

# Análisis y diseño Orientado a Objetos

## Programación II LCC - UNCuyo

Javier J. Rosenstein

[Javier.rosenstein@ingenieria.uncuyo.edu.ar](mailto:Javier.rosenstein@ingenieria.uncuyo.edu.ar)

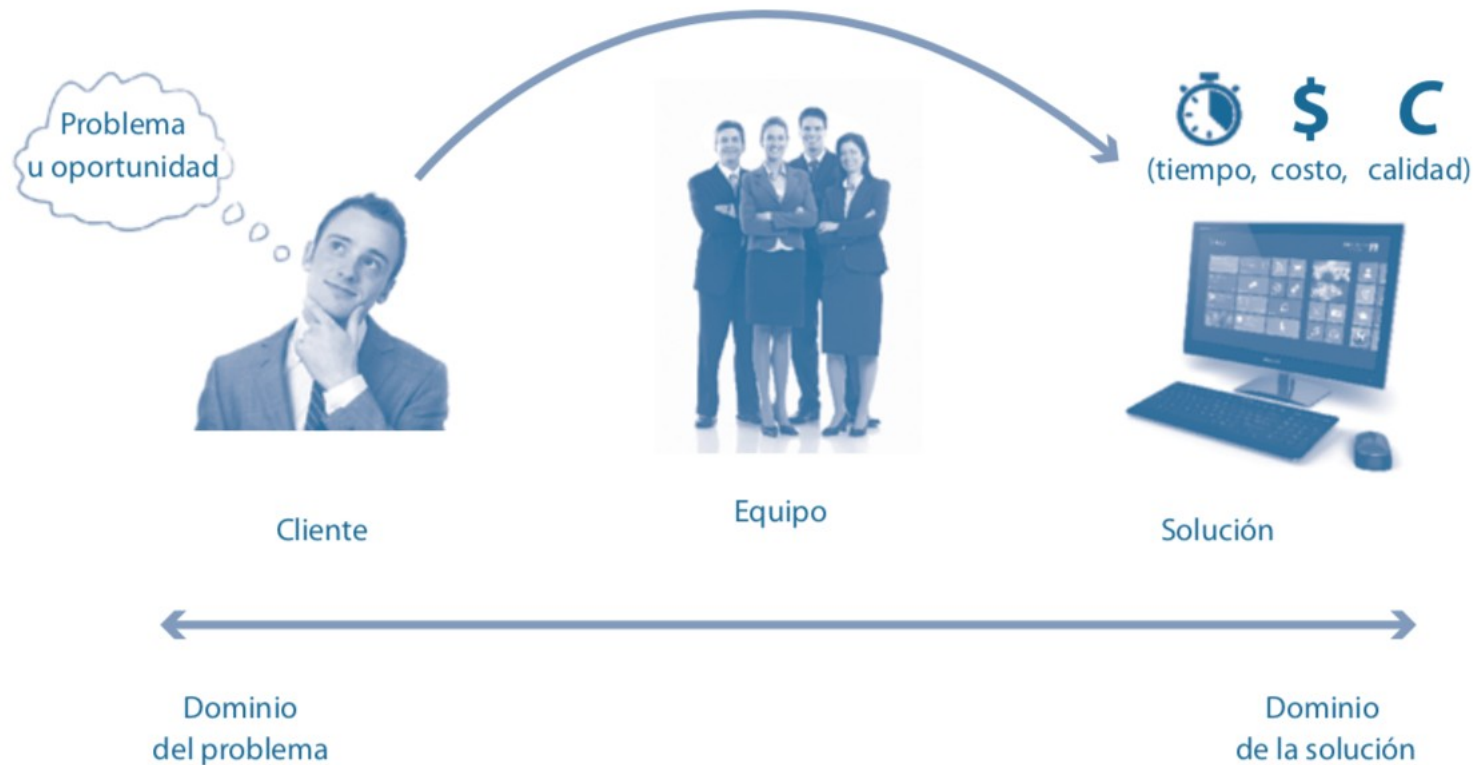
Daniel Fontana

[daniel.fontana@ingenieria.uncuyo.edu.ar](mailto:daniel.fontana@ingenieria.uncuyo.edu.ar)

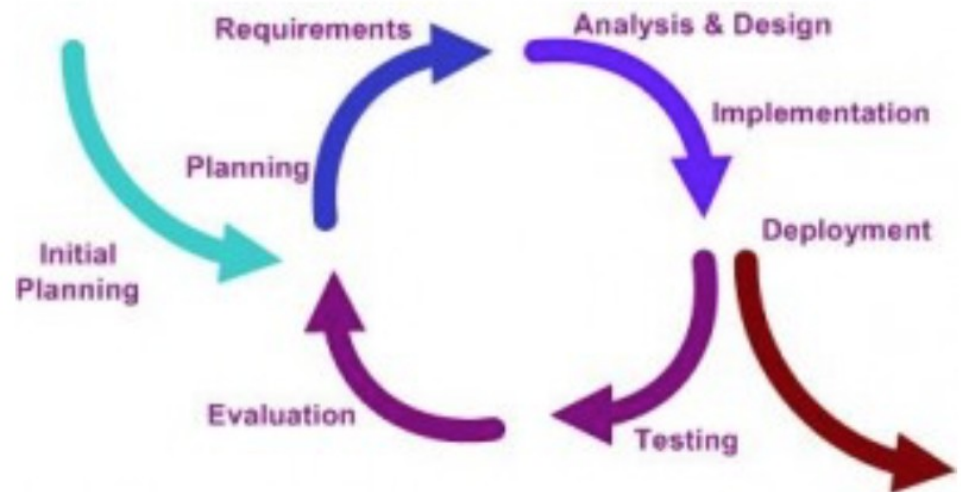
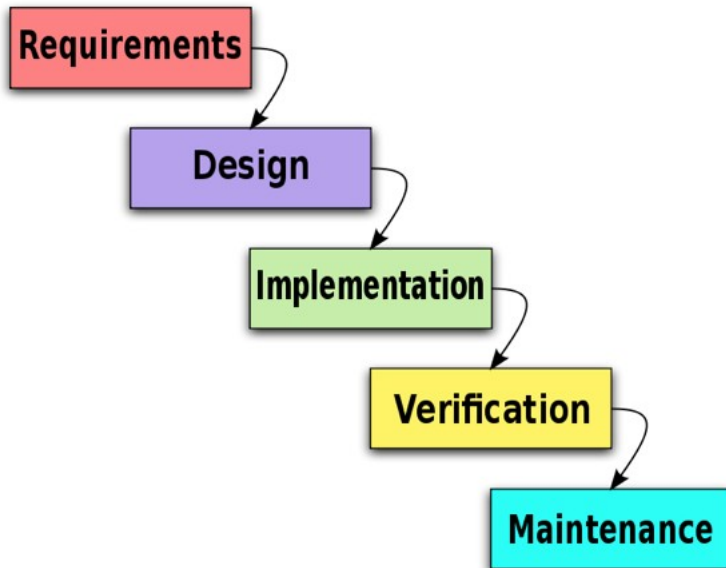
# Aprendizaje previo

- ✓ Ingeniería de Requisitos
- ✓ Ingeniería de Software I
- ✓ Ingeniería de Software II

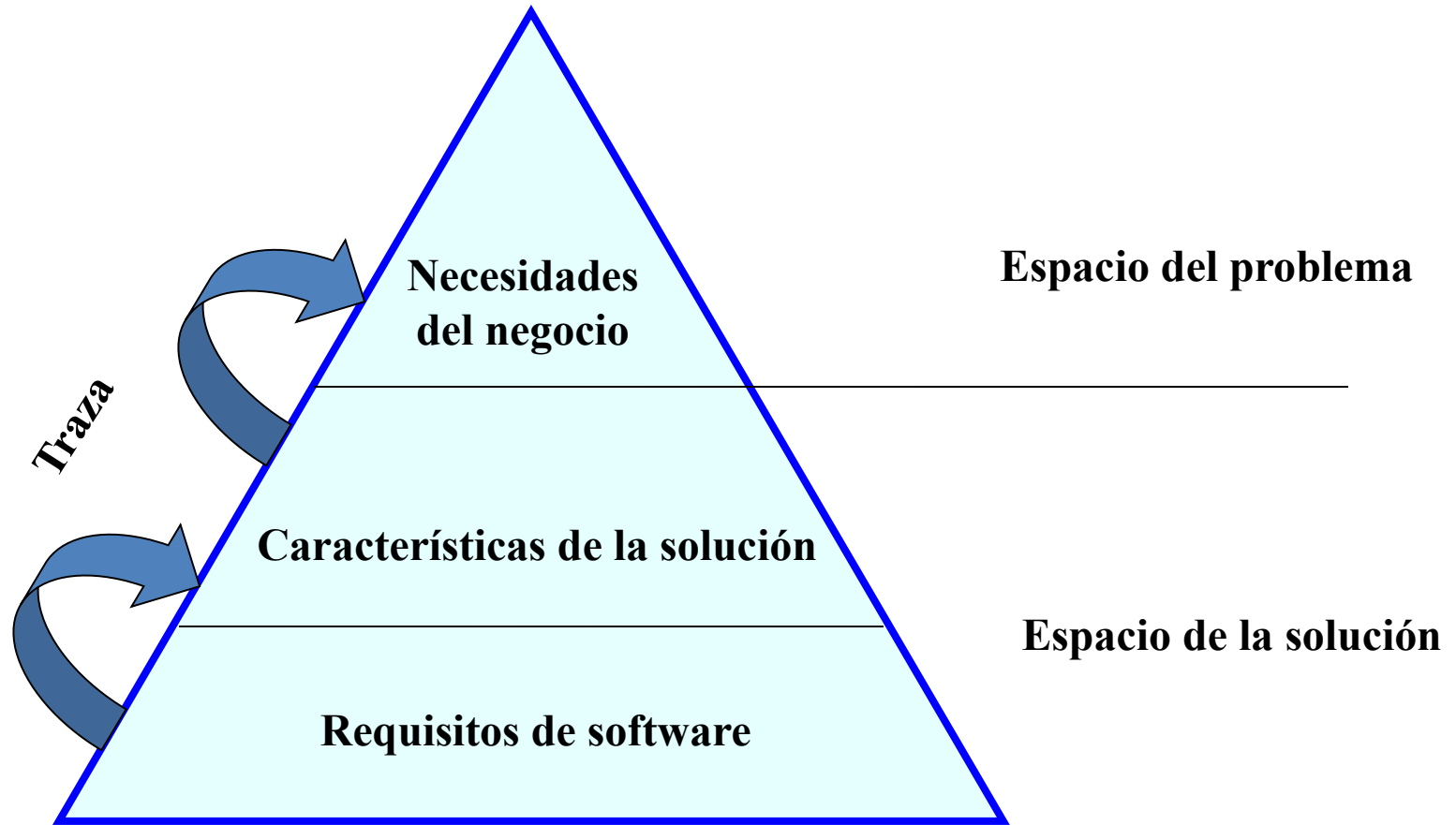
# Visión simplificada del desarrollo de sistemas de software



