Universidad de Alcalá

Departamamento Ciencias de la Computación

Programación Orientada a Objetos

Tema 2:

Fundamentos de la programación orientada a objetos

Tema 2-2: Modularidad - Tipos abstractos de datos

Universidad de Alcalá

Departamento Ciencias de la Computación

Contenidos

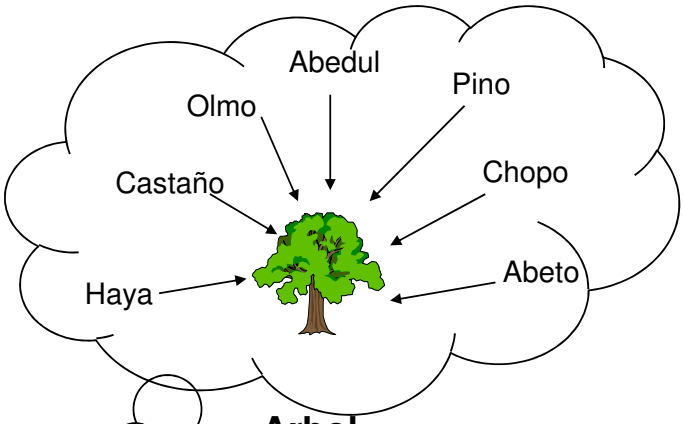
- Tema 2-2: Modularidad - Tipos abstractos de datos

1. Abstracción
2. Tipos de datos
3. Tipos abstractos de datos
4. Modularidad
5. Diseño modular
6. Reutilización

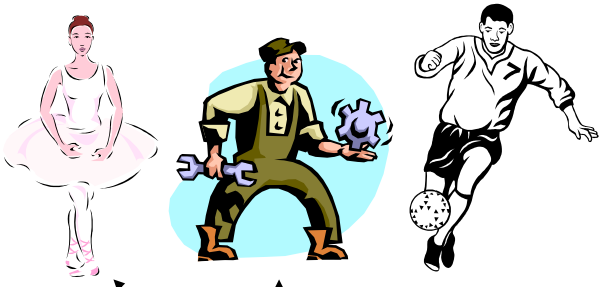
Abstracción

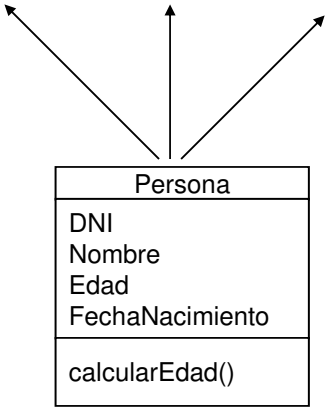
- Supresión intencionada, u ocultamiento, de algunos detalles de un proceso o artefacto, con el objeto de destacar de manera más clara otros aspectos, detalles o estructuras.
- Capacidad de centrarse en las características esenciales de las distintas partes de un sistema, ignorando sus propiedades accidentales.
- Permite dividir la información en componentes aislados que posteriormente se ensamblan para construir el "todo".
- Limitación de la capacidad humana para operar la complejidad:
  - Ordenando el caos: "*divide et impera*".
  - En SW: Abstracción → Modularidad

Abstracción



Arbol
- altura
- tipo de madera
- tipo de hoja
- tipo de fruto





Abstracción

Abstracción aplicada:

Diferentes niveles: Nos centramos en los elementos más grandes e importantes.

Progresivamente: Tratamos volúmenes de información menores que revelen más detalles.

Diferentes tipos: Funcional o procedural, de Datos.

Abstracción

- **Encapsulación:**

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

  La encapsulación es un mecanismo que consiste en organizar datos y operaciones de una estructura, conciliando el modo en que el módulo se implementa, es decir, evitando el acceso a datos por cualquier otro medio distinto a los especificados. Por lo tanto, la encapsulación garantiza la integridad de los datos que contiene un módulo.

Tipos de Datos

- Un tipo de dato es un conjunto de valores y un conjunto de operaciones definidas por sus valores.
- Tipo de dato = Representación + Operaciones.
- Ejemplos:
  - Tipo de datos entero, operaciones de +,-,*,/.
  - Tipo cadena, operaciones de concatenación, subcadena, etc.

