la especificación.

- Cambios son frecuentes puesto que en la base de todo software hay algún fenómeno humano.
- Dificultad de adaptación proporcional al tamaño del sistema.
- Principios esenciales para facilitar la extensibilidad
  - **Simplicidad** de la arquitectura del software
  - **Descentralización**: módulos autónomos

## Reutilización

Es la capacidad de un producto software de ser utilizado en la construcción de diferentes aplicaciones

- No reinventar soluciones para problemas ya resueltos.
- Se escribe menos software, luego se puede dedicar mas tiempo a mejorar otros factores (fiabilidad)

## Compatibilidad

Es la facilidad de combinar unos elementos software con otros

- Los sistemas necesitan interactuar con otros
- Convenciones estándar de comunicación inter módulos

## Eficiencia

Es la capacidad de un sistema software de requerir la menor cantidad posible de recursos hardware.

- Factor importante para la utilización
- Algunos están obsesionados con micro optimizaciones
- Debemos conjugar eficiencia con los otros objetivos
- Los mecanismos OO deben ser implementados de un modo eficiente tanto en tiempo como en espacio

## Portabilidad

Es la facilidad de transferir productos software a diferentes plataformas (entornos hw y sw).

## Facilidad de uso

Es la facilidad con la que personas con diferentes niveles de experiencia pueden aprender a usar los productos software y aplicarlos a resolver problemas. También incluye la facilidad de instalación, operación y supervisión.

## Funcionalidad

Conjunto de posibilidades ofrecido por un sistema

- Evitar añadir propiedades de forma incontrolada
- Buen producto software debe estar basado en un pequeño número de grandes ideas
- Mantener constante el nivel de calidad

## Oportunidad

Es la capacidad de un sistema software de ser lanzado cuando los usuarios lo desean, o antes.

### Otros factores

- **Economía:**  
completarse con el presupuesto asignado
- **Integridad:**  
proteger contra modificaciones y accesos no autorizados
- **Facilidad para reparaciones** (de defectos)
- **Facilidades de verificación:**  
datos de prueba y procedimientos para detectar fallos

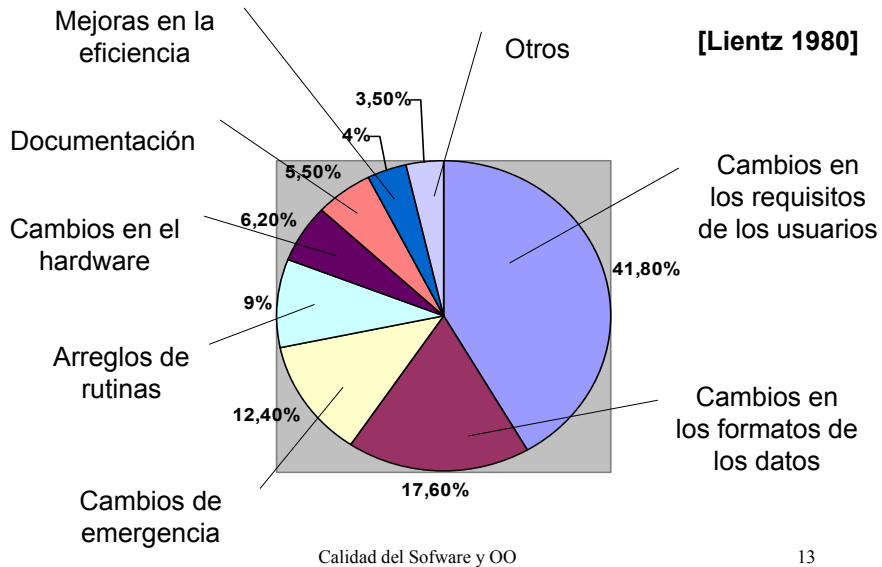
## Consecuencia de estos criterios:

- Necesidad de una **BUENA DOCUMENTACIÓN**:
  - externa (usuarios)  $\Rightarrow$  facilidad de uso
  - interna (desarrolladores)  $\Rightarrow$  extensibilidad
  - interfaz del módulo  $\Rightarrow$  extensibilidad y reutilización
- Factores pueden entrar en **CONFLICTO**:
  - integridad  $\Leftrightarrow$  facilidad de uso
  - economía  $\Leftrightarrow$  funcionalidad
  - eficiencia  $\Leftrightarrow$  portabilidad
  - ajustarse a la especificación  $\Leftrightarrow$  reutilización

## Mantenimiento del software

- No figura como factor *facilidad de mantenimiento*
- Mantenimiento es lo que sucede después de que se ha distribuido un producto de software.
- Se le dedica el 70 % del coste del software
- ¿Qué significa “mantenimiento” en software?
  - Parte noble:           MODIFICACIÓN  
                                  adaptación a los cambios
  - Parte no noble:       DEPURACIÓN  
                                  quitar errores

# Costes de mantenimiento del software



13

## Conclusiones del estudio

- 41'8% extensiones y modificaciones usuario → **Ausencia de extensibilidad**
- 17'6% cambio de los datos → **Estructura física de los datos dispersa por muchas partes del sistema**
- 5'5% Documentación → No se hace documentación a posteriori
- 4% Mejoras en la eficiencia → Cuando el sistema funciona no se buscan mejoras en eficiencia.

