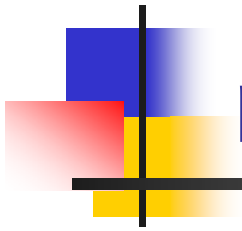
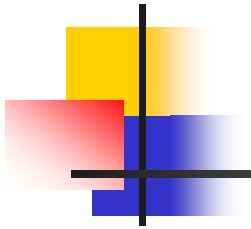


TEMA 1: Orientación a Objetos, una técnica para mejorar la calidad del software

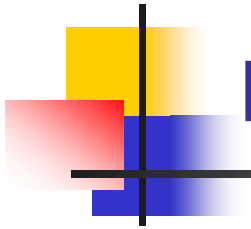
Programación Orientada a Objetos

Curso 2009/2010



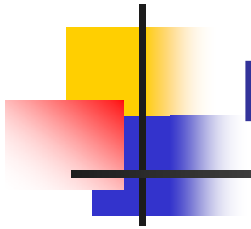
Índice

- Calidad del software
- Modularidad
- Reutilización
- Criterios para encontrar los módulos: Orientación a Objetos
- Lenguajes de Programación OO
- Modelo de objetos



Problemas en la creación de software

- A finales de los 60 se acuñó el término *crisis del software:*
  - Los proyectos no cumplían los plazos y presupuestos.
- Dificultades inherentes a la *naturaleza del software*:
  - Complejidad
    - Dificultad de enumerar todos los estados posibles del programa
    - Dominios de aplicación complejos
    - Dificultad de comunicación entre los miembros del equipo
  - Sujeto a continuos cambios



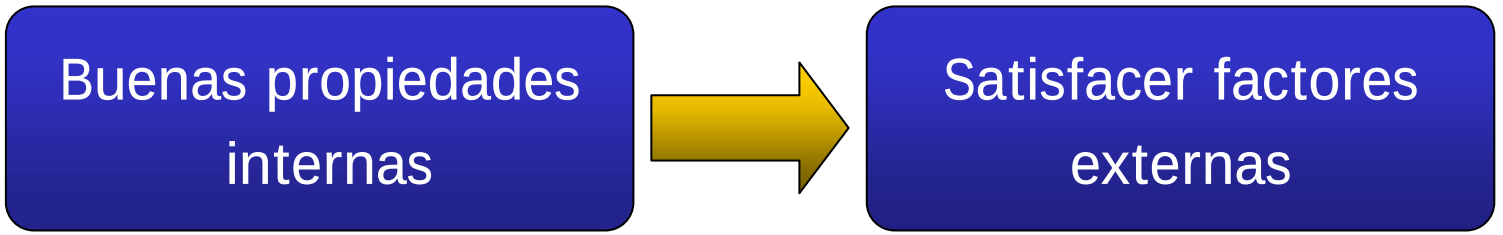
Problemas en la creación de software

- "La construcción de software siempre será una tarea difícil. **No hay bala de plata**" [Brooks,1987]

- **Soluciones:**
  - Reutilizar código de calidad
  - Buenos programadores/diseñadores

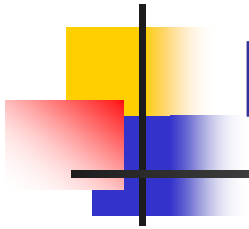
Calidad del software

- **Factores externos:**
  - Pueden ser detectados por los *usuarios*
  - Calidad externa es la que realmente preocupa
- **Factores internos:**
  - Sólo lo perciben los diseñadores e implementadores
  - Medio para conseguir la calidad externa
- Objetivo:



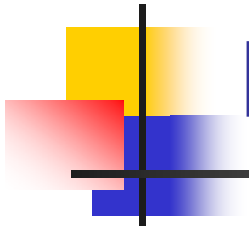
Calidad del software

- **Factores Externos**
  - Corrección
  - Robustez
  - Extensibilidad
  - Reutilización
  - Compatibilidad
  - Eficiencia
  - Portabilidad
  - Facilidad de uso
  - Funcionalidad
  - Oportunidad
- **Factores Internos**
  - Modularidad
  - Legibilidad



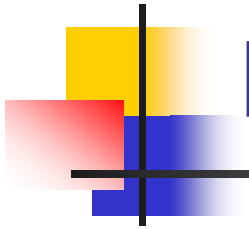
Factores de calidad externos

- Corrección:
  - Es la capacidad de los productos software de realizar con exactitud su tarea, tal y como es definida en la especificación.
- Robustez:
  - Es la capacidad de los productos software de reaccionar adecuadamente ante situaciones excepcionales



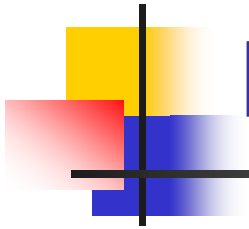
Factores de calidad externos

- Extensibilidad:
  - Es la facilidad de adaptación de los productos software a los cambios en la especificación.
  - La dificultad de adaptación es proporcional al tamaño del sistema.
  - Principios esenciales para facilitar la extensibilidad
    - Simplicidad de la arquitectura del software
    - Descentralización: módulos autónomos



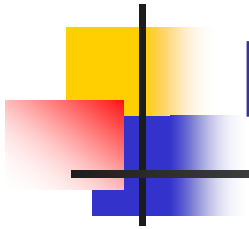
Factores de calidad externos

- Reutilización:
  - Es la capacidad de un producto software de ser utilizado en la construcción de diferentes aplicaciones
  - Se escribe menos software, luego se puede dedicar mas tiempo a mejorar otros factores como la fiabilidad (corrección y robustez)
- Compatibilidad:
  - Es la facilidad de combinar unos elementos software con otros



Factores de calidad externos

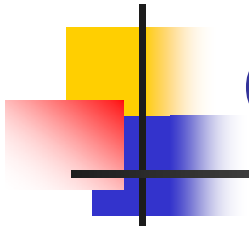
- Eficiencia:
  - Es la capacidad de un sistema software de requerir la menor cantidad posible de recursos hardware.
- Portabilidad:
  - Es la facilidad de transferir productos software a diferentes plataformas (entornos hw y sw)



Factores de calidad externos

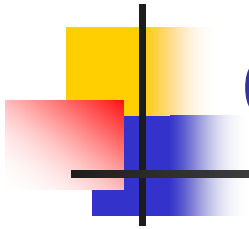
- Facilidad de uso:
  - Es la facilidad con la que personas con diferentes niveles de experiencia pueden aprender a usar los productos software y aplicarlos a resolver problemas. También incluye la facilidad de instalación, operación y supervisión.
- Funcionalidad:
  - Conjunto de posibilidades ofrecido por un sistema
  - Evitar añadir propiedades de forma incontrolada
  - 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 de calidad externos

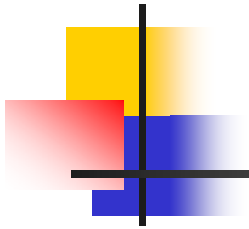


- Economía:
  - completarse con el presupuesto asignado
- Integridad:
  - proteger contra modificaciones y accesos no autorizados
- Facilidad para reparación de errores
- Facilidades de verificación:
  - datos de prueba y procedimientos para detectar fallos

