Introducción a la Orientación a Objetos. Concepto de clase y objeto.

Conclusiones

- Conclusiones acerca de los casos de uso:
  - Recogen las necesidades del usuario de una forma clara y precisa.
  - Los analistas y diseñadores de software saben las necesidades que hay que satisfacer.
  - El impacto de un error de análisis es enorme, ya que afecta a todas las demás etapas de desarrollo.

Conclusiones

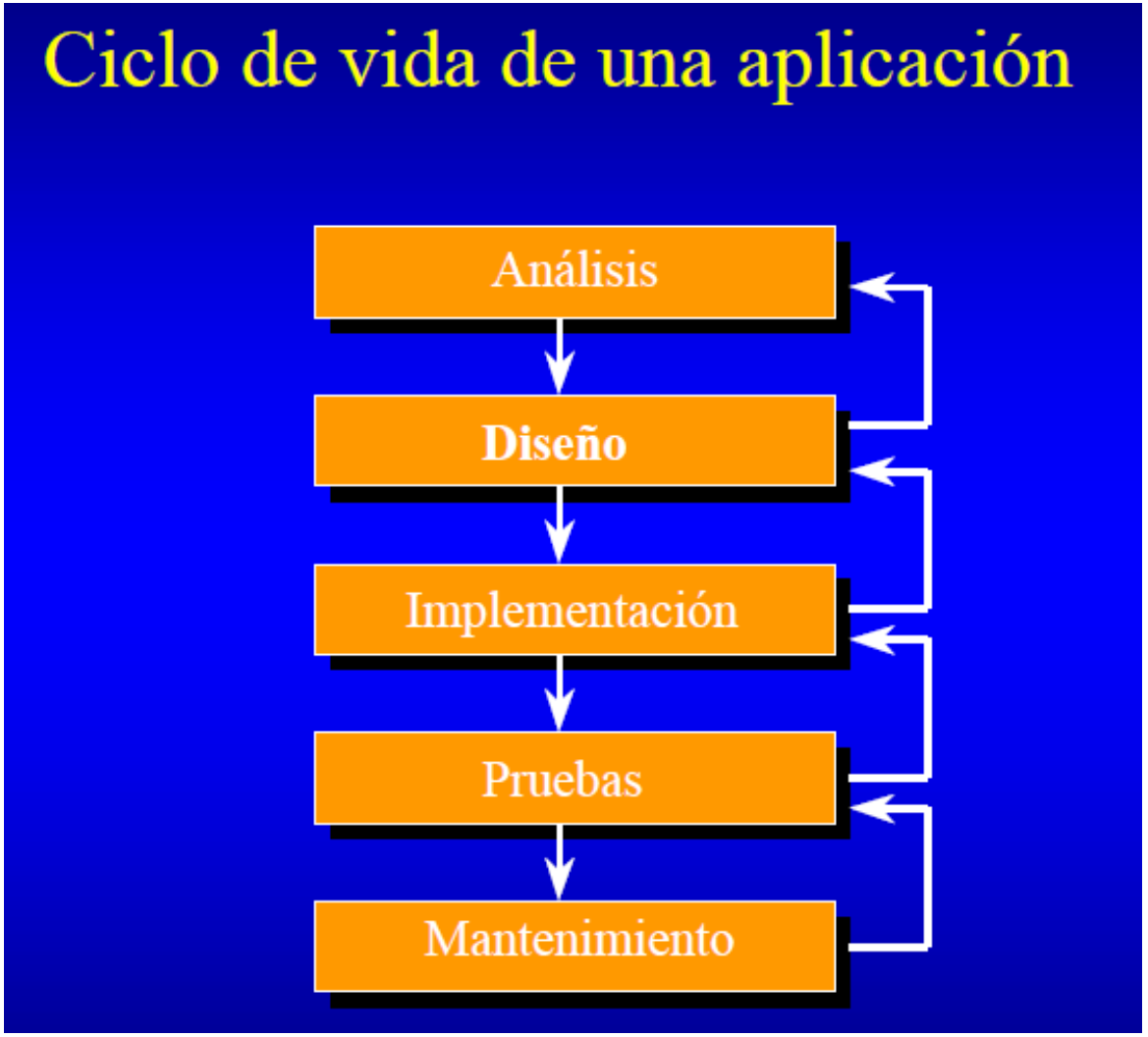
- Si existen malentendidos y el proyecto tiene problemas o fracasa nunca es culpa del cliente, es nuestra. Fracasar un proyecto puede suponer cerrar una empresa.