Tipos Abstractos de Datos

- Los TADs permiten ampliar los tipos de datos definidos por el lenguaje de programación.
- Un tipo de dato definido por el programador se denomina TAD.
- Un TAD es un tipo de datos que consta de datos y operaciones que se pueden realizar sobre esos datos.
- Los TADs ocultan la implementación de las operaciones definidas por el usuario asociadas con el tipo de datos.
- La ocultación de información de un TAD significa que poseen interfaces públicos (operaciones que se pueden realizar), sin embargo, las implementaciones de esos interfaces son privados.

Tipos Abstractos de Datos

- **Un TAD consta de:**

TIPO: tipo (=cjto. de objetos) que se está especificando
FUNCIONES: signatura (tipo de los argumentos y resultado)
AXIOMAS: definición implícita del valor de la función
INVARIANTES: condición booleana que debe mantenerse con exactitud
PRECONDICIONES
POSTCONDICIONES

Universidad de Alcalá
Departamento Ciencias de la Computación

Tipos Abstractos de Datos

Ejemplo TAD "Pila"

TIPO *Pila[X]*

FUNCIONES
poner: Pila[X] x X → Pila[X]
vacia: Pila[X] → Boolean
item: Pila[X] → X
new: Pila[X]

AXIOMAS
Para x: T, s: Pila[T];
item(poner(s,x)) = x
vacia(new)
not vacia(poner(s,x))

PRECONDICIONES
item (s:Pila[T]) requiere not vacia(s)

Modularidad

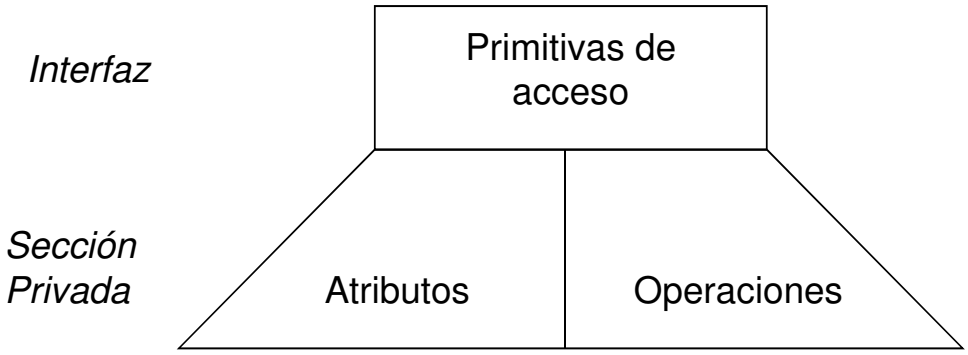
- Programa modular: Formado por un conjunto de módulos.
- Módulo: Unidad básica de descomposición de un sistema software. Los módulos deben ser lo más independientes posibles.
- Un método de construcción de software es modular si **ayuda a producir sistemas software a partir de elementos autónomos interconectados por una estructura simple y coherente.**

Modularidad

- La programación modular trata de descomponer un programa en un pequeño número de abstracciones coherentes que pertenecen al dominio del problema y cuya complejidad interna esta oculta por la interfaz.
- Para obtener unidades modulares el lenguaje debe proporcionar estructuras modulares con las cuales se puedan describir las diferentes unidades. En POO son las Clases.

Modularidad

- Un módulo se estructura mediante una interfaz y una implementación.
- Esta compuesto por un conjunto de operaciones y atributos.