Consecuencia de los criterios de calidad



- Buena **documentación**:
  - externa (usuarios) → facilidad de uso
  - interna (desarrolladores) → extensibilidad
  - interfaz del módulo → extensibilidad y reutilización
- Pueden entrar en **conflicto**. Por ejemplo:
  - Eficiencia y portabilidad
  - Corrección y reutilización
  - Facilidad de uso e integridad
  - Economía y funcionalidad

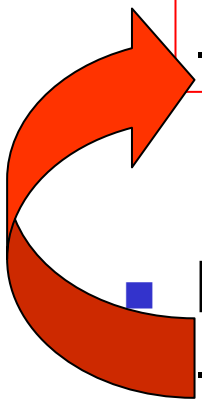


Mantenimiento del software

- Fase del ciclo de vida del software que sucede después de que se haya distribuido.
- Se le dedica el 70% del coste del software
- El mantenimiento comprende:
  - DEPURACIÓN: quitar errores
  - MODIFICACIÓN: adaptación a los cambios
- Se favorece el mantenimiento si:
  - El sistema es extensible y reutilizable
  - Es fácil reparar errores

Calidad del software

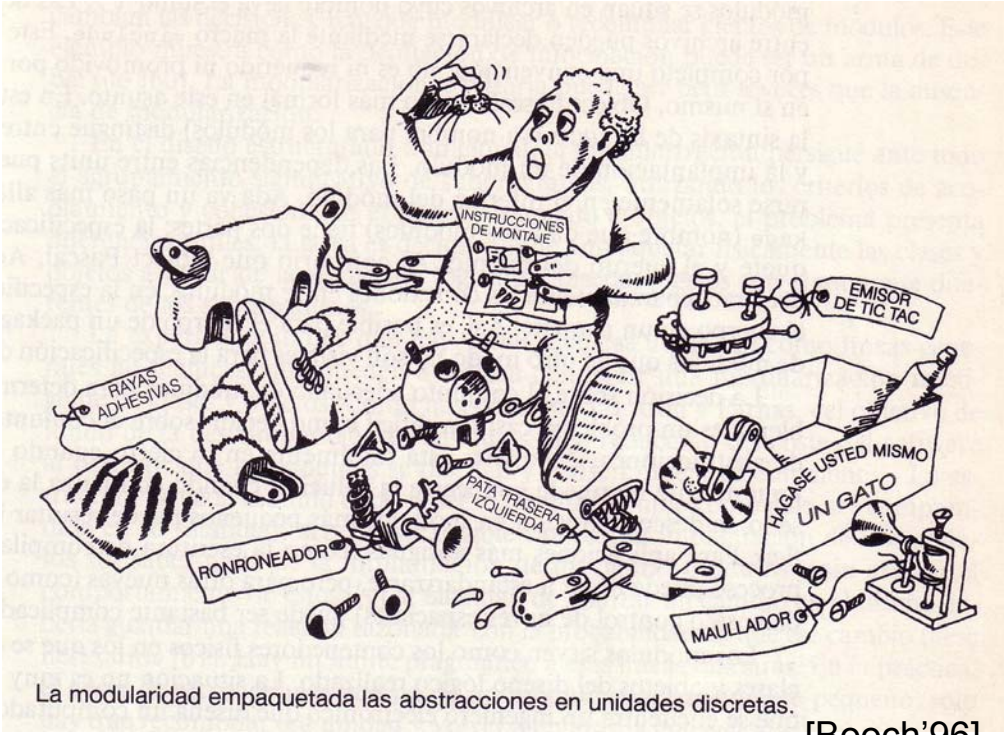
- **Factores Externos**
  - Corrección
  - Robustez
  - Extensibilidad
  - Reutilización
  - Compatibilidad
  - Eficiencia
  - Portabilidad
  - Facilidad de uso
  - Funcionalidad
  - Oportunidad



- **Factores Internos**
  - Modularidad
  - Legibilidad

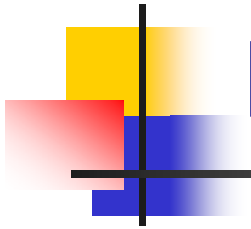
Factores de calidad internos: Modularidad

- "Propiedad que tiene un sistema que ha sido descompuesto en un conjunto de ***módulos*** *cohesivos* y *débilmente acoplados*"



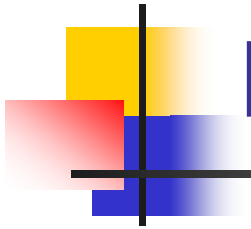
La modularidad empaquetada las abstracciones en unidades discretas.

[Booch'96]



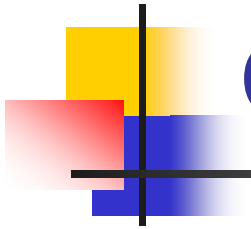
Modularidad

- **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.



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

INTERFAZ

parte pública visible a los clientes

IMPLEMENTACIÓN

Parte privada visible sólo dentro del módulo

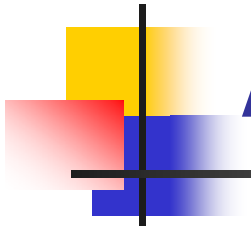
Ocultación de Información. Ejemplo: Pila

INTERFAZ

```
Push(x:T)
Pop(X:T)
```

IMPLEMENTACIÓN

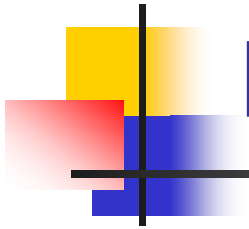
```
contenido = array [1.. MAX] de T
Constante MAX = 100
Variable tope: entero
```



Auto-documentación

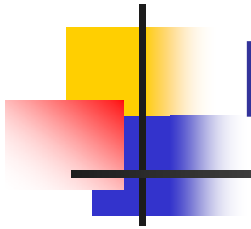
- "*El diseño de un módulo debería esforzarse para lograr que toda la información relativa al módulo forme parte del propio módulo*"

- Útiles herramientas que generan la documentación de usuario a partir de los módulos documentados



Principio de acceso uniforme

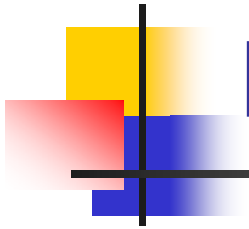
- "*Todos los servicios ofrecidos por un módulo deben estar disponibles mediante una notación uniforme, que no considere si se han implementado mediante almacenamiento o cálculo*"
- Sea c una variable representando una cuenta bancaria y `saldo` un servicio proporcionado por el módulo de cuentas bancarias,
  - `c.saldo` → saldo es un campo
  - `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 de Abierto-Cerrado

- *"Los módulos deberían ser **a la vez** abiertos y cerrados"*

- Un módulo está abierto si está disponible para ampliarlo
  - Extender o modificar la funcionalidad
- Un módulo está cerrado si está disponible para su uso