# Actividades claves del Desarrollo de sistemas software



# Entender el problema



# Desarrollo tradicional de SW

Descripciones operacionales

Requisitos funcionales de alto nivel

Sistemas legados



Arquitectura de sistema específico

Arquitectura del software

Diseño detallado

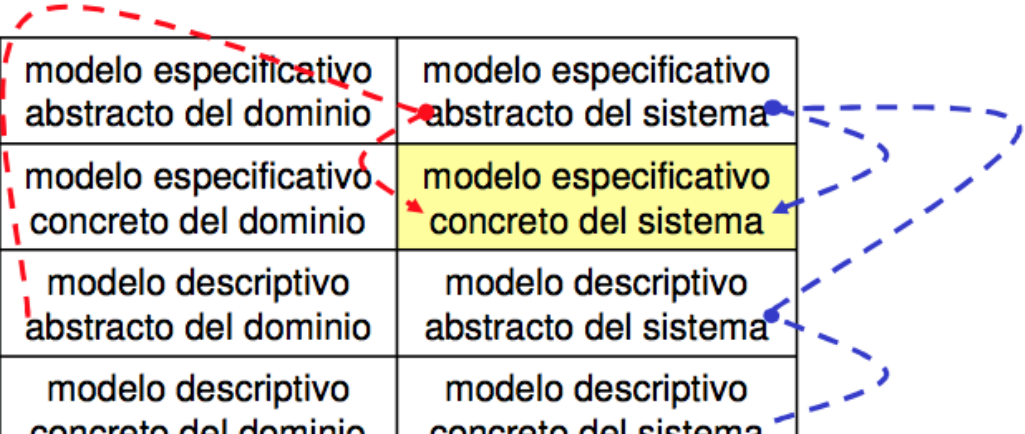
Implementación

- Los atributos de calidad rara vez se capturan como parte de la especificación de requisitos;
- generalmente son sólo vagamente comprendidos;
- frecuentemente pobremente articulados

# Transformación de modelos: del Análisis al Diseño

- Dos trayectorias típicas:
  - modelo del mundo real (“modelo de análisis” en un sentido)
    - análisis de requisitos (“modelo de análisis” en otro sentido)
    - modelo de diseño
  - modelo descriptivo concreto del sistema heredado
    - modelo descriptivo abstracto del sistema heredado
    - modelo especificativo abstracto del sistema nuevo
    - modelo especificativo concreto del sistema nuevo

Especificación	Vista abstracta	modelo especificativo abstracto del dominio	modelo especificativo abstracto del sistema
	Vista concreta	modelo especificativo concreto del dominio	modelo especificativo concreto del sistema
Descripción	Vista abstracta	modelo descriptivo abstracto del dominio	modelo descriptivo abstracto del sistema
	Vista concreta	modelo descriptivo concreto del dominio	modelo descriptivo concreto del sistema
		Dominio	Sistema



# ¿Qué son el Análisis y el Diseño?

- **ANÁLISIS** → investigación del problema (necesidades o requisitos)
  - En qué consiste
  - Qué debe hacerse
- **DISEÑO** → Solución lógica = ¿cómo el sistema cumple con los requisitos?
  - Descripción de alto nivel
  - Descripción detallada





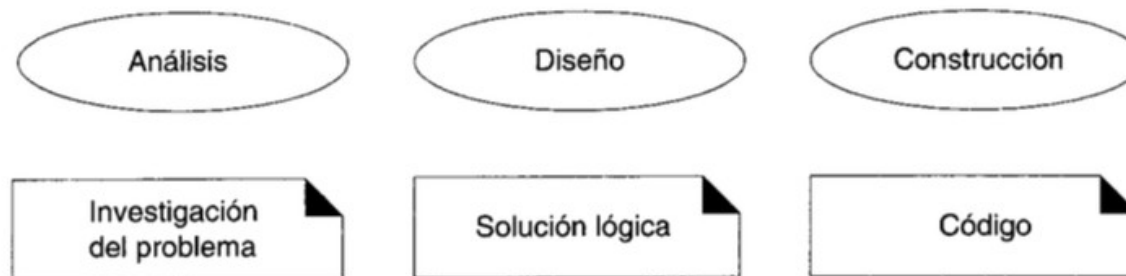
# ¿Qué son el Análisis y el Diseño OO?

La esencia del **análisis y el diseño orientado a objetos**, consiste en **situar** el dominio de un problema y su solución lógica dentro de una perspectiva (**paradigma**) de los **objetos** (cosas, conceptos, entidades).

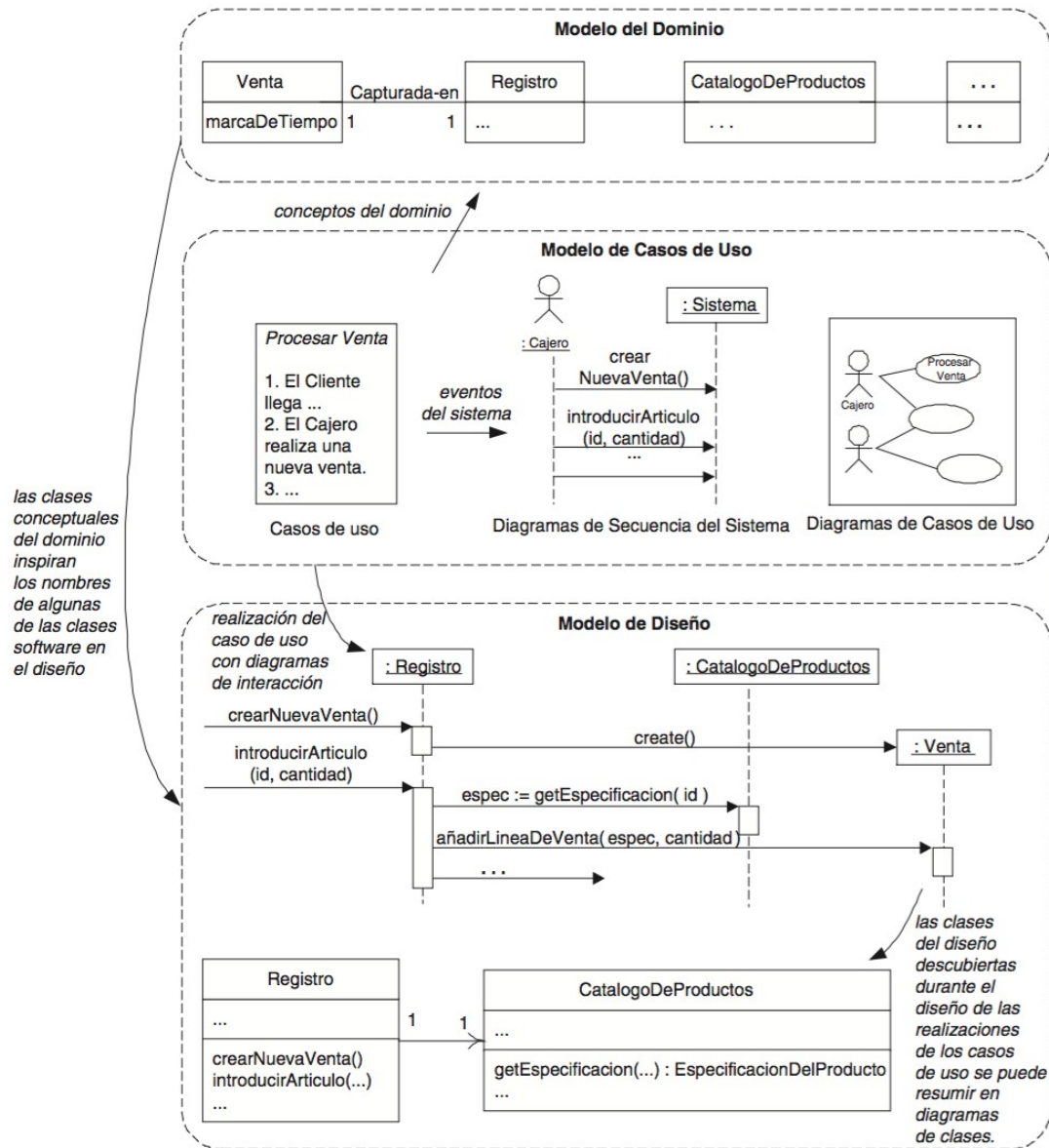
**ANÁLISIS OO** → Identificar y describir los objetos del dominio del problema