## 2.- Modelo de Objetos

### 1. Abstracción:

“Supresión intencionada, u ocultamiento, de algunos detalles de un proceso o artefacto, con el objeto de destacar de manera más clara otros aspectos, detalles o estructuras” [Budd'02]



[Booch'96]

La abstracción se centra en las características esenciales de algún objeto, en relación a la perspectiva del observador.

15

### – Métodos de abstracción:

- **Abstracción por parametrización:** nos permite abstraernos de la identidad de los datos sustituyéndolos por parámetros.
- **Abstracción por especificación:** nos permite abstraernos de los detalles de implementación y fijarnos sólo en el comportamiento en el que pueden confiar los usuarios.



## – Tipos de abstracciones:

### • Abstracción procedural:

- Permite introducir nuevas operaciones.
- Permite abstraer una única acción o tarea.

### • Abstracción de datos:

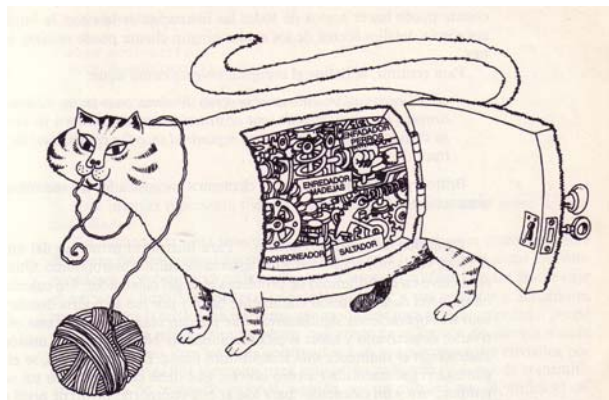
- Permite introducir nuevos tipos de datos
- Se consigue la abstracción por especificación haciendo las operaciones parte del tipo que definen el comportamiento.

### • Abstracción de iteración:

- Permite iterar sobre los elementos de una colección sin revelar la manera en la que se obtiene cada elemento.

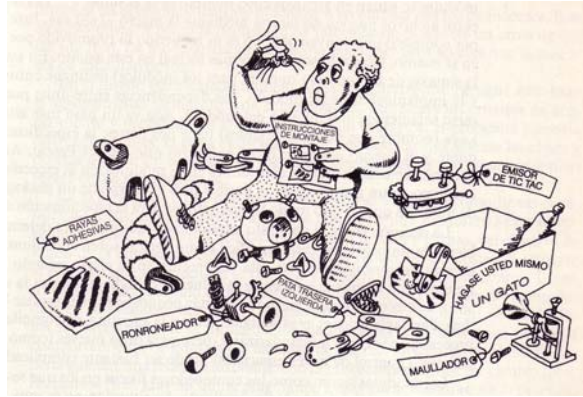
## 2. Encapsulación:

“Proceso de almacenar en un mismo compartimento los elementos de una abstracción que constituyen su estructura y su comportamiento” [Booch'96]



### 3. Modularidad:

“Propiedad que tiene un sistema que ha sido descompuesto en un conjunto de **módulos cohesivos** y **débilmente acoplados**” [Booch’96]



Calidad del Software y OO

[Booch’96]

19

- **Alta cohesión:**

- Un módulo con responsabilidades altamente relacionadas y que no hace una gran cantidad de trabajo

- **Bajo acoplamiento:**

- Un módulo que no depende de *demasiados* otros módulos.
- Favorece:
  - **Comprensión modular:** es posible entender un módulo sin conocer los otros.
  - **Continuidad modular:** un cambio en la especificación afecta sólo a un módulo o a unos pocos.
  - **Protección modular:** el efecto de una situación anormal producida en un módulo afecta sólo a éste o a unos pocos.
- Los módulos se comunican mediante *interfaces estrechas y bien definidas*.