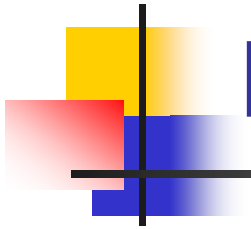
Principio de Abierto-Cerrado

- Los dos objetivos son incompatibles con las técnicas tradicionales:
  - o está abierto → no se puede utilizar todavía
  - o se cierra → cualquier cambio provoca cambio en cadena o gestión de versiones
- 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

```
if (tipo == CIRCULO)
    print "Circulo. r=" + radio;
else if (tipo == CUADRADO)
    print “Cuadrado. lado=" + longLado;
else if (tipo == RECTANGULO)
    print "Rect. h=" + altura + " b=" + base;
```

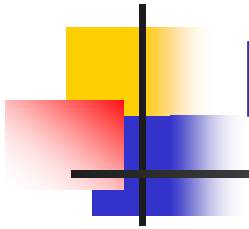
- La misma estructura para el cálculo de:
  área, intersección, ...
- ¿Si añadimos un nuevo tipo de figura?
  - Hay que modificar todos los métodos
  - Dificulta extensibilidad, reutilización y por tanto el mantenimiento



Principio de Elección Única

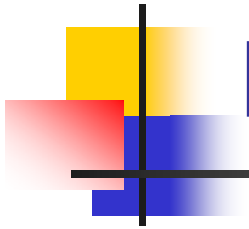
- *"Siempre que un sistema software debe manejar una lista de variantes, uno y sólo uno de los módulos del sistema debe conocer la lista exhaustiva"*

- Muy relacionado con el P. Abierto-Cerrado
- Favorece la extensibilidad
- **SOLUCIÓN:** Definir jerarquías de herencia



Reutilización del software

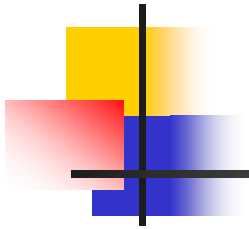
- ¿Por qué el software no es como el hardware (catálogos de dispositivos que se combinan)?
- ¿Por qué cada nuevo proyecto software arranca de la nada?



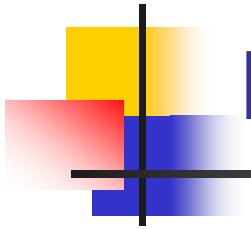
Problemas Reutilización

- No técnicos:
  - Cautela respecto al uso del código desarrollado por otros
  - Se requiere una mayor inversión
- Técnicos:
  - A pesar de la naturaleza repetitiva de la programación, hacemos las mismas cosas pero no de la misma forma.
  - Difícil capturar las similitudes.
  - Se debe permitir la adaptación (P. Abierto-Cerrado).

Beneficios esperados de la reutilización del software



- Beneficios **de reutilizar** software:
  - Aumento de la productividad
  - Disminuye el esfuerzo de mantenimiento
  - Aumenta la fiabilidad y eficiencia
- Beneficios **de producir** software reutilizable:
  - Preservar la experiencia de los mejores desarrolladores

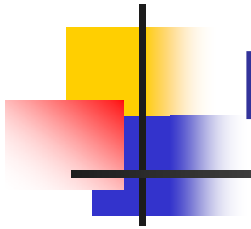


Requisitos del código reutilizable

- Ejemplo: algoritmo de búsqueda de un elemento en una colección secuencial:

```
operacion buscar (x: T; C: Coleccion[T]): booleano
  Comenzar(C)
  mientras Actual(C) ≠ x AND NOT EsUltimo(C) hacer
   Avanzar(C)
  devolver Actual(C) == x
```

¿Qué requisitos debe cumplir para que sea reutilizable?



Requisitos del código reutilizable

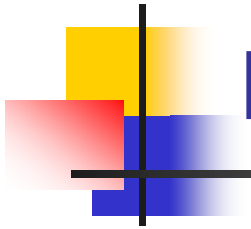
- Variación de tipos
  - El algoritmo de búsqueda debería ser aplicable a muchos tipos diferentes de elementos (enteros, reales, etc.)
- Variación en estructuras de datos y algoritmos, variación de implementación
  - El tipo `Colección[T]` puede estar implementado de diferentes formas (`Tabla`, `Lista`, etc.)
  - `Comenzar`, `Avanzar`, `Actual`, `EsUltimo` puede estar ligado a diferentes rutinas según sea el tipo de la colección

Requisitos del código reutilizable

- Independencia de la representación
  - Se puede usar una operación sin conocer su implementación → `existe = buscar(e,c)`
  - Evitar análisis por casos en función del tipo de la colección:

```
if “C es de tipo Lista” then
  “aplicar algoritmo de búsqueda en Listas”
else if “C es de tipo Tabla” then
  “aplicar algoritmo de búsqueda en Tablas”
 else if …
```

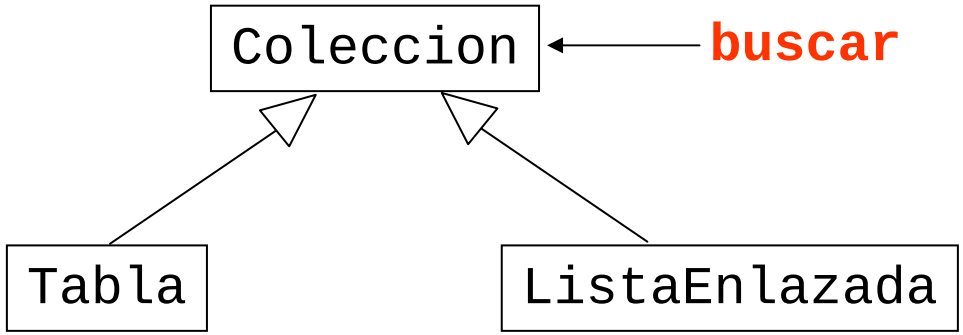
    - Viola los principios de diseño modular: Elección única, Abierto-Cerrado → Dificulta la extensibilidad
  - **SOLUCIÓN**: Mecanismo automático que determine la versión a ejecutar (LIGADURA DINÁMICA)



Requisitos del código reutilizable

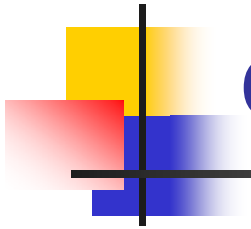
- Agrupar rutinas relacionadas
  - Definir en un mismo módulo las operaciones de un tipo
- Factorizar comportamiento común
  - Ejemplo: “Una secuencia es un caso particular de colección que puede ser implementada como un array, una lista enlazada, un fichero secuencial, ..”
  - Evitar repeticiones de código en una familia de módulos relacionados.
  - Definición incremental: Esquema General y Añadir propiedades específicas.