**DISEÑO OO** → Definir los objetos lógicos del software (atributos y métodos)

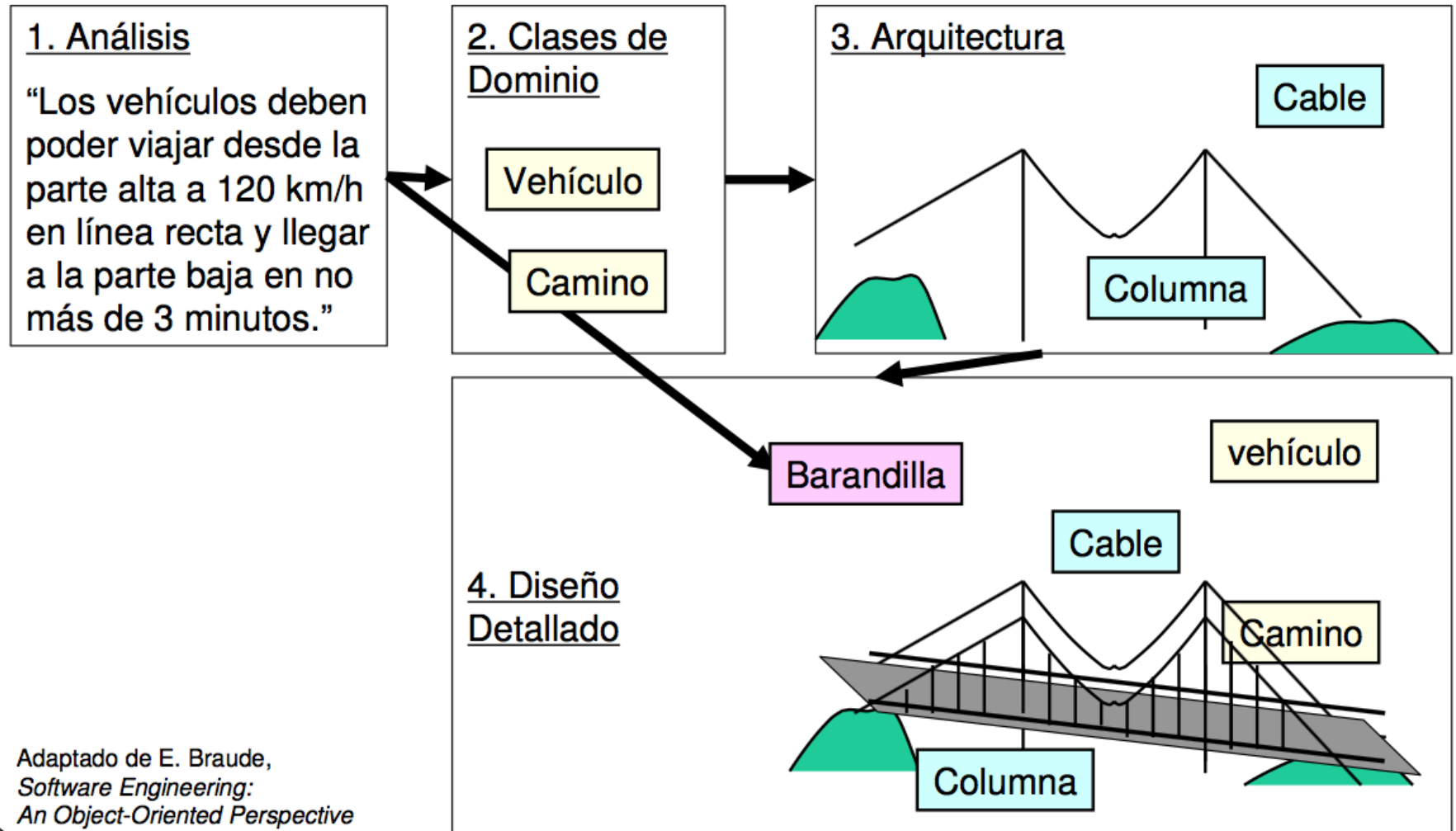
**CONSTRUCCIÓN OO** → Implementar los componentes del diseño mediante un lenguaje OO.



# Relación entre modelos de AyD OO

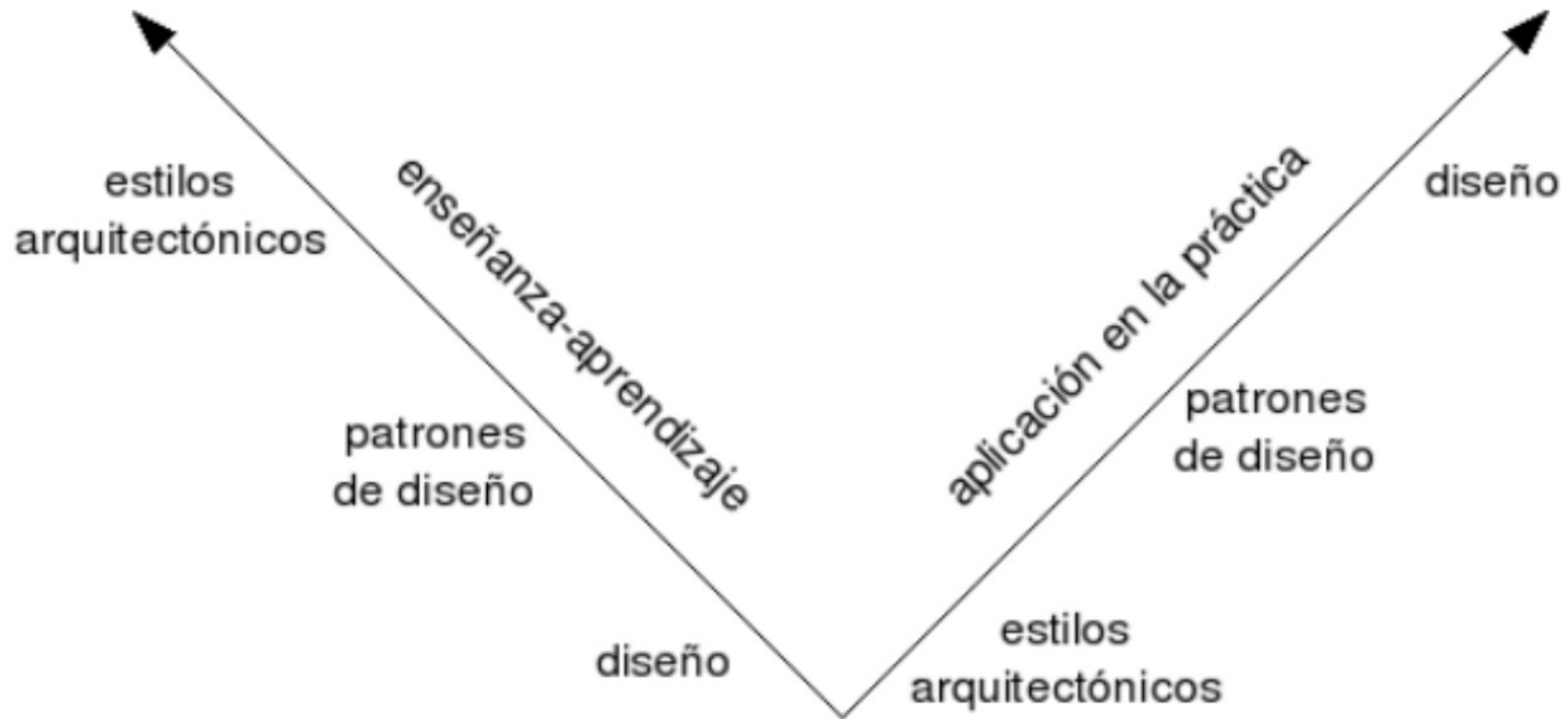


# Relación entre análisis, arquitectura y diseño detallado



Adaptado de E. Braude,  
*Software Engineering:  
An Object-Oriented Perspective*

# Diseño, Patrones y Arquitectura



# Orientación a Objetos - OO

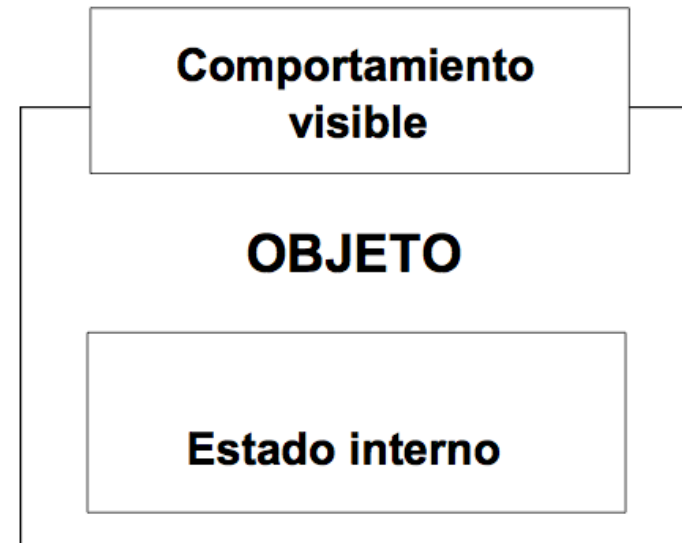
- **¿Por qué la Orientación a Objetos?**
  - **Proximidad** de los conceptos de **modelado** respecto de las entidades del **mundo real**
    - Mejora captura y validación de requisitos
    - Acerca el “espacio del problema” y el “espacio de la solución”
  - Modelado **integrado** de propiedades **estáticas y dinámicas** del ámbito del problema
    - Facilita construcción, mantenimiento y reutilización

# Objeto - conceptos

- La representación abstracta del objeto informático es una **imagen simplificada** del objeto del mundo real.
- Se acostumbra a considerar los objetos como seres animados con vida propia (nacen, viven y mueren).
- Un objeto puede caracterizar una **entidad física** (coche) o **abstracta** (ecuación matemática).