- Claros y concisos... siempre.

Los casos de uso son análisis pero ...

- Todo el proceso de desarrollo está guiado por los casos de uso.
- Está fuera del alcance del módulo, pero en la práctica se convierten en métodos de la fachada del modelo, a la que a su vez accede el controlador. Es decir, al final los **casos de uso tienen una implementación real**.
  - Para saber más:
  - http://corej2eepatterns.com/Patterns2ndEd/BusinessDelegate.htm

Ciclo de Vida



Paradigmas

Distintas formas de construcción de sistemas software:

- Programación estructurada.
  - C, Pascal, Modula.
- Programación funcional.
  - Lisp, Cammel.
- Programación lógica.
  - Prolog.
- Programación orientada a objetos.
  - Smalltalk, C++, C#, Java.

Paradigma de los objetos

- Un sistema software es un conjunto de objetos que cooperan.

  – Ej.: en un editor de gráficos: cuadrado, círculo, línea, imagen, barra de herramientas, etc.).

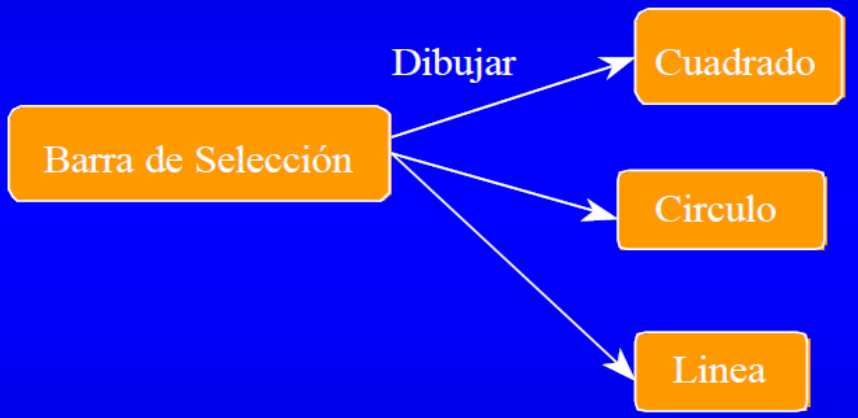
- Los objetos poseen una funcionalidad (operaciones que son capaces de hacer o mensajes a los que son capaces de reaccionar).

  – Ej.: Dibujar, CambiarTamanho, Mover, Eliminar, etc.

Paradigma de los objetos

- El diseño orientado a objetos es un paradigma opuesto a una visión algorítmica (descomposición funcional) del sistema implementar.

```
Proceso ()
{
    SubProceso1();
    …
    SubProcesoN();
}
```



Descomposición funcional vs orientación a objetos

Descomposición funcional	**Orientación a objetos**
• Módulos construidos alrededor de las **operaciones**. • Datos globales o distribuidos entre módulos. • Entrada/Proceso/Salida. • Organigramas de flujo de datos.	• Módulos construidos alrededor de las **clases**. • Clases débilmente acopladas, y sin datos globales • Encapsulación/mensajes. • Diagramas jerárquicos de clases.