Ejemplo: Factorizar comportamiento común

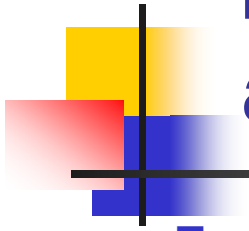


	ARRAY	LISTA
Comenzar	i:=1	l=cabeza
Avanzar	i:=i+1	l=l->next
EsUltimo	i>tamaño	l=null
Actual	t[i]	l->item

Ejemplo: Factorizar comportamiento común

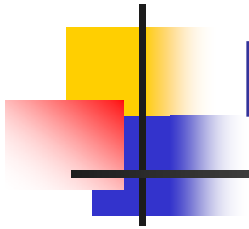


- La operación de búsqueda se escribe una única vez para toda colección secuencial
- Una nueva variante sólo tiene que especificar cómo implementar estas cuatro rutinas
- **SOLUCIÓN:** Métodos Plantilla



Limitación de las rutinas para alcanzar los criterios de reutilización

- Soluciones para implementar el algoritmo de búsqueda en una colección secuencial:
  - Una única rutina que intente abarcar todos los casos:
    - Análisis por casos enorme.
    - Muchos argumentos.
    - Violaría los principios de diseño modular: elección única, abierto-cerrado (la única forma de adaptar una rutina es pasarle diferentes argumentos).
  - Biblioteca de rutinas :
    - Enfoque clásico de reutilización
    - Rutinas muy similares (no se factoriza comportamiento)
- Las rutinas no son suficientemente flexibles para satisfacer los requisitos de reutilización.



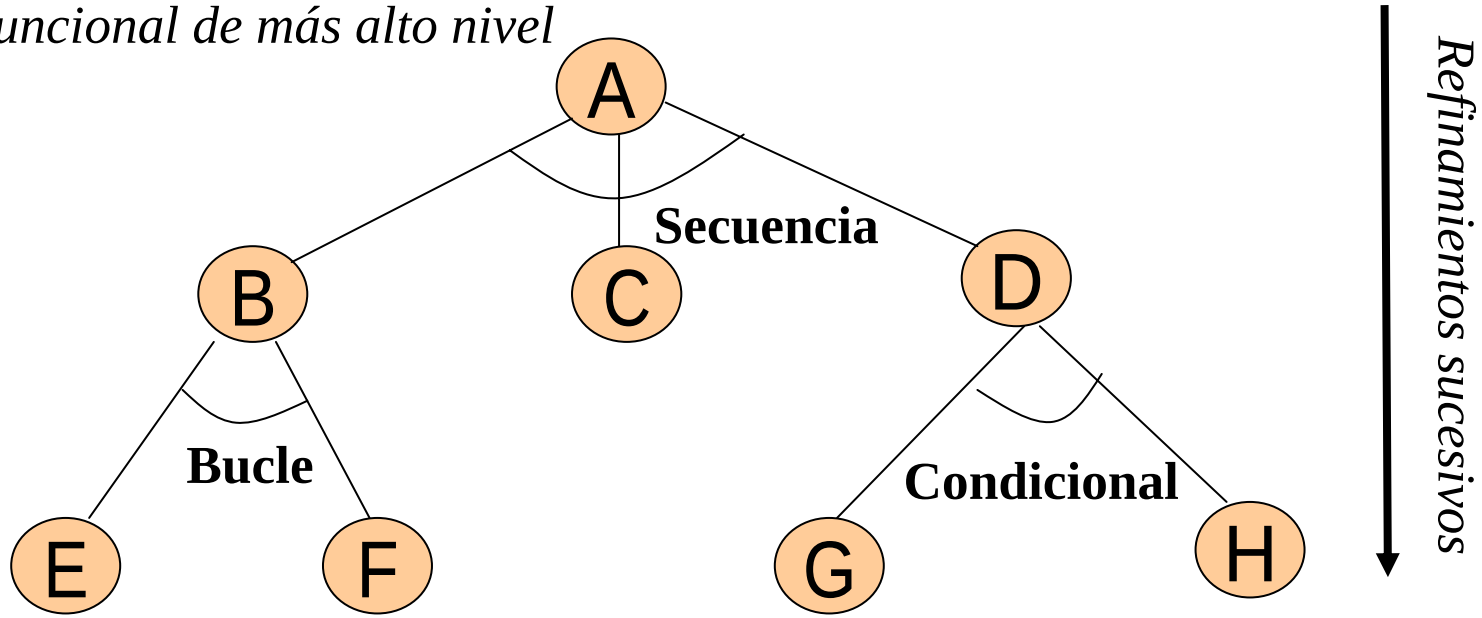
Enfoque estructurado vs. OO

- ¿Qué criterio utilizamos para encontrar los módulos?
- Descomposición modular atendiendo:
  - Funciones:
    - Enfoque tradicional
    - Descomposición funcional descendente
  - Datos:
    - Enfoque orientado a objetos

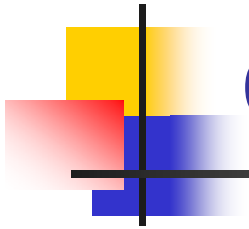
Descomposición funcional descendente

¿Qué hace el sistema?

Abstracción funcional de más alto nivel

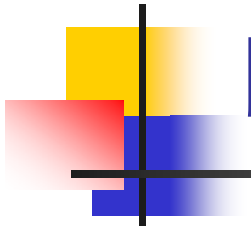


Limitaciones de la descomposición funcional



- Se basa en propiedades poco estables que dificulta la extensibilidad
  - Supone que todo sistema se caracteriza por una función principal
  - Se basa en la interfaz externa
  - Ordenación temporal prematura
- No promueve la reutilización
  - Se desarrollan elementos software para satisfacer necesidades específicas de otro elemento del nivel superior.

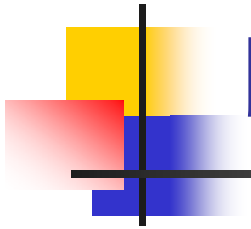
Descomposición modular basada en los datos



- Los datos son más estables que las funciones lo que favorece la extensibilidad

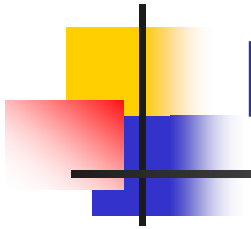
- Los tipos de datos equipados con las operaciones asociadas proporcionan unidades estables para la reutilización

Descomposición modular basada en los datos



- ¿Qué significa **Orientación a Objetos**?
  - El software se organiza como una colección de objetos que contienen tanto estructura como comportamiento.

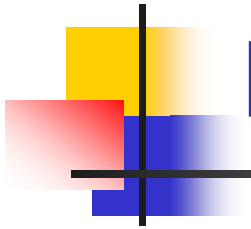
- ¿Qué es el **desarrollo Orientado a Objetos**?
  - Una nueva forma de pensar acerca del software basándose en abstracciones que existen en el mundo real.