Calidad del Software y OO

20

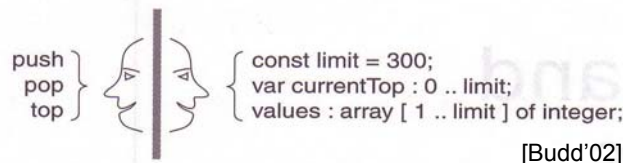
## Principios de diseño modular

### • Ocultación de información

*“El diseñador de cada módulo debe seleccionar un subconjunto de propiedades de un módulo como información oficial para ponerla a disposición de los autores de módulos clientes”*

– Una abstracción de datos puede verse como que tiene dos caras:

- **Interfaz:** operaciones que definen el comportamiento (cliente)
- **Implementación** (programador)



21

### • Auto-documentación

*“El diseño de un módulo debería esforzarse para lograr que toda la información relativa al módulo forme parte del propio módulo”*

### • Acceso uniforme

*“Todos los servicios ofrecidos por un módulo deben estar disponibles mediante una notación uniforme, que no considere si se han implementado mediante almacenamiento o cálculo”*

Sea **c** una variable representando una cuenta bancaria y **saldo** una propiedad aplicable a **c**,

`c.saldo` → saldo es un **campo** de un registro

`saldo(c)` → saldo es una **función**

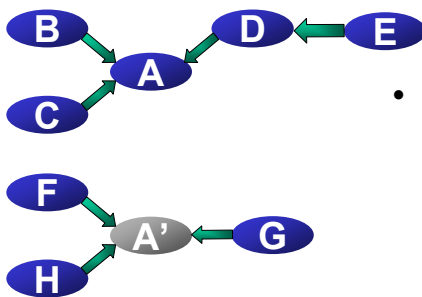
Necesitamos constructores sintácticos que nos permitan expresar de la misma manera el acceso a una función y a un atributo.

## • Principio Abierto-Cerrado

*“Los módulos deberían ser a la vez **abiertos** y **cerrados**”*

- Un módulo está **abierto** si está disponible para ampliarlo
- Un módulo está **cerrado** si está disponible para su uso
- Los dos objetivos son **incompatibles** con las técnicas tradicionales:
  - o está abierto → no se puede utilizar todavía
  - o se cierra → cualquier cambio provoca cambio en cadena

## Principio Abierto-Cerrado

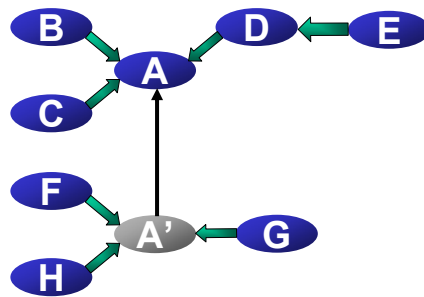


- Dos soluciones:
  - **Adaptar el módulo A**  
(cambios en cadena en los clientes)
  - **Crear una copia de A**  
(explosión de variantes de módulos)

¿Es posible adaptar A sin afectar a los clientes?

¿Cómo se pueden obtener módulos que sean a la vez abiertos y cerrados?

## Principio abierto-cerrado



### Solución: HERENCIA

A' contiene solamente las diferencias :

- nuevas características
- redefiniciones para los clientes

## • Elección única. Ejemplo

Módulo A:

```
TYPE Publicacion
    autor, titulo: String
    año: Integer;
    CASE tipoPubli (libro, revista, articulo) of
        libro: (editorial: String);
        revista: (editor: String; periodicidad: Periodo);
        articulo: (revista, volumen, numero: String)
    END;
```

Sea B un cliente de A, tendría *p:Publicación*

case p of

libro: ... *acceso a los campos de libro*

¿Si añadimos un nuevo tipo de publicación?

Cualquier cliente de A tendría que actualizar su estructura

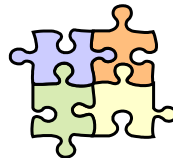
- **Elección única**

*“Siempre que un sistema software debe manejar una lista de variantes, uno y solo un módulo del sistema debe conocer la lista exhaustiva”*

- Consecuencia del Principio Abierto Cerrado
- Forma fuerte de ocultamiento de la información (la lista de variantes a los clientes)
- Se favorece la continuidad
- Se limita la “distribución del conocimiento” (cliente sólo conoce lo necesario para su funcionamiento)

### **3.- Reutilización del software**

- ¿Por qué el software no es como el hardware (catálogos de dispositivos que se combinan)?
- ¿Por qué cada nuevo proyecto software arranca de la nada?
- Creciente importancia de los **componentes** en la industria del software: (COM, JavaBeans, ...)
- Internet favorece la reutilización.