Modularidad

- "Propiedad que tiene un sistema que ha sido descompuesto en un conjunto de módulos *cohesivos* y *débilmente acoplados*" [Booch'96]
- **Alta cohesión**:
  - Un módulo con responsabilidades altamente relacionadas y que no hace una gran cantidad de trabajo.
- **Bajo acoplamiento**:
  - Los módulos se comunican mediante *interfaces bien definidas*.
  - Un módulo debe comunicarse con el menor número de módulos posible.
  - Si dos módulos se comunican deben intercambiar la menor cantidad de información posible.
  - Favorece:
    - **Comprensión modular**
    - **Continuidad modular**
    - **Protección modular**

- **Criterios del Diseño Modular:**
  - Descomposición
  - Composición
  - Comprensibilidad
  - Continuidad
  - Protección

- **Descomposición Modular:**
- "*Un método de construcción de software satisface la Descomposición Modular si ayuda a la tarea de descomponer el problema de software en un pequeño número de subprogramas menos complejos, interconectados mediante una estructura sencilla, y suficientemente independientes para permitir que el trabajo futuro pueda proseguir por separado en cada uno de ellos*"
- Proceso iterativo – si tenemos subproblemas complejos hay que seguir descomponiendo.

- **Composición Modular:**

- "*Un método de construcción de software satisface la Composición Modular si favorece la producción de elementos software que se puedan combinar libremente unos con otros para producir nuevos sistemas, posiblemente en un entorno bastante diferente de aquel en que fueron desarrollados inicialmente*"

- Ejemplo: Librerías software.

- **Comprensibilidad Modular:**

- "*Un método de construcción de software favorece la Comprensibilidad Modular si ayuda a producir software en el cual un lector humano puede entender cada módulo sin tener que conocer los otros, o, en el peor caso, teniendo que examinar sólo unos pocos de los restantes módulos*"

- Facilita el proceso de mantenimiento.

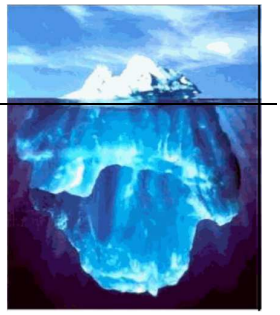
- **Continuidad Modular:**
- "*Un método de construcción de software favorece la Continuidad Modular si en la arquitectura software que produce, un pequeño cambio en la especificación de un problema provoca sólo cambios en un solo módulo o en un pequeño número de módulos*"
- Ligado al objetivo de la extensibilidad.

- **Protección Modular:**
- "*Un método de construcción de software favorece la Protección Modular si produce arquitecturas software en las cuales el efecto de una situación anormal que se produzca dentro de un módulo durante la ejecución queda confinado a dicho módulo o en el peor caso se propaga solo a unos pocos módulos vecinos*"

- **Principios de Diseño Modular:**
  - Ocultación de Información
  - Auto-documentación
  - Acceso Uniforme
  - Abierto-Cerrado
  - Elección Única

- **Ocultación de la 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*"
- Consiste en ocultar los detalles de la implementación al código cliente.

- Ocultación de información:
  - Todos los módulos deben seguir el principio de ocultación de información.
  - Una abstracción de datos puede verse como que tiene dos caras:
    - Interfaz: Parte pública visible a los clientes (operaciones que definen el comportamiento)
    - Implementación: Parte privada visible sólo dentro del módulo



- **Auto-documentación:**
- *"El diseño de un módulo debería esforzarse para lograr que toda la información relativa al módulo forme parte del propio módulo"*
- Relacionado con los criterios de comprensibilidad y continuidad modular.
- Herramientas que generan la documentación a partir de los módulos documentados.

- **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 un servicio proporcionado por el módulo de cuentas bancarias,
  - c.saldo → saldo es un atributo
  - 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.

- **Abierto-cerrado:**
- *"Un módulo se considera a la vez cerrado (terminado, útil o activo) y abierto (cambios y modificaciones). No debe afectar a los demás módulos"*
  - Un módulo está abierto si está disponible para ampliarlo. Extender o modificar la funcionalidad.
  - Un módulo está cerrado si está disponible para su uso.

- **Abierto-cerrado:**
  - Los dos objetivos son **incompatibles** con las técnicas tradicionales:
    - o está abierto → no se puede utilizar todavía.
    - o se cierra → cualquier cambio provoca cambios en cadena.
  - Necesitamos un mecanismo que nos permita:
    - Adaptar un módulo sin afectar a los clientes.
    - Que un módulo esté cerrado y abierto al mismo tiempo.
    - **SOLUCIÓN**: mecanismo de **Herencia**.