Desarrollo de software OO

- 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

- Encontrar los objetos relevantes
- Encontrar las operaciones para los tipos de objetos
- Describir los tipos de objetos
- Encontrar relaciones entre objetos
- Utilizar los tipos de objetos y las relaciones para estructurar el software

Ejemplo: objeto coche

id: número de bastidor

Funciones que puede realizar:

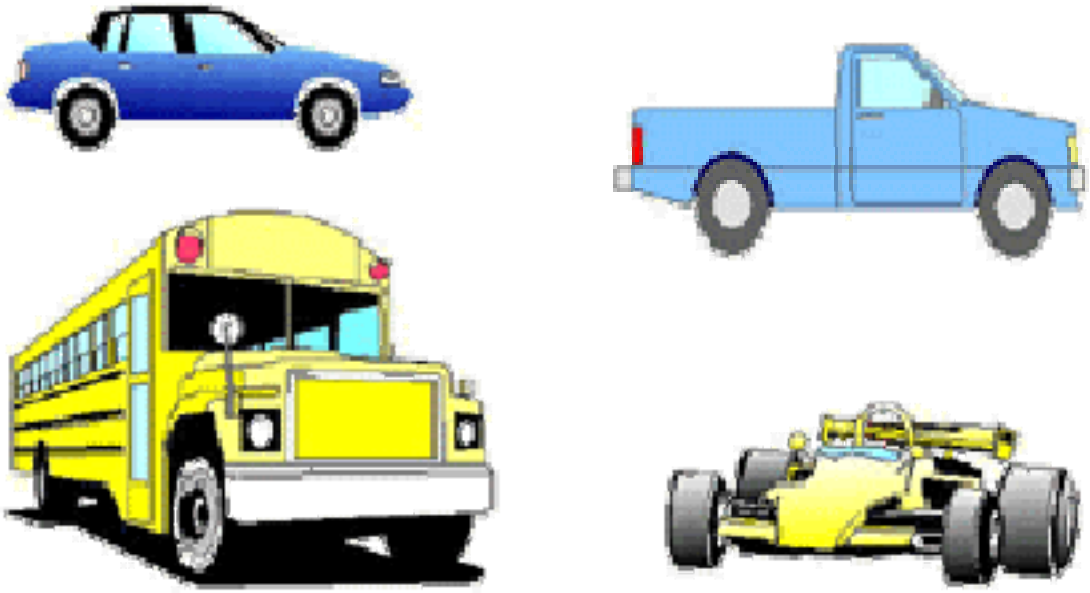
- Ir
- Parar
- Girar a la derecha
- Girar a la izquierda
- Arrancar

Tiene las **características**:

- Color
- Velocidad
- Tamaño
- Carburante

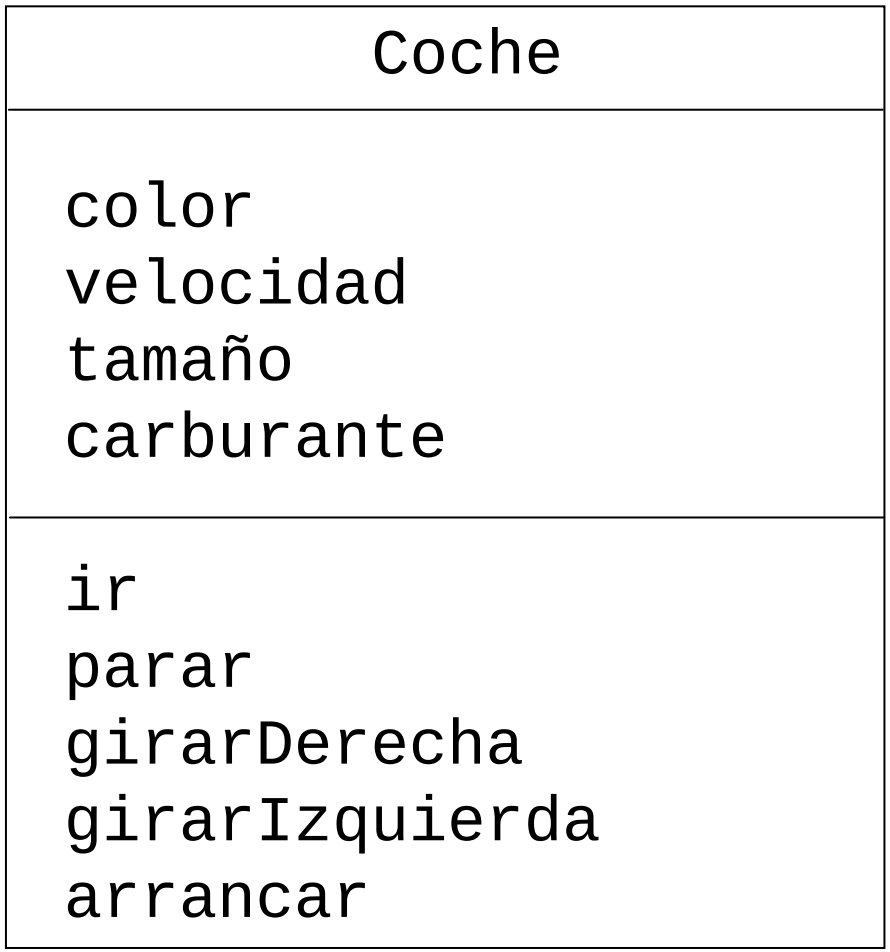
Clases de objetos

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

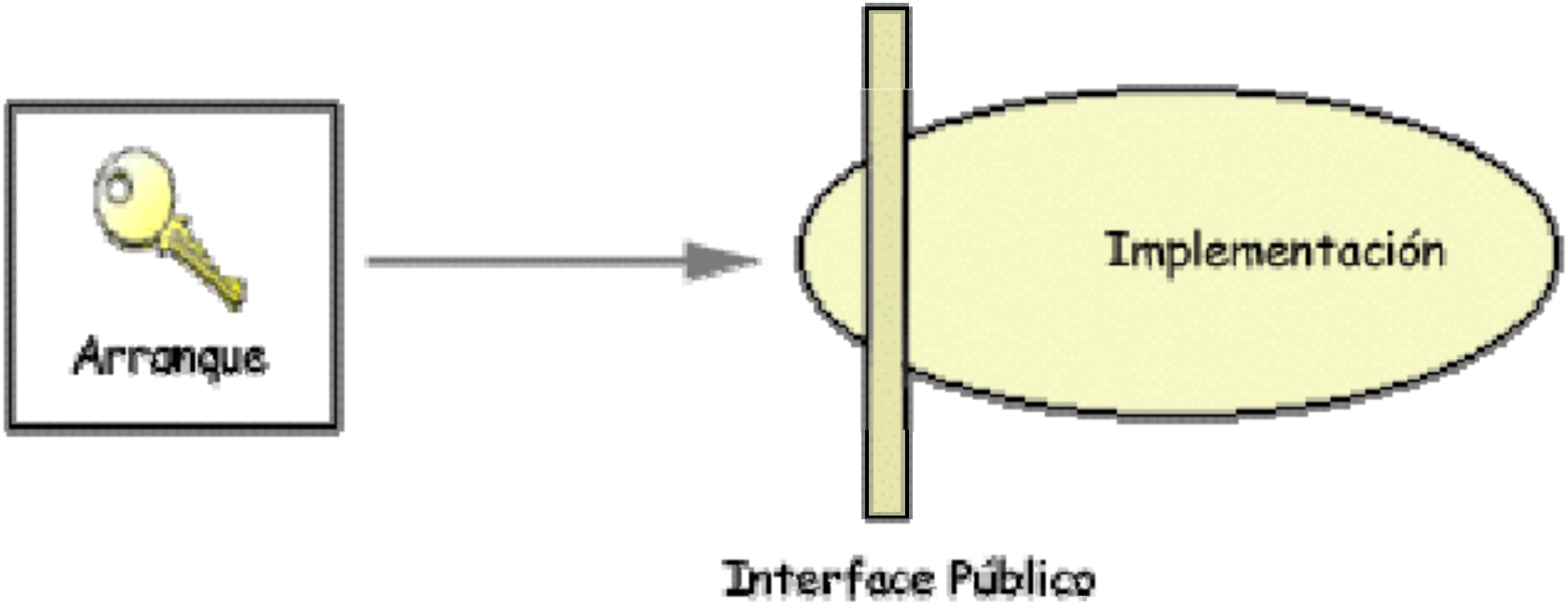


Objetos de la clase Coche

Clase Coche

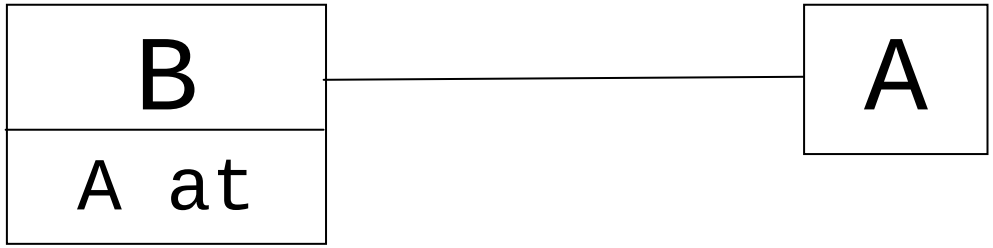

Coche
color
velocidad
tamaño
carburante
ir
parar
girarDerecha
girarIzquierda
arrancar

Objetos se comunican mediante paso de mensajes

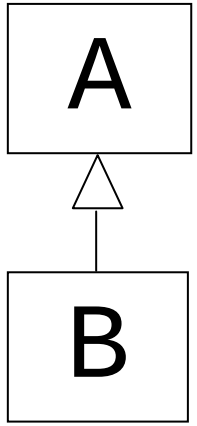


Relaciones entre objetos

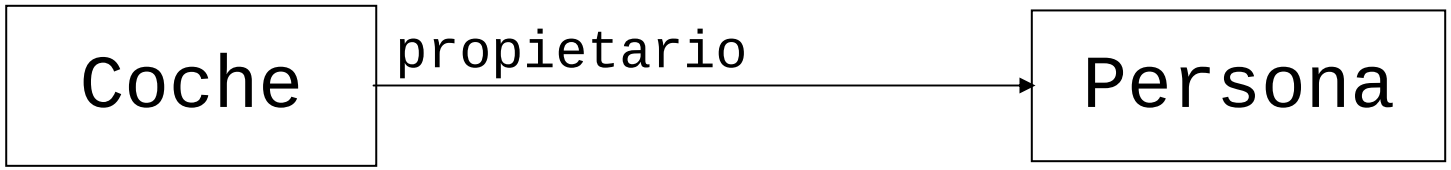
- B es un cliente de A si todo objeto de B puede contener información sobre uno o mas objetos de A



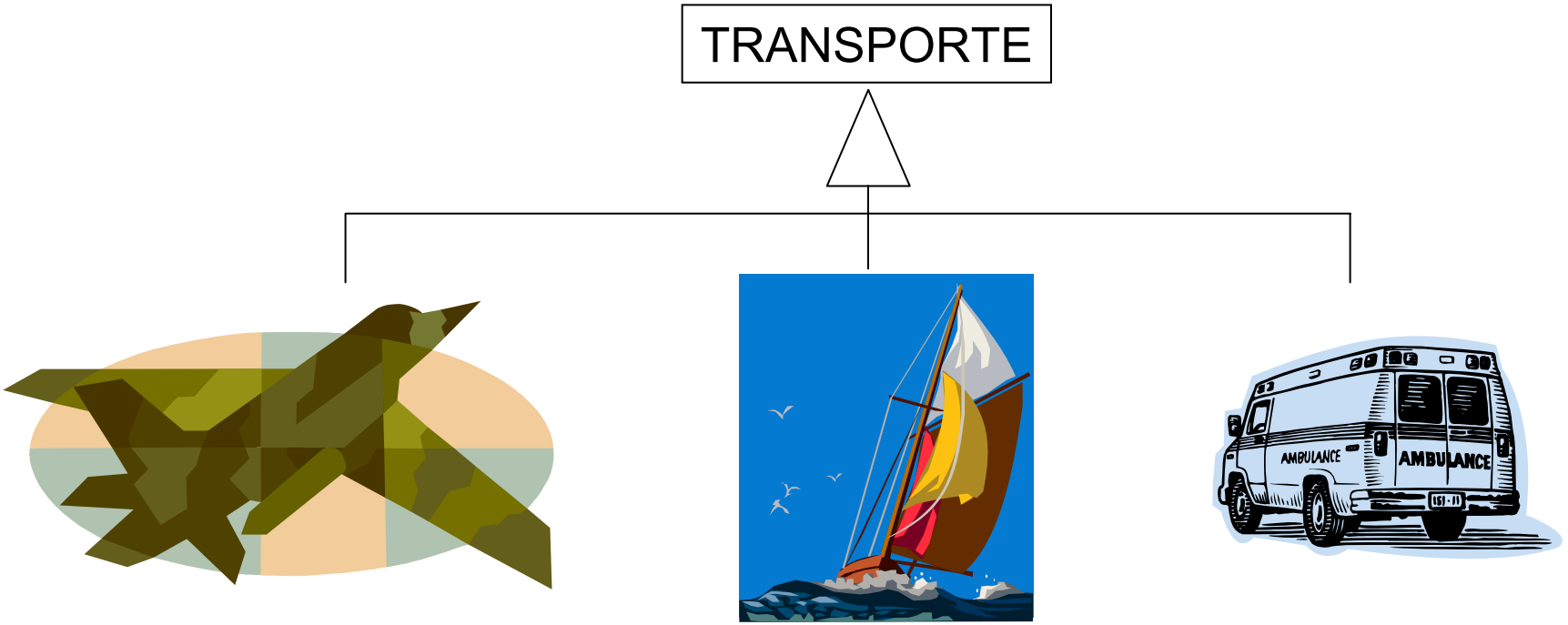
- B hereda de A si B denota una versión especializada de A