**“La tecnología OO hará realidad en un futuro cercano el sueño de una industria basada en componentes”**

# Beneficios esperados de la reutilización

## A) CONSUMIR elementos reutilizables:

- Oportunidad (se reduce el tiempo de desarrollo)
  - => Mejora productividad
- Disminuye el esfuerzo del mantenimiento
- Aumenta fiabilidad
- Aumenta eficiencia

## B) PRODUCIR elementos reutilizables:

- Inversión: preservar la experiencia de los mejores desarrolladores
- “Si un elemento software se utilizará en muchos proyectos es rentable invertir en mejorar su calidad”

**“Consumir antes de producir”**

# ¿Qué debemos reutilizar?

## • PERSONAL:

- experiencia previa ayuda en el nuevo desarrollo
- necesidad de personal con experiencia para conocer qué aplicar

## • DISEÑO:

- difícil garantizar compatibilidad diseño-implementación.
- Interesante enfoques donde la diferencia entre módulo diseño y módulo de implementación desaparece
- Nos recuerda la necesidad de generalidad en los componentes

## • PATRONES DE DISEÑO:

- Ideas aplicables a toda una gama de dominios.
- Un patrón propone solución para un problema de diseño.
- No tiene que ser una mera descripción de un libro sino un componente software o un conjunto de componentes.

## ¿Qué debemos reutilizar?

- **CÓDIGO FUENTE:**

- ninguno de los anteriores se pueden incluir en un nuevo producto de software => texto fuente
- Impedimentos económicos y psicológicos
- Viola el principio de ocultamiento de información y reglas de modularidad
- Recuerda la necesidad de que componente reutilizable= elemento software

- **MÓDULOS ABSTRACTOS:**

“Son las **unidades de reuso**, software directamente utilizable con una descripción abstracta de sus propiedades”

## ¿Por qué no es común la reutilización?

- **Naturaleza repetitiva** de la programación (ordenar, buscar, recorrer, ...)
- ¿Cuántas veces en los últimos 6 meses has escrito código para buscar un elemento en una colección?
- ¿ENTONCES? {
  - Problemas técnicos
  - Problemas no técnicos



## A) Obstáculos no técnicos

- Síndrome N.I.H. (*Not Invented Here*):
  - reacción cautelosa frente a componentes nuevos
  - coste adicional de aprendizaje
- Económicos:
  - se centran en los costes a corto plazo
- Estrategias de las compañías software:
  - “¿Y si el cliente no vuelve a necesitarlos?”
- Acceso a los componentes:
  - www facilita la búsqueda de información útil
- Formato de distribución
  - ¿fuente o binario?

## B) Dificultades técnicas

- Diseñar código reutilizable es difícil.
- Hacemos las mismas cosas pero no de la misma forma.
- Difícil captura de las similitudes.
- Permitir adaptación
- La noción correcta de módulo debe reconciliar:
  - **abierto - cerrado**
  - **reutilización - extensibilidad**

## ¿Cuál será la estructura de los módulos?

Ejemplo: ”¿Hay ocurrencia de  $x$  en  $t$ ?”

```
FUNCTION busquedaColección (x: elemento; t: coleccion): BOOLEAN
VAR
  pos : posición;
BEGIN
  pos:= Comenzar(x,t);
  WHILE not Completa(pos,t) and not Encontrado(pos,x,t) DO
    pos:=Siguiente(pos, x, t);
  busquedaColección:=not Completa(pos,t);
END;
```

Rutina genérica para “búsqueda en una tabla”

## Requisitos de los módulos para facilitar la reutilización

### 1. Variación en tipos

Un módulo reutilizable de búsqueda debería ser aplicable a muchos tipos diferentes de elementos

*x: elemento (tabla que contiene tipo Elemento)*

### 2. Variación en estructuras de datos y algoritmos (= variación de implementación)

El tipo *Colección* puede estar implementado de diferentes formas.

*Siguiente(pos,x,t)* puede estar ligado a diferentes rutinas.

### 3. Independencia de la representación.

Se puede usar una operación sin conocer su implementación

*siguiente(pos,x,t)*, *comenzar(x,t)*,

*encontrado(pos,x,t)*, *completa(pos,t)*

Extensión de la regla de Ocultamiento de la Información

Importante para la extensibilidad.