# Objeto en el paradigma OO

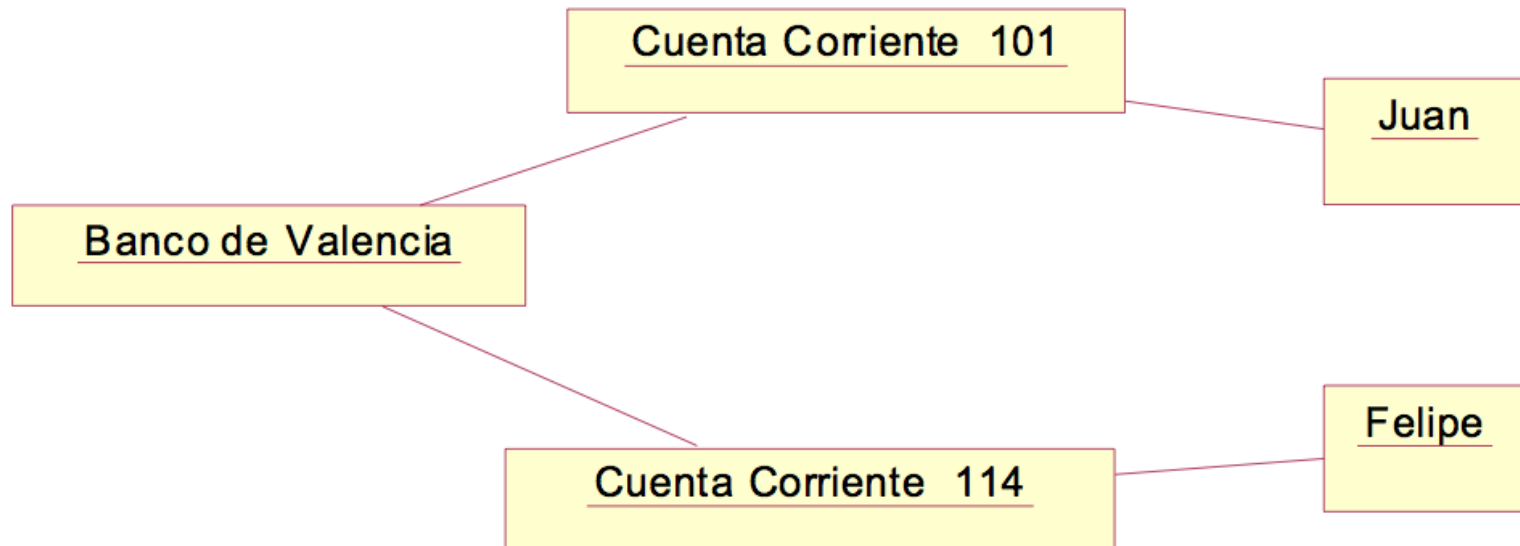
- Unidad atómica formada por la unión de estado y comportamiento.
  - Estado: valores de los atributos
  - Comportamiento: conjunto de servicios asociados
- La **encapsulación** proporciona una cohesión interna fuerte y un acoplamiento externo débil.
- Para manipular los objetos se utilizan **mensajes**.



**Objeto = Estado + Comportamiento + Identidad**

# Objeto - representación

- En UML, un objeto se representa por un rectángulo con un nombre subrayado





# Objeto – Programación OO

Por ejemplo en JAVA:

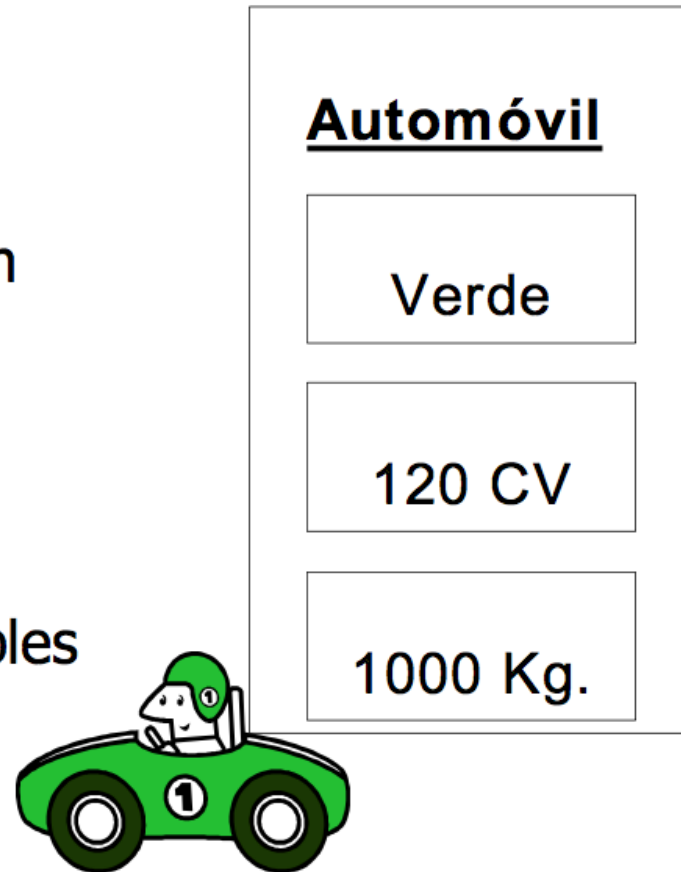
**1. Persona objeto = new Persona();**

El operador new crea un objeto de la clase Persona.

En el lado izquierdo definimos la variable objeto del tipo Persona y del lado derecho instanciamos la clase Persona con new y la asignamos a dicha instancia a la variable objeto.

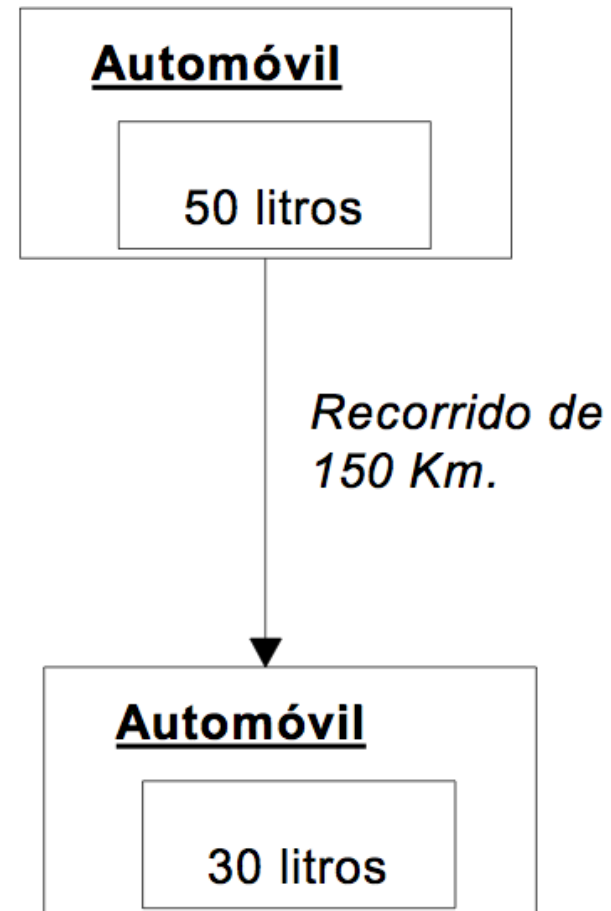
# Objeto - Estado

- El **estado** contiene los valores de sus atributos (variables) cuya información cualifica al objeto.
- Un atributo toma un valor en un dominio concreto
- El estado en un instante dado corresponde a una selección de valores de entre todos los posibles en cada atributo.

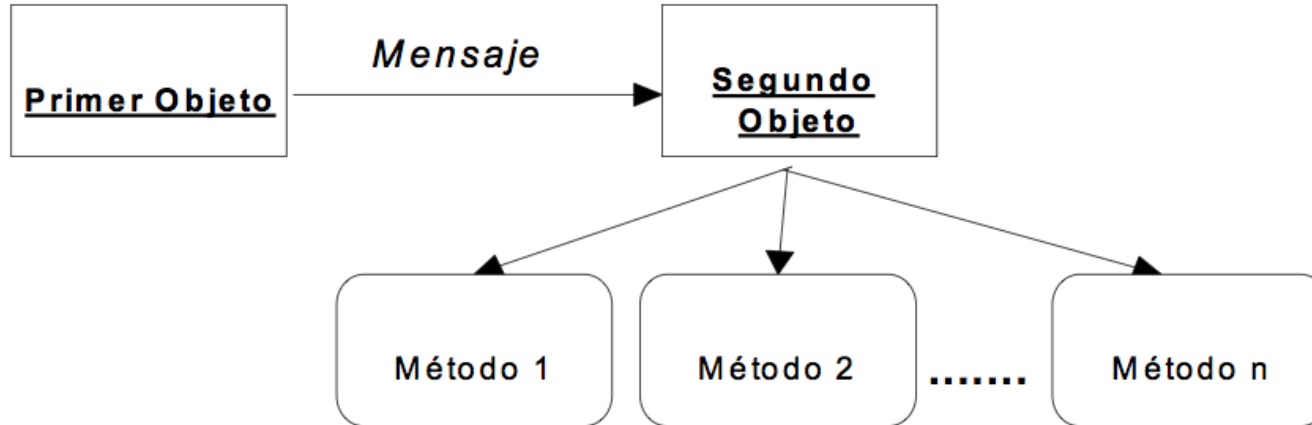


# Objeto – Estado - 2

- El estado evoluciona con el tiempo.
- Hay componentes constantes (marca del automóvil, potencia, etc.).
- Generalmente el estado de un objeto es variable.