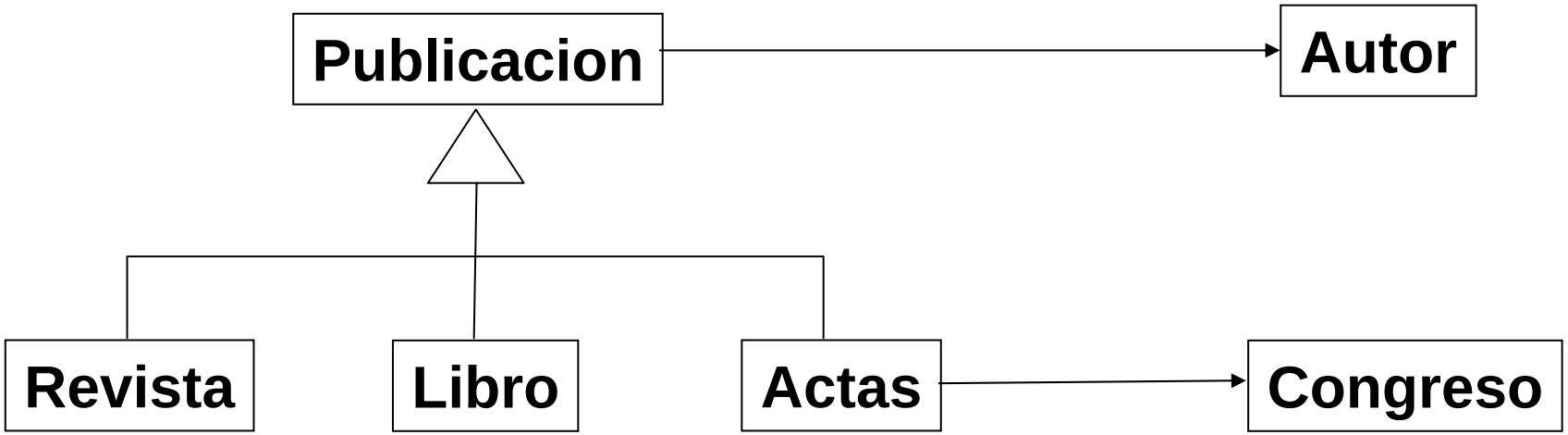
Relación de clientela

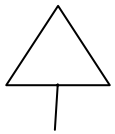


Relación de Herencia



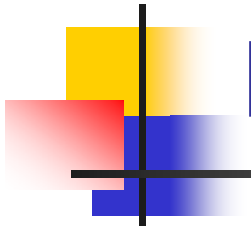
Estructura del software





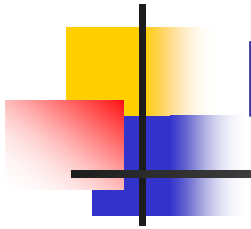
"*Libro* **es una especialización de** *Publicacion*"

"*Publicacion* **usa servicios de** *Autor*"



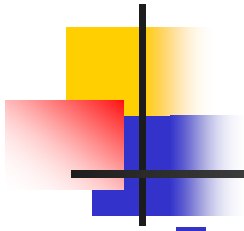
Historia de los lenguajes OO

- **SIMULA** (Dahl y Nygaard, 1964):
  - comienzo de la OO
  - Se identificaba como lenguaje de simulación
  - Ha influido en el desarrollo de otros LPOO
- **SMALLTALK** (Kay, Goldberg, Ingalls, 1972)
  - Simula + LISP (sin tipos) -> LPOO puro
  - Características de un LPOO [Byte81]:
    - Todo es un objeto.
    - Programa = cjto de objetos que se comunican mediante mensajes
    - Todo objeto es instancia de una clase (tiene un tipo).
    - La clase es el repositorio de comportamiento asociado con un objeto
    - Las clases se organizan en jerarquías de herencia



Historia de los LPOO–Años 80

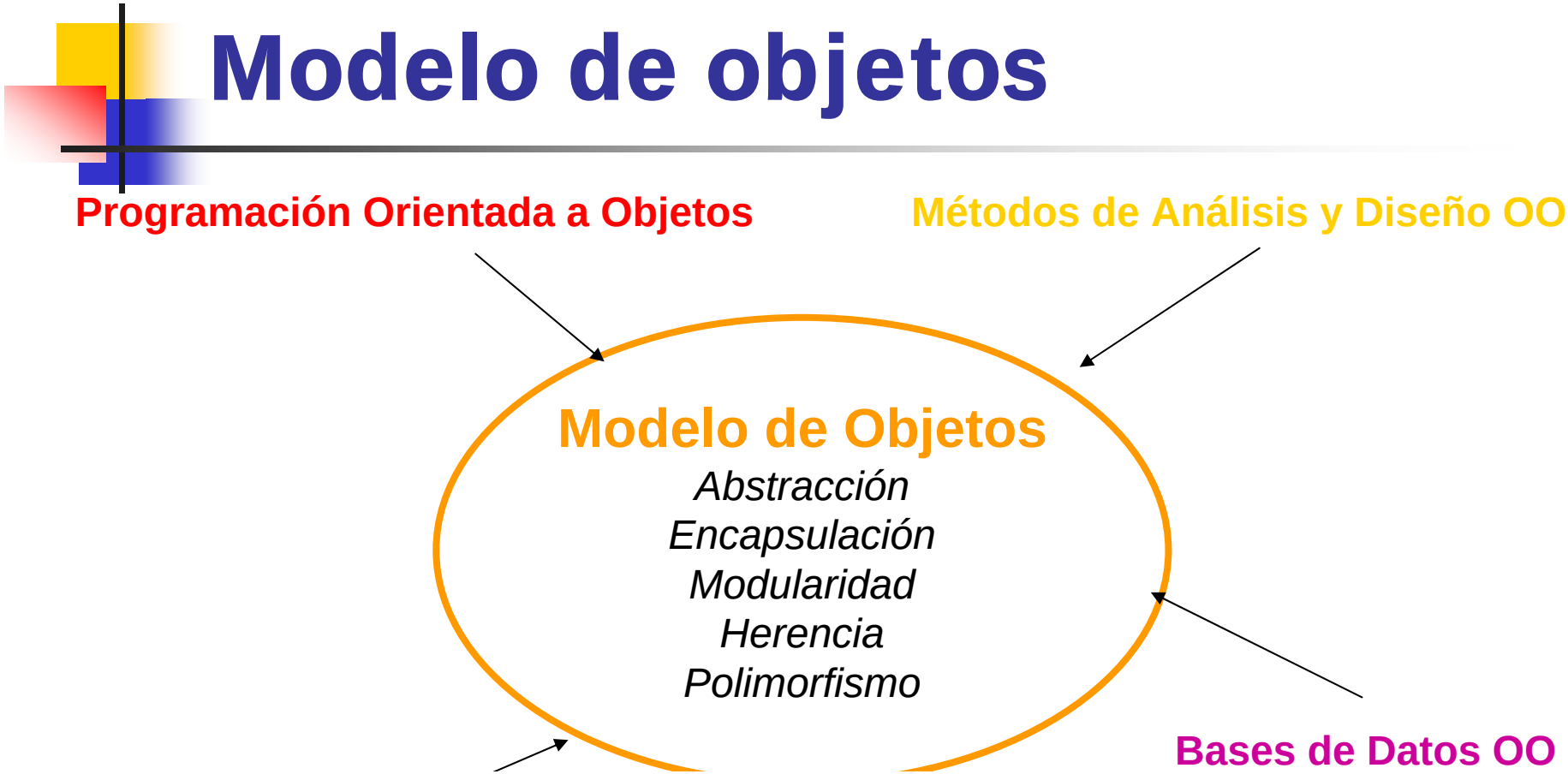
- Popularidad de los lenguajes OO
- **C++** (Stroustrup, 1985)
  - Lenguaje OO “híbrido”
    - Extensión de C con características OO
  - Muy popular ayudó a difundir la OO
- **Eiffel** (B. Meywe, 1985)
  - Lenguaje OO “puro”