# Independencia de la representación

Alternativa: (●\* Elección única)

```
if "t es de tipo A" then
    "aplicar algoritmo de búsqueda A"
elseif "t es de tipo B" then
    "aplicar algoritmo de búsqueda B"
elseif ...
```

- Este código sería incluido en:
  - **Un único módulo:** Grande y sujeto a constantes cambios
  - **En cada cliente:** Dificulta la extensibilidad
- SOLUCIÓN: **LIGADURA DINÁMICA**

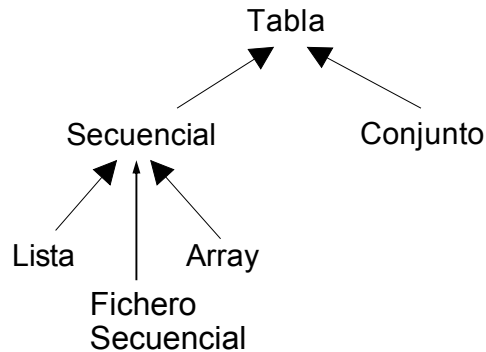
## Requerimientos de los módulos para facilitar la reutilización

### 4. Agrupar rutinas relacionadas

### 5. Capturar similitudes entre un subgrupo de un conjunto de posibles implementaciones

- Afecta a los creadores de módulos reutilizables.
- Ejemplo:
  - “Una secuencia es un caso particular de colección que puede ser implementada como un array, una lista enlazada, un fichero secuencial, ..”
- Evitar repeticiones de código en una familia de módulos relacionados:
- Definición incremental: Esquema General y Añadir propiedades específicas.

## Ejemplo: Factorizar comportamientos comunes



- La operación “*búsqueda*” se escribe una sola vez.

```

FUNCTION busqueda (x: elemento; t:Secuencia): BOOLEAN
BEGIN
  Comenzar;
  WHILE not Completa and not Encontrado(x) DO
    Avanzar;
    busqueda:=not Completa;
END;
  
```

	ARRAY	LISTA ENLAZADA	FICHERO SECUENCIAL
Comenzar	i:=1	l:=cabeza	reset
Avanzar	i:=i+1	l:=l^.next	read
Completa	i>tamaño	l=nil	eof
Encontrado	t[i] = x	l^.item = x	f^= x

**Una nueva variante sólo tiene que especificar cómo implementar estas cuatro rutinas**

## Estructuras modulares tradicionales

- Rutinas
- Paquetes o Módulos

### ¿Por qué no son suficientes?

Posibles soluciones para darles flexibilidad:

- Sobrecarga
- Genericidad

### ¿cubre los requisitos de módulos reutilizables?

Calidad del Software y OO

41

## Rutinas

- Unidad de software que puede llamar a otras unidades para ejecutar un cierto algoritmo.
- Enfoque de reutilización: **librerías de rutinas**
- Adecuadas en áreas donde pueden ser identificado un conjunto de problemas individuales con las siguientes **limitaciones**:
  1. Admiten especificación simple: pequeño conjunto de parámetros.
  2. No estén implicadas estructuras de datos complejas.
  3. Problemas claramente diferenciados.

Calidad del Software y OO

42

## Ejemplo de las limitación de una rutina

- Soluciones para: "Búsqueda en una colección secuencial":

### 1) Una rutina:

- "CASE" enorme.
- Muchos argumentos.

### 2) Conjunto de rutinas:

- Rutinas muy similares (●\* factorizar comportamiento)
- Cliente debe buscar en un laberinto de rutinas.

**Conclusión:** Un módulo reutilizable debe estar abierto a la adaptación y, la única forma de adaptación de una rutina es pasarle diferentes argumentos.

**Las rutinas no son suficientemente flexibles**

## Paquetes/Módulos

- Son unidades de descomposición de software que:
  - Son estructuras del lenguaje (nombre y ámbito claro)
  - Contienen variables y **rutinas relacionadas** = *características del paquete*
  - Admiten la **Ocultación de la Información** (*módulos abstractos*)
  - Permite la implementación de tipos definidos por el programador
- Los módulos sólo resuelven **"rutinas relacionadas"**
  - facilita depuración y cambio al proveedor
  - es más fácil para los clientes encontrar y usar los servicios de los módulos

## Ejemplo: Paquetes/Módulos

```
package TablaEnteros;
type ArbolBinEnteros
  record
    info: Integer
    izq, der: ArbolBinEnteros
  end;

new: ArbolBinEnteros begin ... end;
añadir(t:ArbolBinEnteros; x: Integer) begin ... end;
existe(t:ArbolBinEnteros; x: Integer): Boolean begin .. end
....
end
```

→ **PAQUETE**

**DECLARACIÓN DE TIPO**

**OPERACIONES SOBRE EL TIPO**

## Sobrecarga

- Identificador con más de un significado.
- Ejemplo: **Sobrecarga de rutinas**

```
FUNCTION Búsqueda (x:Cuenta; t: ListaCuentas): Boolean;
FUNCTION Búsqueda (x:CHAR; t: ARRAY [CHAR]): Boolean;
FUNCTION Búsqueda (x:REAL; t: ARRAY[REAL]): Boolean;
```

- Facilidad sintáctica orientada a los clientes.
- Permite escribir el mismo código cliente usando diferentes implementaciones de cierto concepto.