- **Principio de Elección Única:**
  - Ejemplo: Sistema bibliotecario.
  - Tendremos un módulo que contiene la estructura de datos "PUBLICACION" (registro variante):

```
type PUBLICACION
  record
        autor, titulo: STRING;
        año_publicacion: INTEGER;
  case tipo_pub: (libro, revista, acta) of
        libro: (editorial: STRING);
        revista: (volumen, numero: STRING);
        actas: (editorial, lugar: STRING)
  end
```

- **Principio de Elección Única:**
  - Ejemplo: Sistema bibliotecario.
  - Módulo cliente que manipula una *p: PUBLICACION*:

```
case p of
        libro: instrucciones para acceder al campo p.editorial…
        revista: instrucciones para acceder a los campos
                 p.volumen y p.numero…
        actas: : instrucciones para acceder a los campos
                 p.editorial y p.lugar…
end
```

  - ¿Y si necesitamos otro tipo de publicación? Tendremos que modificar el módulo que contiene PUBLICACION y actualizar todos los módulos clientes.

Universidad de Alcalá
Diseño Modular
Departamento Ciencias de la Computación

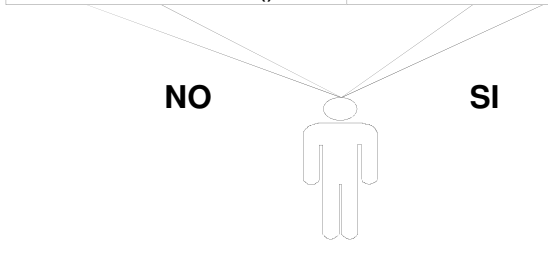
- **Principio de Elección Única:**
  - *"Siempre que un sistema software debe manejar una lista de variantes, uno y sólo uno de los módulos del sistema debe conocer la lista exhaustiva"*
  - Muy relacionado con el P. Abierto-Cerrado.
  - Favorece la extensibilidad.
  - SOLUCIÓN: Definir jerarquías de herencia.

- **Lenguajes de programación:**
  - La abstracción es la clave para diseñar buen software.
  - Los lenguajes de programación de alto nivel permiten al programador abstraerse de la arquitectura de la máquina donde se ejecuta el software (de propósito general).
  - Mecanismos para diseñar programas modulares:
    - Procedimientos o funciones
    - Módulos
    - Tipos abstractos de datos (TADS)
    - Clases/Objetos

EJEMPLO: Módulo que define "cuentas bancarias"

Un modulo incluye una **estructura de datos** junto con un conjunto de **operaciones** para manipularla.

Representación	**Interfaz**
NumCuenta: String Titular: String Saldo: double FechaApertura: Date	ingresar() retirar() verSaldo() transferencia()
Operaciones	
ingresar() retirar() verSaldo() transferencia() calcularIntereses()	



Reutilización

- ¿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?
- Herramientas que favorecen la reutilización:
  - Librerías en lenguajes de programación.
  - Creciente importancia de los **componentes** en la industria del software.
  - Internet.

Reutilización

- **Beneficios esperados de la reutilización:**
- **CONSUMIR elementos reutilizables:**
  - Oportunidad (se reduce el tiempo de desarrollo).
    => Mejora la productividad.
  - Disminuye el esfuerzo del mantenimiento.
  - Aumenta fiabilidad.
  - Aumenta eficiencia.
- **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:
  - La experiencia previa ayuda en el nuevo desarrollo.
- DISEÑO**:**
  - Difícil garantizar compatibilidad diseño-implementación.
  - Seguir un enfoque donde la diferencia entre módulo diseño y módulo de implementación desaparece.
  - Necesidad de generalidad en los componentes.
- PATRONES DE DISEÑO:
  - Ideas aplicables a toda una gama de dominios.
  - Un patrón propone una solución para un problema de diseño.

- **¿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?
- **Obstáculos:**
  - 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 necesitarnos?".
- **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**