Ventajas de la orientación a objetos

- Un diseño orientado a objetos maximiza la:
  - Modularidad y encapsulación.
    - El sistema se descompone en objetos con unas responsabilidades claramente especificadas.
- Extensibilidad

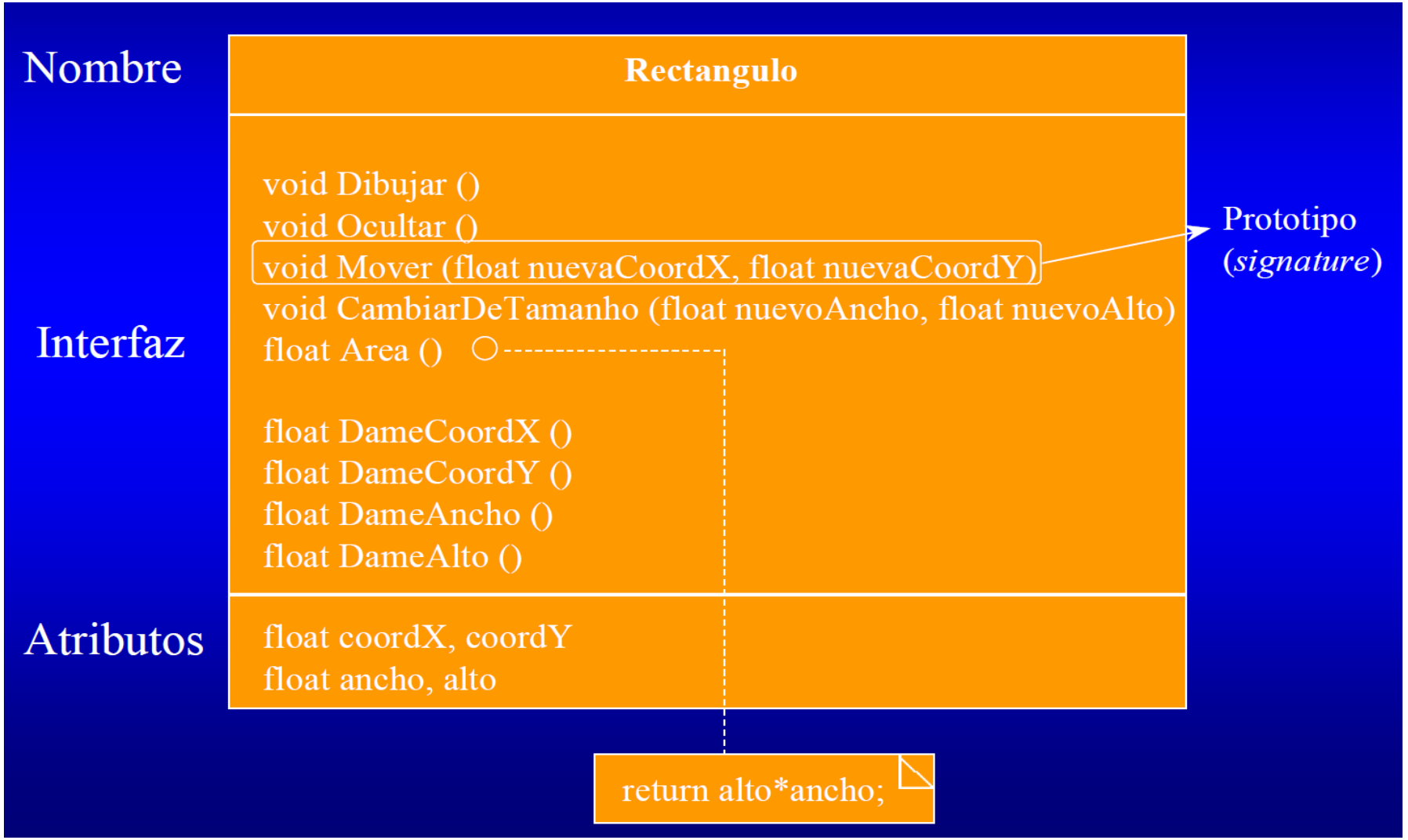
  - Posibilidad de ampliar la funcionalidad de la aplicación de manera sencilla.
- Reusabilidad

  - Posibilidad de reutilizar parte del código para el desarrollo de una aplicación similar.