Historia de los LPOO-Años 90

- Aplicaciones centradas en el web
- **JAVA** (SUN, 1995)
  - "comportamiento" en páginas HTML
- **C#** (Microsoft, 2000)
  - Plataforma .NET
  - Combina Java y C++
- Puntos fuertes de ambos lenguajes:
  - Máquina virtual -> portabilidad
  - Librerías de código -> reutilización

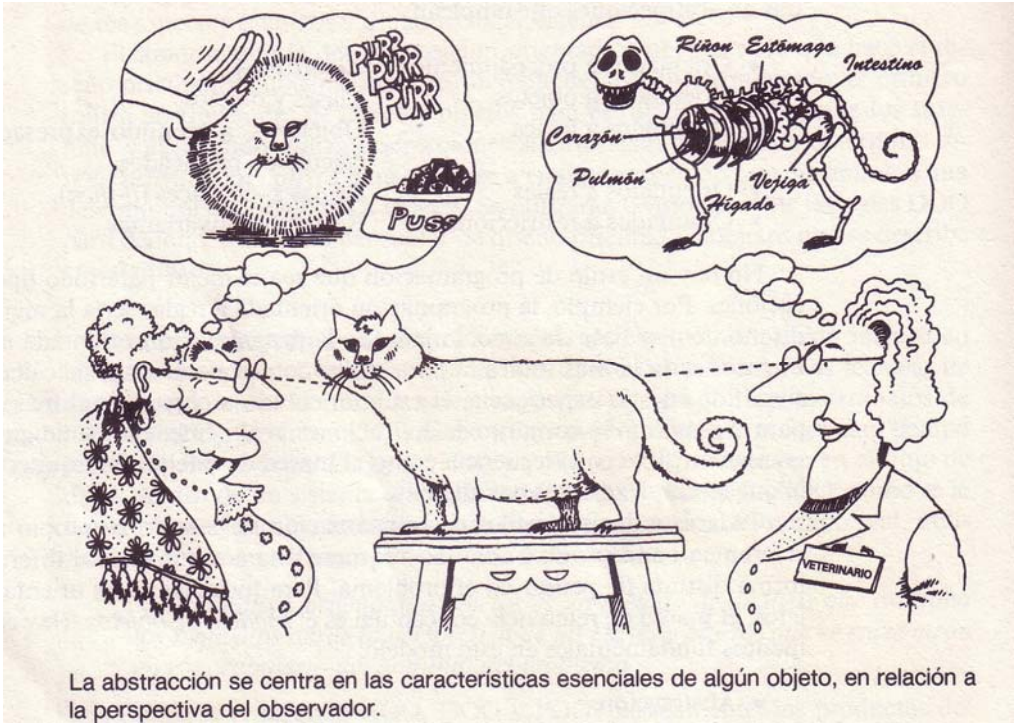
Modelo de objetos



Otros desarrollos se asientan en el paradigma OO

Desarrollo basado en componentes
Tecnología de objetos distribuidos

Abstracción



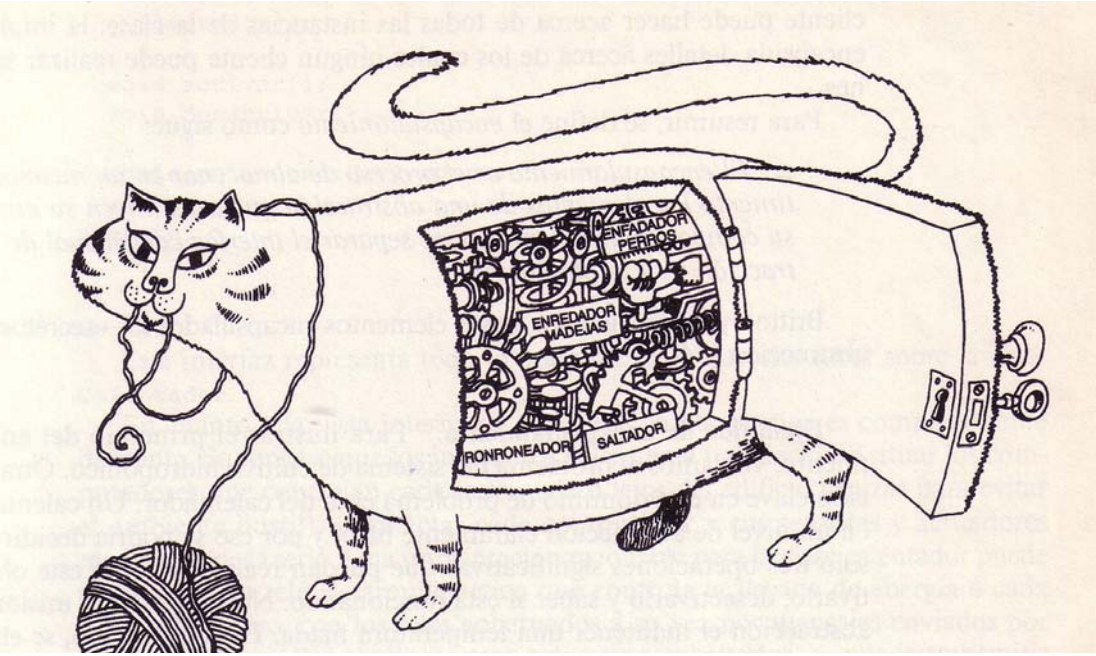
La abstracción se centra en las características esenciales de algún objeto, en relación a la perspectiva del observador.

- "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"
- Tipos de abstracción:
  - Datos, procedural
- Métodos de abstracción:
  - Parametrización
  - Especificación

[Booch'96] *Booch, G. Análisis y Diseño Orientado a Objetos con Aplicaciones. Addison-Wesley. 1996*

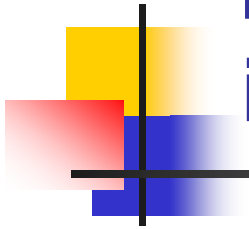
Encapsulación

- “Proceso de almacenar en un mismo compartimento los elementos de una abstracción que constituyen su estructura y su comportamiento”



El encapsulamiento oculta los detalles de la implementación de un objeto.

[Booch’96]



Encapsulación y Ocultación de la información

- Son cosas distintas
- Encapsulación es una facilidad del lenguaje
  - Agrupar estructura y comportamiento en una misma unidad sintáctica (módulo)
- Ocultación de Información
  - Principio de diseño modular
  - Organizar el contenido del módulo:
    - Parte pública (interfaz)
    - Parte privada (implementación)