# Objeto - Comportamiento



- Describe las *acciones* y *reacciones* de los objetos.
- Cada **operación / método** es un átomo de comportamiento.
- Las operaciones se desencadenan por estímulos externos (*mensajes*), enviados por otros objetos.

# Objeto – Comunicación

- Un sistema OO puede verse como un conjunto de objetos autónomos y concurrentes que trabajan de manera coordinada en la consecución de un fin específico.
- El comportamiento global se basa pues en la **comunicación** entre los objetos que lo componen.

# Objeto - Estímulo

- Un **estímulo** (evento) causará la invocación de una operación, la creación o destrucción de un objeto o la aparición de una señal.
- Un **mensaje** es la especificación de un **estímulo**.
- Tipos de **flujo de control** en mensajes:
  - Llamada a procedimiento o flujo de control anidado —————→
    - Síncrono
  - Flujo de control plano —————→
    - Asíncrono
  - Retorno de una llamada a procedimiento - - - - ->

# Objeto - Identidad

- Cada objeto posee un **oid**, que establece la **identidad** del objeto y tiene las siguientes características:
  - Constituye un **identificador único y global** para cada objeto dentro del sistema.
  - Es determinado en el momento de la **creación** del objeto.
  - Es **independiente** de la **localización física** del objeto, es decir, provee completa independencia de localización.
  - Es independiente de las propiedades del objeto, lo cual implica **independencia de valor y de estructura**.
    - Dos objetos se pueden distinguir a pesar de tener atributos idénticos.
  - **No cambia** durante toda la vida del objeto. Además, un oid no se reutiliza aunque el objeto deje de existir.
  - No se tiene ningún control sobre los oids y su manipulación resulta transparente.
  - No se representa en el modelado de manera específica.

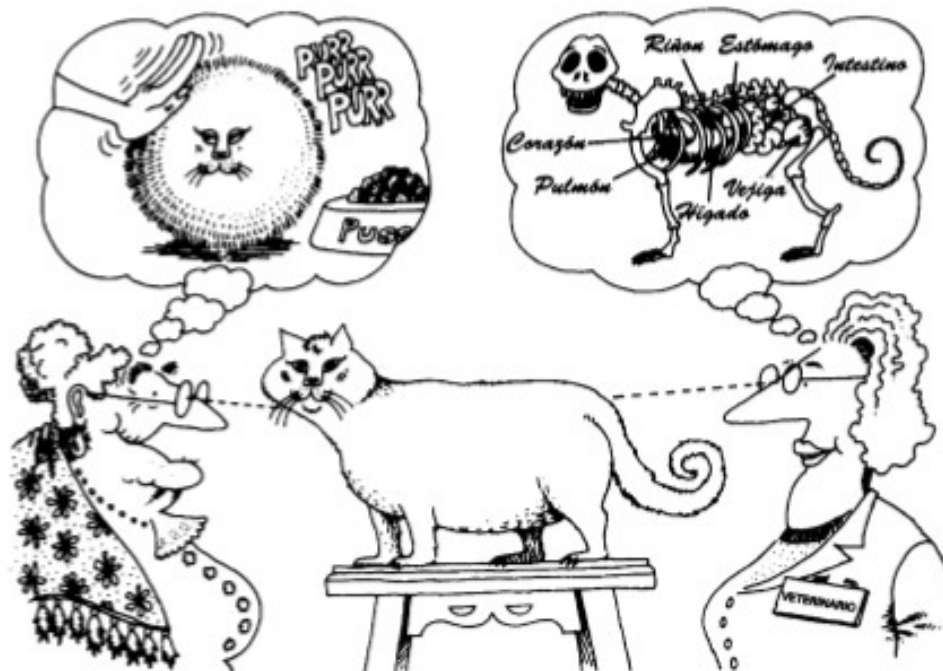
# Clase

- El mundo real puede ser visto desde **abstracciones** diferentes (subjetividad)
- Mecanismos de abstracción:
  - **Clasificación** / Instanciación
  - Composición / Descomposición
  - Agrupación / Individualización
  - Especialización / Generalización
- La clasificación es uno de los mecanismos de abstracción más utilizados



# Abstracción

*Una abstracción denota las características esenciales de un objeto que lo distinguen de todos los demás tipos de objeto y proporciona así fronteras conceptuales nítidamente definidas respecto a la perspectiva del observador*  
 [Booch 94]

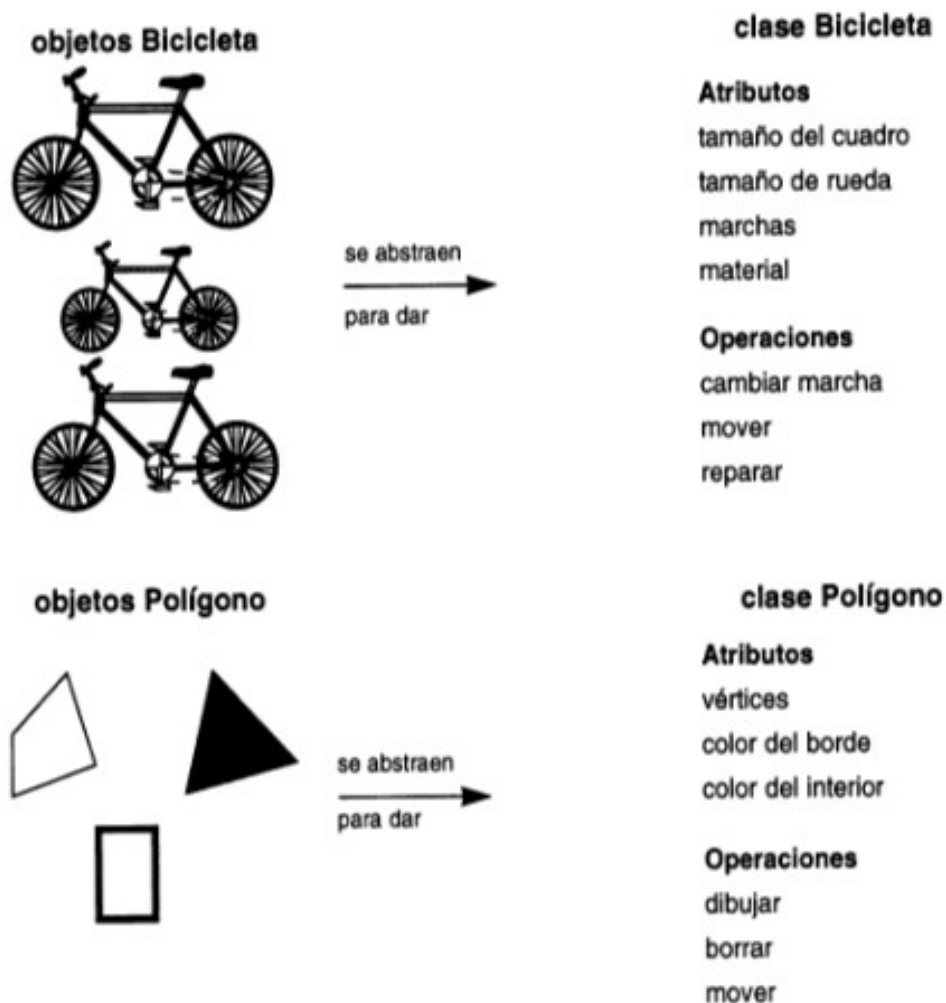


La abstracción se centra en las características esenciales de algún objeto, en relación a la perspectiva del observador.

# Clase – Concepto

- La **clase** define el ámbito de definición de un conjunto de objetos
- Cada **objeto** pertenece a una clase
- Los objetos se crean por **instanciación** de las clases

# Clases y objetos



# Clase - Definiciones

- Otras definiciones
  - **Plantilla** a partir de la que se crean objetos. Contiene una definición del estado y los métodos del objeto.
  - **Módulo** software que encapsula atributos, operaciones, excepciones y mensajes.
  - **Conjunto de objetos** que comparte una estructura y comportamiento comunes
  - **Instrumentación** que puede ser instanciada para crear múltiples objetos que tienen el mismo comportamiento inicial

# Clase - Ejemplo

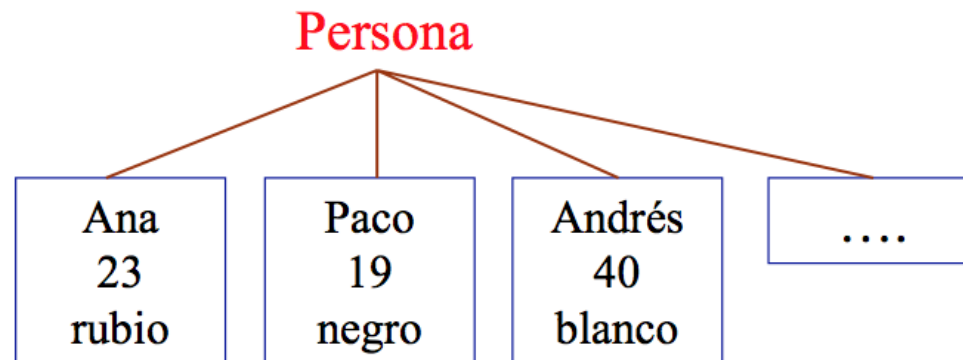
- La clase "Persona"

- Atributos

- Nombre: string
- Fecha de nacimiento: fecha
- Color del pelo: (negro, blanco, pelirrojo, rubio)

- Métodos

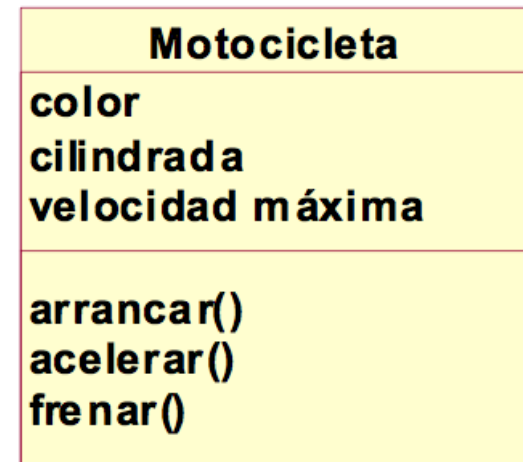
- Nacer
- Crecer
- Morir



Objetos de la Clase Persona

# Clase - Representación

- En UML cada clase se representa en un rectángulo con tres compartimentos:
  - nombre de la clase
  - atributos de la clase
  - operaciones de la clase



# Clase – Programación OO

Por ejemplo en JAVA:

**1. Public class Persona { }**

Entre las llaves se escribe la definición completa de la clase (cuerpo), como ser las propiedades (atributos) y los métodos (operaciones).