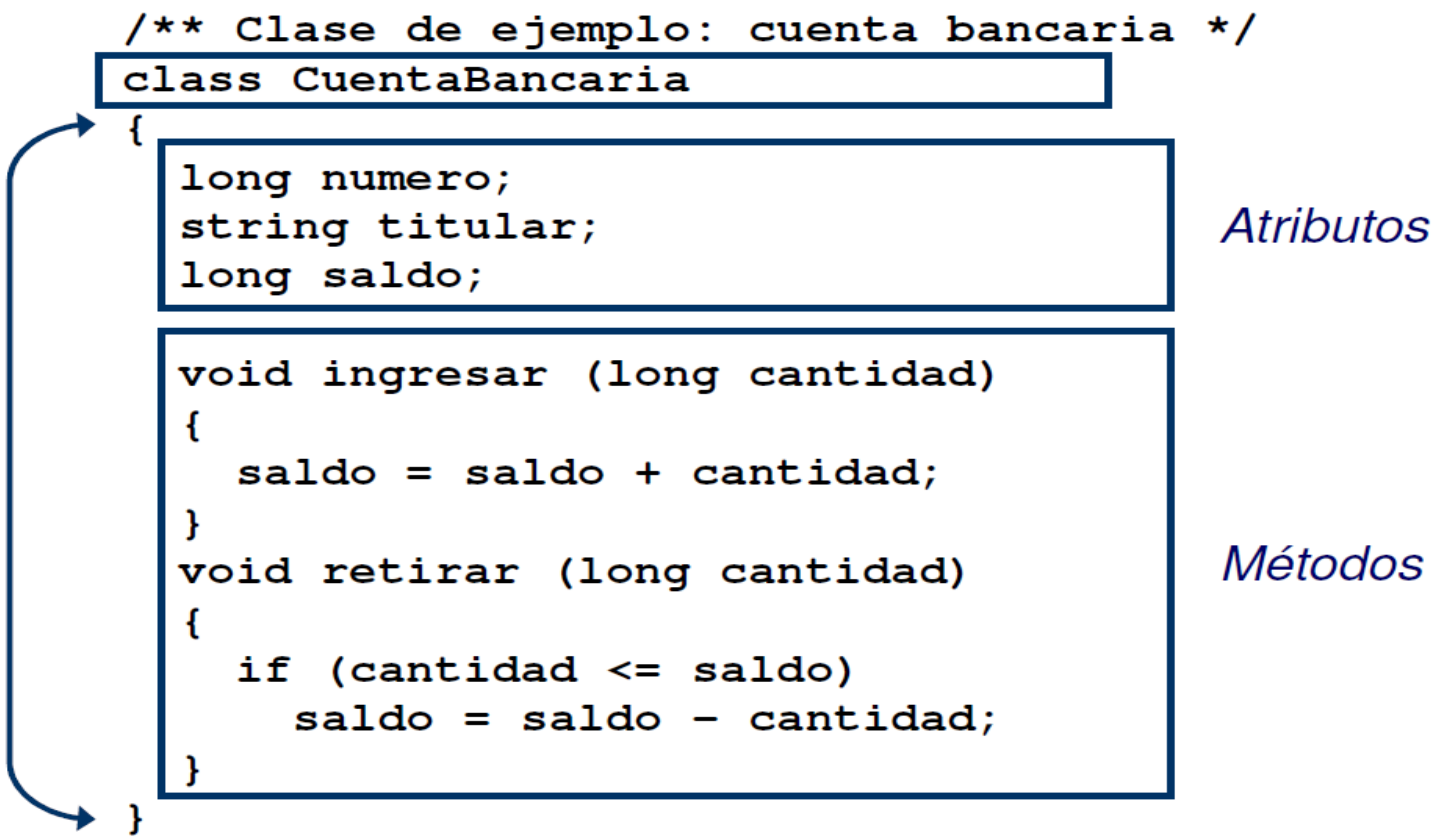
Concepto de clase

- Una clase describe un conjunto de ejemplares con propiedades y comportamientos similares.
- Una clase se describe por:
  - Nombre.
  - Interfaz: Operaciones (métodos, mensajes) que manipulan el estado.
  - Conjunto de atributos (datos) que definen el estado.
- Desde el punto de vista de la programación una clase es un tipo (ejs.:Rectangulo, Lista, Cola, NumeroComplejo, etc.).