## ¿Qué nos proporciona la sobrecarga?

- ¿Resuelve “Variación en estructuras de datos y algoritmos”?
- ¿Resuelve “Independencia de la representación”?
  - No, en cada invocación “*Busqueda(x,t)*” cliente y compilador saben de que versión se trata.
  - “Independencia en la representación” exige poder escribir “*Busqueda(x,t)*” significando:  
**“Busca x en t usando el algoritmo adecuado, cualesquiera que sea la clase de colección ligada a t en tiempo de ejecución”**

## Genericidad

- Posibilidad de definir módulos parametrizados, cuyos parámetros representan tipos.
- Facilidad orientada a los creadores de módulos permite escribir el mismo código cuando se usa la misma implementación de cierto concepto, aplicado a diferentes tipos de objetos” **Ej: *Array [T], Lista[T]***
- Los módulos cliente deben instanciar el módulo genérico  
**Ej: *Array [Real], Lista[Figura]***
- **Resuelve “variación en tipos”**



# Ejemplo de Genericidad

```
Package Tabla[T];  
  type ArbolBinario  
    record  
      info: T  
      izq, der: ArbolBinario  
    end;  
  
  new: ArbolBinario begin ... end;  
  añadir(t:ArbolBinario; x: T) begin ... end;  
  existe(t:ArbolBinario; x: T): Boolean begin .. end  
  ....  
end
```

## Conclusión

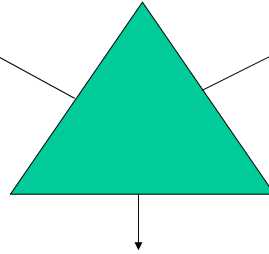
- La genericidad resuelve:
  - 1. Variación en tipos.**
- Los paquetes/módulos resuelven:
  - 5. Rutinas relacionadas.**
- **No se resuelve:**
  2. Variación en estructuras de datos y algoritmos.
  3. Independencia de la representación.
  4. Capturar similitudes entre un subgrupo de un conjunto de posibles implementaciones.

## 4.- Diseño estructurado vs diseño OO

- ¿Qué criterio usamos para encontrar los módulos?

Las tres fuerzas de la computación

Acciones/  
Funciones



Objetos/  
Datos

Procesadores

A) Unidades de descomposición funcional → Enfoque tradicional

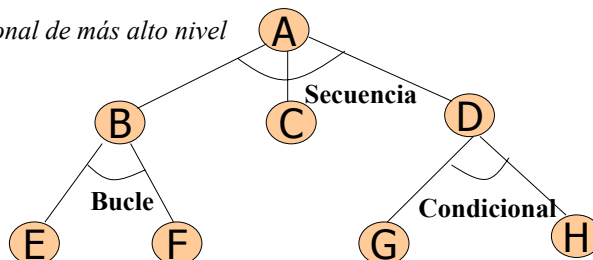
B) Basándose en los principales tipos de datos → Enfoque OO

51

### A) Descomposición Funcional

- Respuesta tradicional a la cuestión de modularización
- ¿Responde a los requisitos de modularidad (*Continuidad*)?

*Abstracción funcional de más alto nivel*



Refinamientos sucesivos

## Inconvenientes de la Descomposición Funcional

### EXTENSIBILIDAD

#### ☹ **Función principal: "Cima del sistema"**

- El "programa principal" es una propiedad volátil
- Sistemas reales no tienen "cima"
- Mejor la visión de un "conjunto de servicios"

#### ☹ **Centrado en la interfaz**

- Primera pregunta: **¿Que hará el sistema?**
- La arquitectura del software debe basarse en propiedades más profundas.

#### ☹ **Ordenación temporal prematura**

## Inconvenientes de la Descomposición Funcional

### REUTILIZACIÓN

☹ Se desarrollan elementos software para satisfacer necesidades específicas de otro elemento del nivel superior.

☹ "Cultura del proyecto actual"

- Las estructuras de datos son descuidadas
  - Funciones y datos deben jugar un papel complementario
- Cuando un sistema evoluciona los datos son más estables que los procesos.