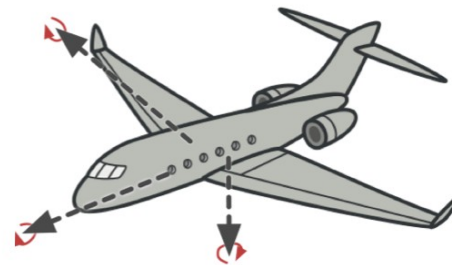
# Pilares de la POO



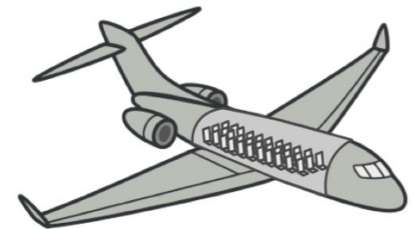


# Abstracción

La *Abstracción* es el modelo de un objeto o fenómeno del mundo real (o imaginario), limitado a un contexto específico, que representa todos los datos relevantes a este contexto con gran precisión, omitiendo el resto.



Airplane
- speed - altitude - rollAngle - pitchAngle - yawAngle
+ fly()



Airplane
- seats
+ reserveSeat(n)

*Distintos modelos del mismo objeto del mundo real.*

# Abstracción – Programación OO

La abstracción nos permite tratar a los objetos de la forma más básica posible abstrayendo todas aquellas características que no son relevantes en el momento. Para ello utilizamos las clases bases para manipularlos en lugar de la clase concreta. Analicemos el siguiente ejemplo:

```
Mamifero animal1 = new Perro();  
animal1.comunicarse();
```

```
Mamifero animal2 = new Murcielago();  
animal2.comunicarse();
```

```
Mamifero animal = getAlgunMamifero();  
animal.comunicarse();
```

De esta última manera tratamos al verdadero mamífero de la forma más simple conocida: como un mamífero. A esta característica se le conoce como Abstracción.

# Encapsulación

La *encapsulación* es la capacidad que tiene un objeto de esconder partes de su estado y comportamiento de otros objetos, exponiendo únicamente una interfaz limitada al resto del programa.

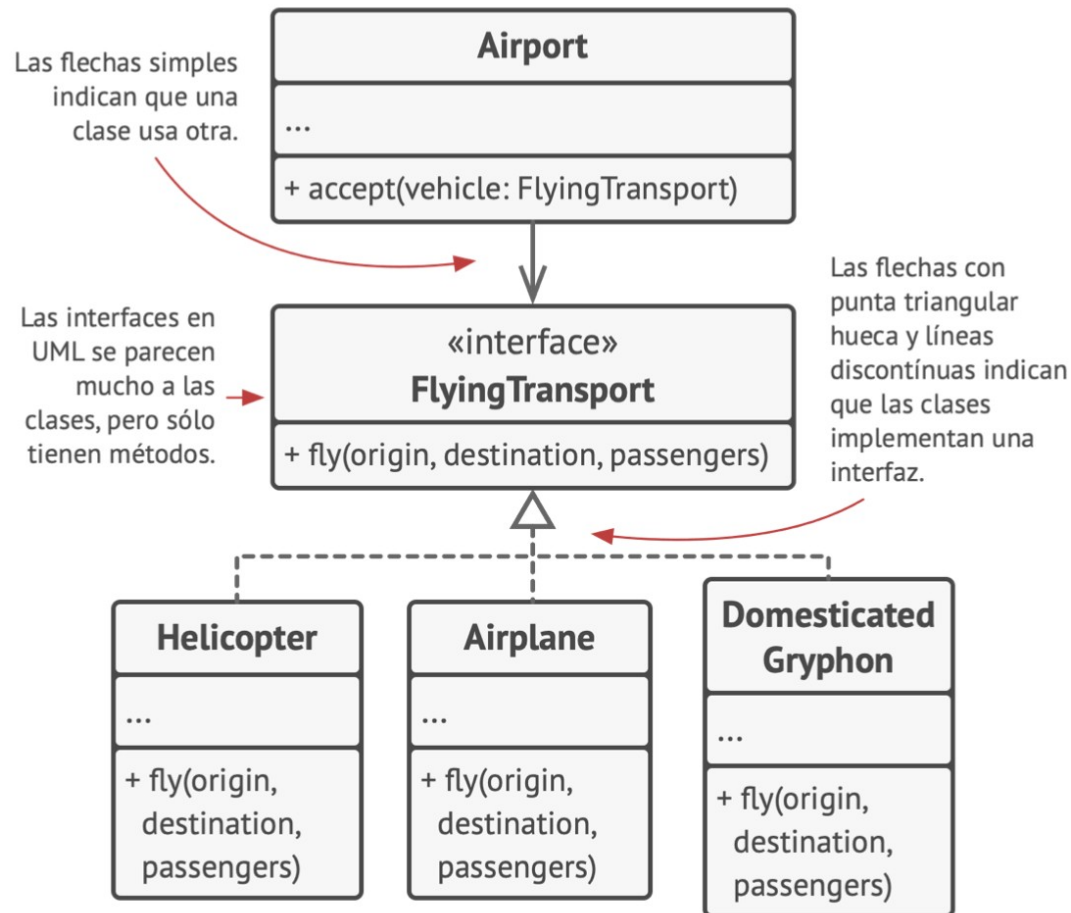
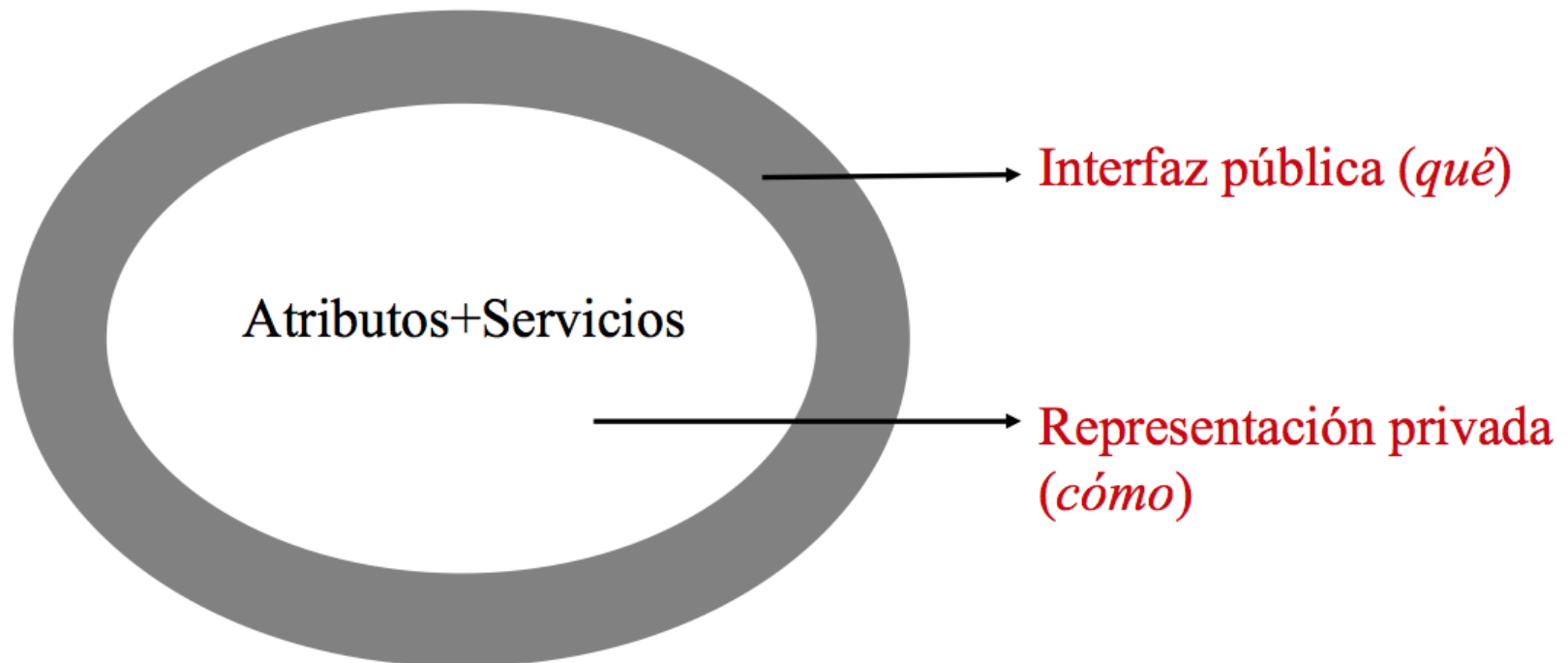