Concepto de clase



Concepto de clase



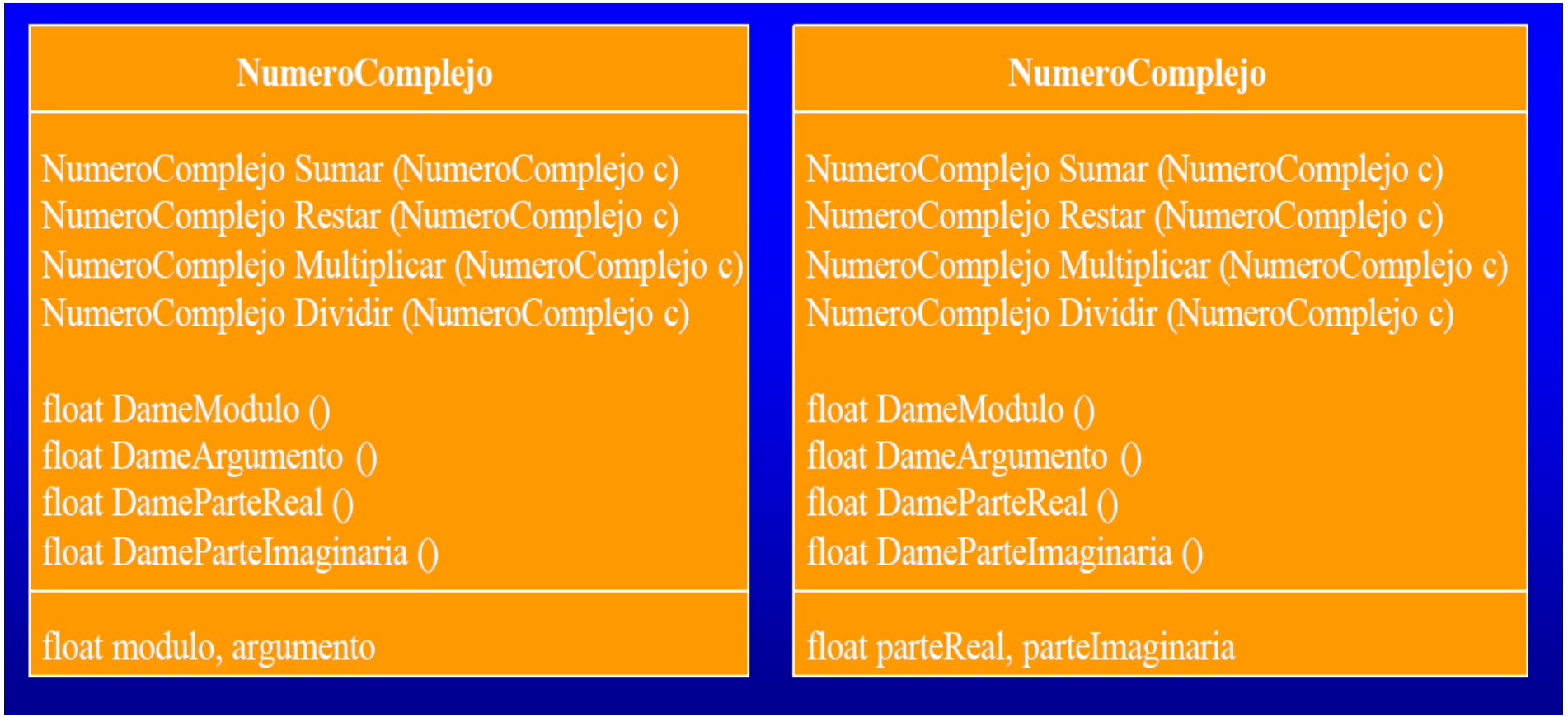
Concepto de objeto

- Objeto (instancia): cada uno de los ejemplares de una clase.