## Ventajas de la Descomposición Funcional

- ☺ Disciplina de pensamiento lógica y bien organizada
- ☺ Técnica simple, fácil de aplicar.
- ☺ Útil para pequeños programas y algoritmos individuales.
- ☺ Buena para documentar diseños (describir algoritmos).
- ☺ Promueve el desarrollo ordenado de sistemas
- ☺ Adecuada para dominar la complejidad

## B) Descomposición basada en objetos

- **EXTENSIBILIDAD:**
  - Los objetos son más estables que las funciones
  - Punto de vista de alto nivel de los objetos = Descripción abstracta
- **REUTILIZACIÓN:**
  - Las rutinas no son suficientes como unidades de reutilización. Ej: búsqueda en una tabla
  - Tipos de objetos equipados con las operaciones asociadas proporcionan unidades estables para la reutilización

# Desarrollo de software orientado a objetos

## Definición

Método de desarrollo de software que basa la arquitectura del sistema en módulos deducidos de los tipos de objetos que se manipulan (en lugar de basarse en la función o funciones a las que el sistema está destinado a asegurar).

*No preguntes primero **qué hace el sistema**, pregunta  
**¡¡A QUIÉN LO HACE!!***

# Desarrollo de software OO

## ¿CÓMO?

1. **Encontrar tipos** de objetos relevantes
2. Encontrar **operaciones** para tipos de objetos
3. **Describir** tipos de objetos
4. Encontrar **relaciones** entre objetos
5. Usar tipos de objetos para **estructurar software**

## ¿Cómo se encuentran los objetos?

- “Los objetos están ahí para cogerlos”
- Reutilización es una fuente de tipos de objetos
- Experiencia e imitación

## ¿Cómo se describen los objetos?

Descripción de **tipos** que satisfagan criterios:

- Descripciones completas, precisas y no ambiguas.
- Descripciones independientes de la representación (abstracción)

Solución  $\Rightarrow$  TADs

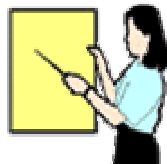
Calidad del Software y OO

59

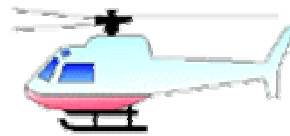
## “Los objetos están ahí para cogerlos”



Antena Parabólica



Objeto Profesor



Helicóptero

**$a + bi$**

Número Complejo



Cuenta Bancaria



Automóvil

Calidad del Software y OO

60

## Definición del objeto coche

**Funciones** que puede realizar:

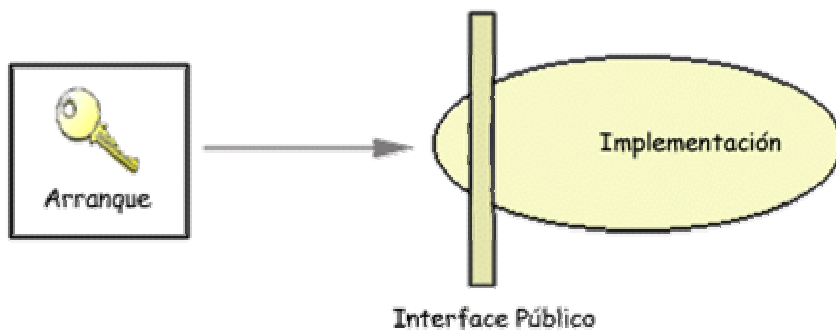
- Ir
- Parar
- Girar a la derecha
- Girar a la izquierda



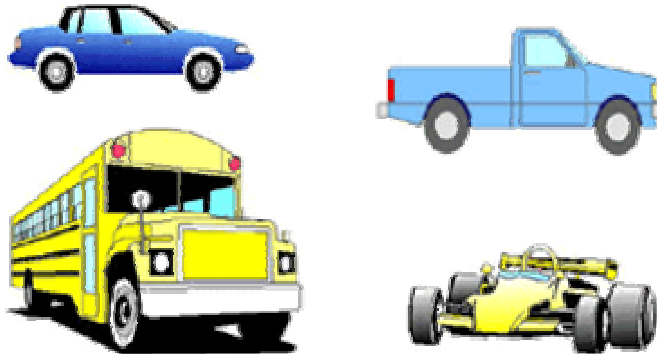
Tiene las **características**:

- Color
- Velocidad
- Tamaño
- Carburante

## Objetos se comunican mediante paso de mensajes



Los objetos con estados similares y el mismo comportamiento se agrupan en clases



Objetos de la clase **Coche**

Calidad del Software y OO

63

## ¿Qué clase de relaciones se permiten?

- B es un **cliente de** A si todo objeto de B puede contener información sobre uno o mas objetos de A
- B **hereda de** A si B denota una versión especializada de A