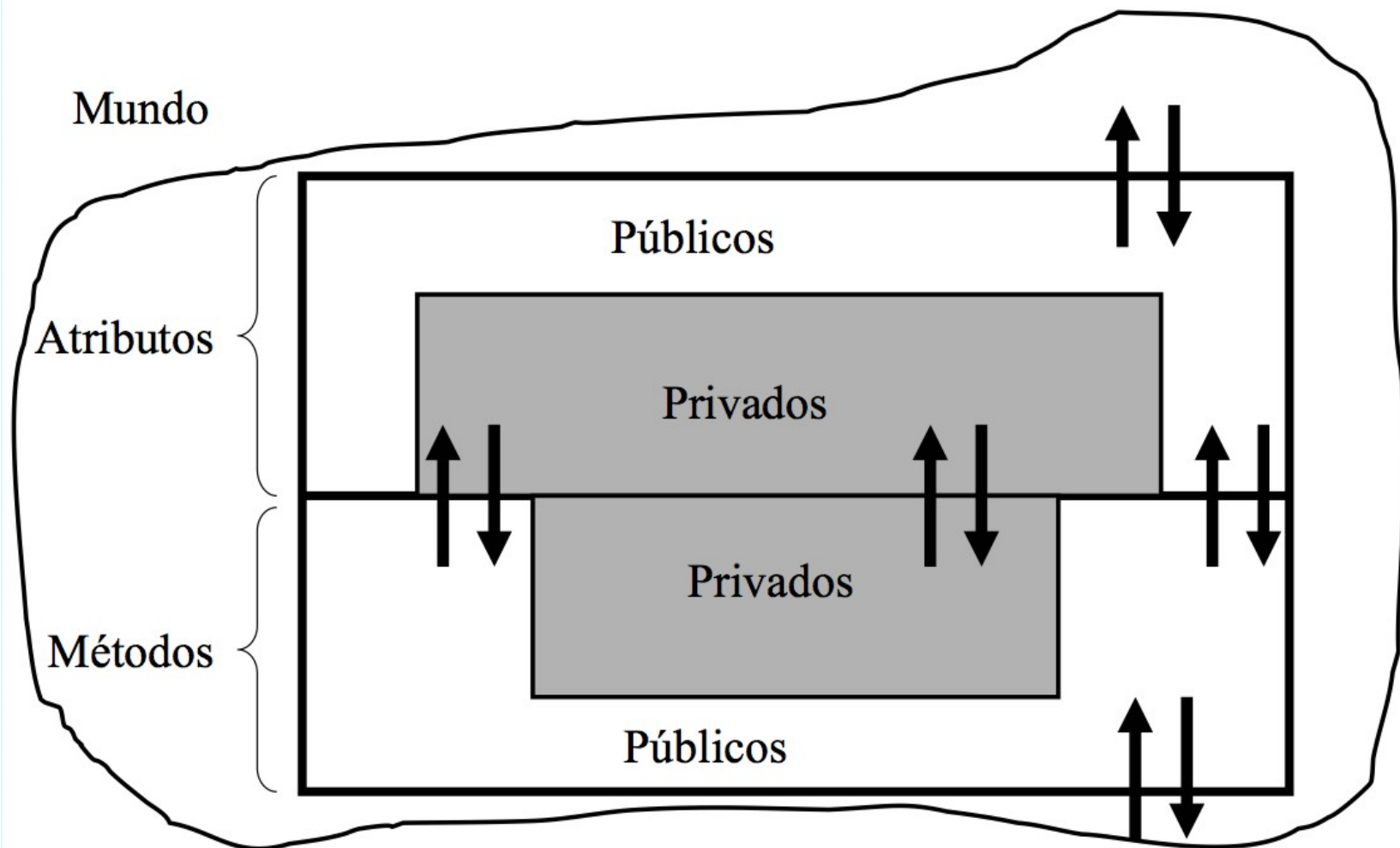
Diagrama UML de varias clases implementando una interfaz.

# Clase - Encapsulamiento

- **Ocultar** al exterior los detalles de implementación, de manera que el “mundo” sólo vea una interfaz inteligible (la *parte pública*).



# Clase – Encapsulamiento - 2



# Clase – Encapsulamiento - ventajas

- El encapsulamiento presenta tres **ventajas** básicas:
  - Se protegen los datos de accesos indebidos.
  - El acoplamiento entre las clases se disminuye.
  - Favorece la modularidad y el mantenimiento.
- Los atributos de una clase no deberían ser manipulables directamente por el resto de objetos
  - Generalmente los atributos de una clase son siempre privados y los métodos que los manipulan públicos

# Clase – Encapsulamiento - Representación

- En UML los **niveles de encapsulamiento** son:
  - **(-) Privado:** es el más fuerte. Esta parte es totalmente invisible hacia el exterior, sólo es visible para la propia clase.
  - **(#) Protegido:** Los atributos/operaciones protegidos son visibles sólo para las clases derivadas de la original.
  - **(+) Público:** Los atributos/operaciones públicos son visibles a otras clases (cuando se trata de atributos se está transgrediendo el principio de encapsulamiento).

# Encapsulamiento – Programación OO

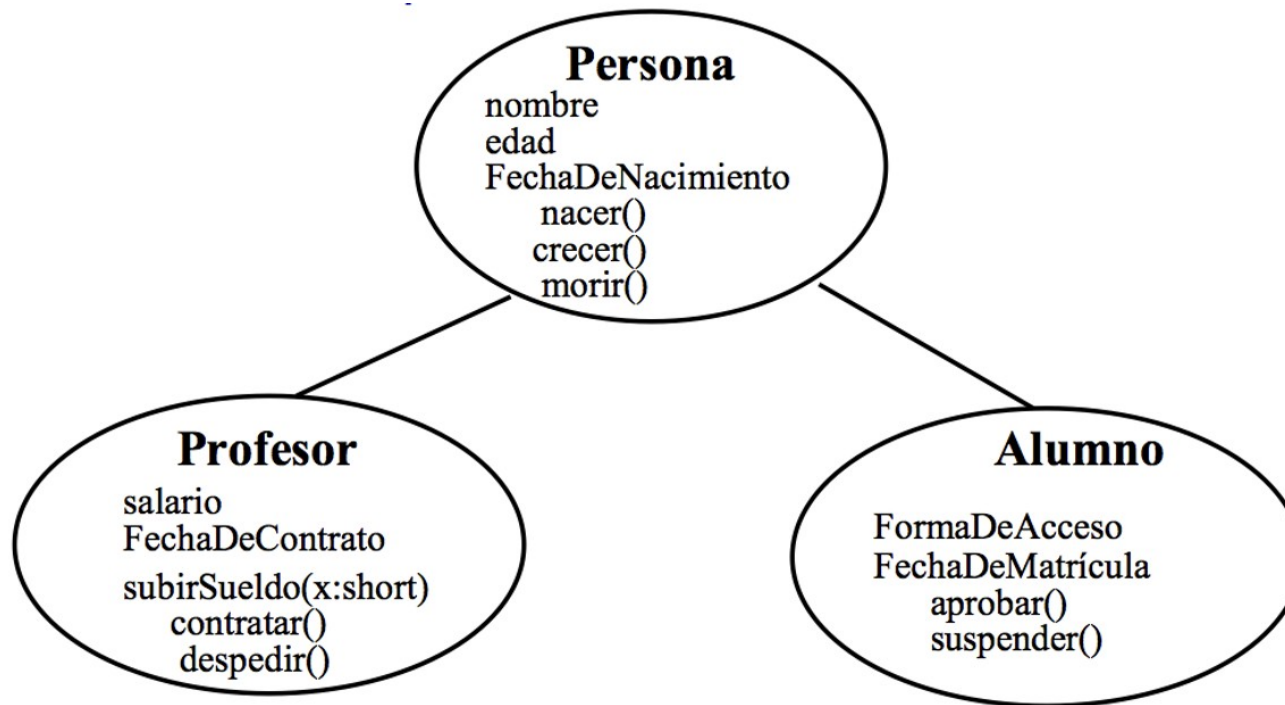
Es la característica de los objetos de ocultar sus propiedades y brindar métodos de acceso para manipular estas propiedades, evitando así que los demás objetos modifiquen directamente sus propiedades, dejándolo en un estado inconsistente. Analicemos el siguiente ejemplo:

```
public class Persona{  
    private String nombre;  
  
    public String getNombre (){  
        return nombre;  
    }  
    public void setNombre(String nombre){  
        this.nombre = nombre;  
    }  
}
```

De esta manera logramos el encapsulamiento, la clase *Persona* controla la forma en que sus propiedades son modificadas. Por ejemplo, podríamos agregar validación al método *setNombre* para que no permita un nombre vacío o mayor a N caracteres.



# Herencia



En POO, se conoce como herencia a la característica de heredar las propiedades de otra clase (su estructura y comportamiento).

La *herencia* es la capacidad de crear nuevas clases sobre otras existentes.

La principal ventaja de la herencia es la reutilización de código.

# Herencia – Programación OO

En Java, al igual que en otros lenguajes de programación, sólo se permite la herencia de una sola clase y de varias interfaces.