Encapsulación

- Los clientes de una clase sólo conocen la interfaz de la misma, es decir, conocen los prototipos de las operaciones pero no cómo están implementadas.
- Por tanto, si la implementación de una clase varía, y la interfaz continúa siendo la misma, no es necesario cambiar el código de los clientes.

Encapsulación



Visibilidad

- Los atributos y operaciones pueden tener los siguientes tipos de acceso (visibilidad):
  - Público
    - Se pueden acceder desde cualquier clase.
  - Privado
    - Sólo se pueden acceder desde operaciones de la clase.
  - Protegido
    - Sólo se pueden acceder desde operaciones de la clase o de clases derivadas.

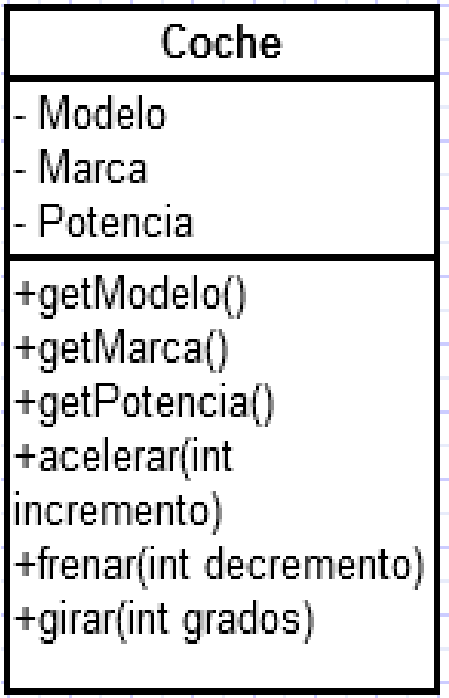
Visibilidad

– El estado debe ser privado.

– Las operaciones que definen la funcionalidad deben ser públicas.

– Las operaciones que ayudan a implementar parte de la funcionalidad deben ser privadas (si no se utilizan desde clases derivadas) o protegidas (si se utilizan desde clases derivadas).

Ejemplos


Coche
- Modelo
- Marca
- Potencia
+getModelo()
+getMarca()
+getPotencia()
+acelerar(int incremento)
+frenar(int decremento)
+girar(int grados)

Ejemplos


Aspirador
- int Potencia
+getPotencia()
+arrancar()
+recogerCable()
+setPotencia(int potencia)

Getters y setters

- Creamos un juego RPG en el que al atributo vida de un personaje se le hacen asignaciones en 5000 lineas de código. Nos damos cuenta de que debemos de validar que la vida se encuentre entre 0 y 100... tenemos 5000 cambios por hacer. Utilizando setters se realiza la validación en el setter: 1 sólo cambio.
- Mejora la encapsulación, desvelamos menos detalles de la implementación interna. Las clases tienen control sobre los cambios de estado que se realizan sobre sus objetos.

Getters y setters

- Los frameworks de gestión de persistencia (EJB, Hibernate) acceden a getters y setters automáticamente, no acceden a atributos.
- Eclipse genera getters y setters automáticamente (Botón derecho->Source->Generate Getters and Setters) … no cuesta nada.

Ejercicio

- Vamos a crear un juego en el que hay un personaje de un guerrero. El guerrero tiene un atributo vida, un atributo destreza y un atributo campoVision, y métodos para golpear, defenderse, así como getters y setters para sus atributos. Representar la clase Guerrero.
- Puedes crear otros atributos y métodos como resistencia o saltar, pero sin embargo verás que al guerrero no lo puedes por ejemplo dotar de arma porque … hay que relacionar clases.

Ejercicio

- Optativo:
  - Implementa tu guerrero en lenguaje Java.
  - Describe qué deficiencias crees que tiene tu modelo de cara a reutilizar el diseño y el código.