## ¿Estructura del software?

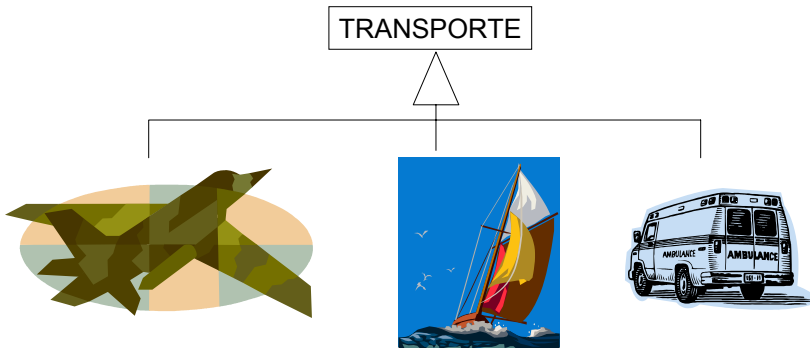
Puesto que los módulos de basarán en los tipos de objetos, las relaciones anteriores determinan las técnicas de estructuración disponibles para construir software a partir de componentes.

Calidad del Software y OO

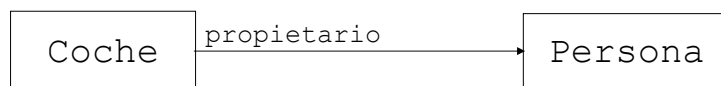
64



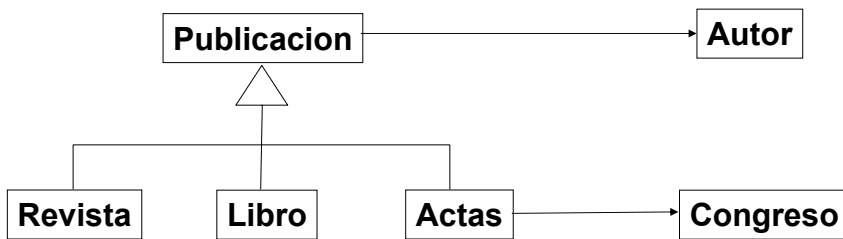
# Relaciones entre clases. Herencia



# Relación de clientela



## Relaciones entre módulos: clientela/herencia



“*Libro es una especialización de Publicacion*”  
 “*Publicacion usa servicios de Autor*”

## 5.- Tipos abstractos de datos

¿Cómo conservar la completitud, precisión y la no ambigüedad sin pagar el precio de una especificación excesiva?

**Solución:** considerar que las operaciones definen la estructura de datos.

¡Es necesario definir las de manera precisa!

- Cómo puede utilizarlas el cliente
- Qué realizarán para estos clientes

# Tipos abstractos de datos

- Un TAD proporciona esta información
- Permite la especificación **abstracta** de un tipo de objeto, libre de cuestiones de implementación.
- Consta de cuatro párrafos:
  - **TIPO**: tipo (=cjo de objetos) que se está especificando.
  - **FUNCIONES**: signatura (tipo de los argumentos y resultado) de las operaciones aplicables.
  - **AXIOMAS**: definición **implícita** de las propiedades de las funciones
  - **PRECONDICIONES**: dominio de las funciones parciales

## Ejemplo: TAD "Pila"

**TIPO** Stack[T]

**FUNCIONES**

put: Stack[T] x T → Stack[T]

remove: Stack[T] ↦ Stack[T]

Funciones orden

item: Stack[T] ↦ T

empty: Stack[T] → Boolean

Funciones de consulta

new: Stack[T]

Función de creación

## Tipo abstracto de dato "Pila"

### AXIOMAS

Para  $x: T, s: \text{STACK}[T]$ ;

$\text{item}(\text{put}(s,x)) = x$

$\text{remove}(\text{put}(s,x)) = s$

$\text{empty}(\text{new})$

$\text{not empty}(\text{put}(s,x))$

### PRECONDICIONES

$\text{item}(s: \text{Stack}[T])$  require  $\text{not empty}(s)$

$\text{remove}(s: \text{STACK}[T])$  require  $\text{not empty}(s)$

## De los TADs a las clases

- Tenemos como punto de partida una teoría matemática para el modelado de estructuras de datos.
- Los TAD proporcionan un mecanismo de descripción de alto nivel, libre de cuestiones de implementación.
- Una clase es una implementación de un tipo abstracto de dato, TAD.
- "Construcción de software OO consiste en crear sistemas software como **colecciones estructuradas** de implementaciones de **TAD's**"

# Desarrollo de software OO

- ¿Qué necesitamos **para implementar un TAD**?
  - La especificación de un TAD (funciones, axiomas y pre.)
  - Elegimos una representación
  - Correspondencia de las funciones con la representación (rutinas y atributos)
- **TAD y Ocultamiento de información**
  - La especificación del TAD determina la parte pública
  - La representación de los objetos y la implementación de las funciones son la parte privada.