En Java la herencia se realiza utilizando la instrucción *extends* de una clase a otra, por ejemplo:

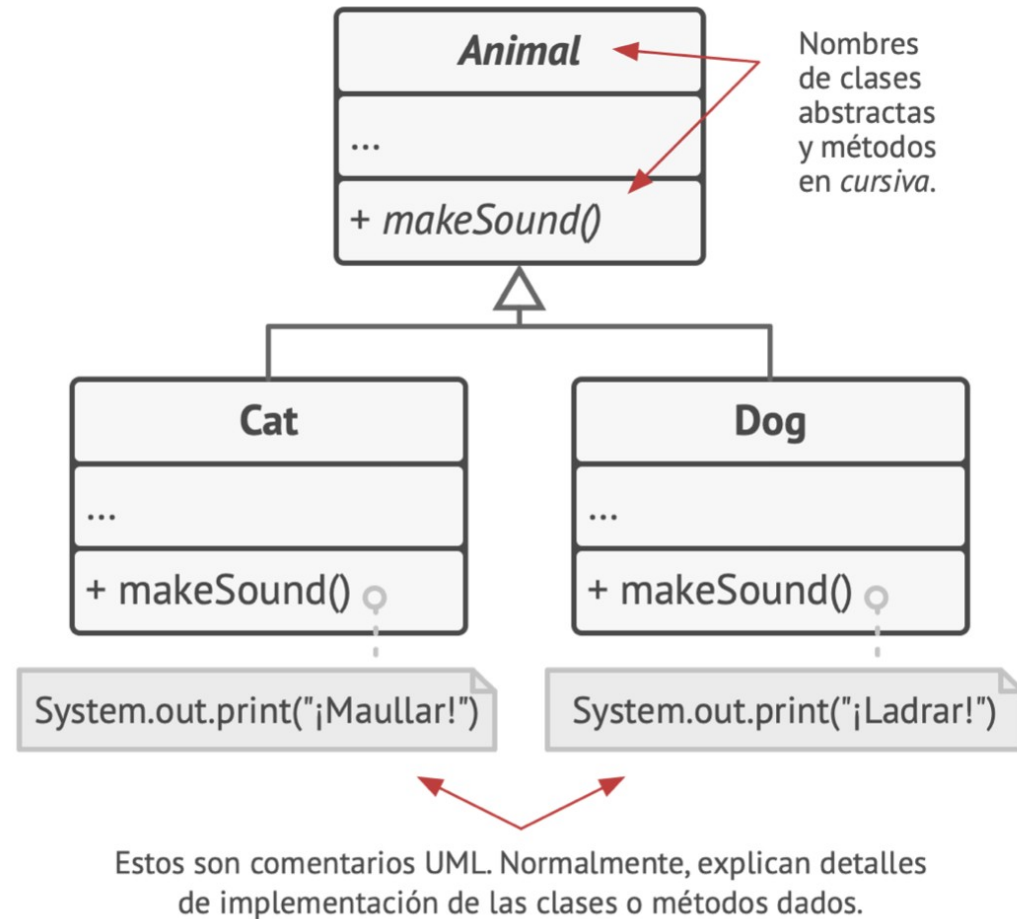
1. **public abstract class** Mamifero{}
2. **public class** Perro **extends** Mamifero{}
3. **public class** Murcielago **extends** Mamifero{}

# Polimorfismo

El término *polimorfismo* se refiere a algo que puede adoptar varias formas diferentes.

En OO se refiere a la capacidad de detectar la verdadera clase de un objeto e invocar su implementación, incluso aunque su tipo real sea desconocido en el contexto actual.

Es la posibilidad de desencadenar operaciones distintas, en respuesta a un mismo mensaje.



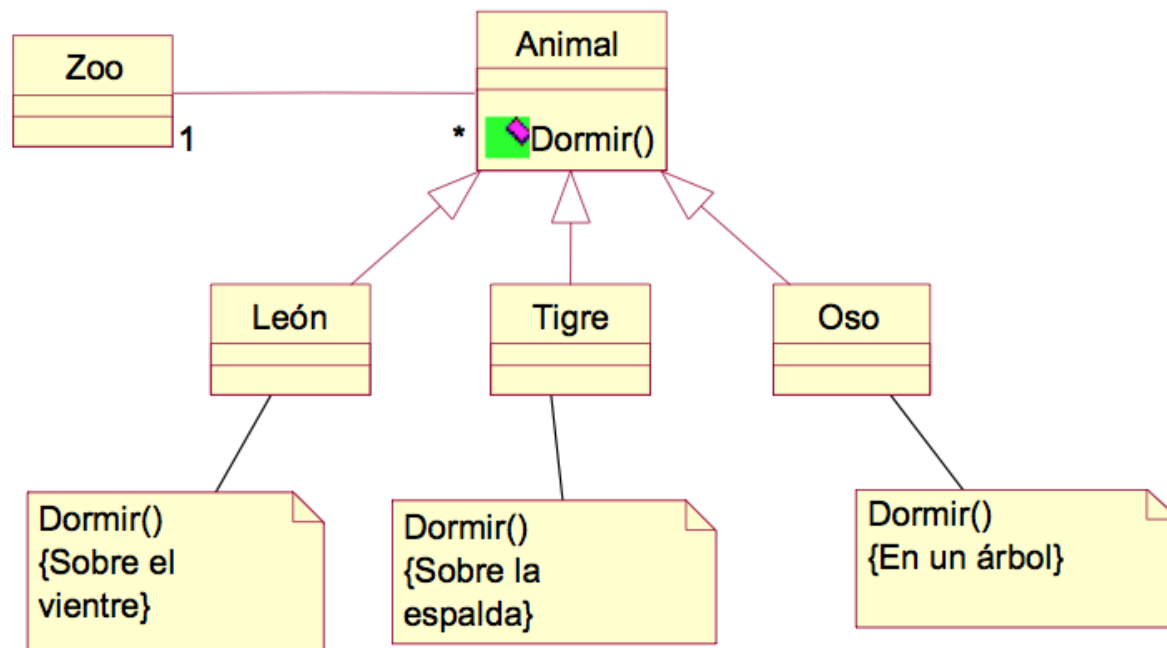
# Polimorfismo – Programación OO

Esta propiedad permite implementar una misma operación pero con comportamiento distinto y es la propiedad de POO más complicada de entender porque tiene sentido en tiempo de ejecución.

```
public abstract class Mamifero{  
    public abstract void comunicarse();  
}  
  
public class Perro extends Mamifero{  
    @Override  
    public void comunicarse(){  
        System.out.println("LadRAR");  
    }  
}  
  
public class Murcielago extends Mamifero{  
    @Override  
    public void comunicarse(){  
        System.out.println("Ultrasonido");  
    }  
}
```

# Polimorfismo - Ejemplo

- Todo animal duerme, pero cada especie lo hace de forma distinta.



# Relaciones

- Los enlaces que relacionan los objetos pueden verse de manera abstracta en el mundo de las clases:
  - A cada familia de enlaces entre objetos corresponde una relación entre las correspondientes clases
  - Un enlace entre dos objetos es una instancia de la relación entre las clases a las que pertenecen ambos objetos.
- Formas de **relación entre clases**:
  - Asociación [agregación como tipo particular]
  - Generalizaciones / Especializaciones ...
- Se pueden crear **jerarquías** entre clases mediante sucesivas agregaciones y/o generalizaciones.

# Asociación



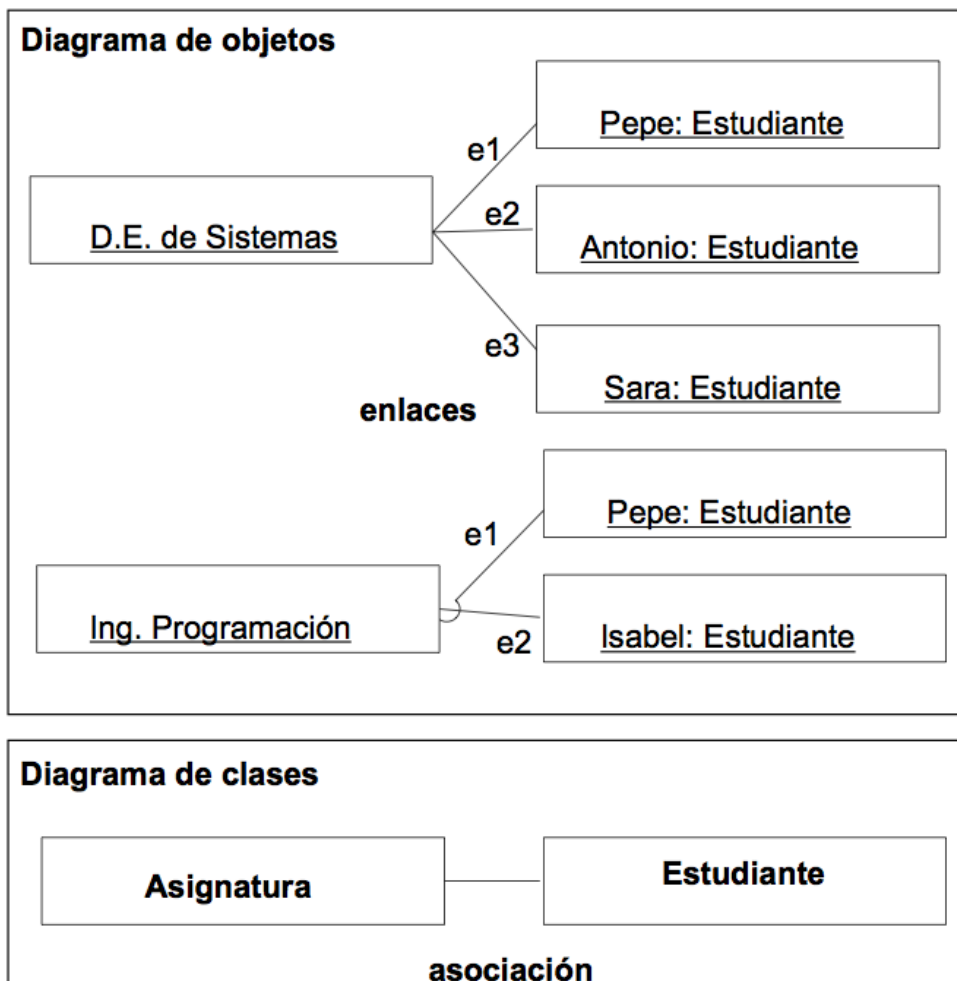
*Asociación en UML. El profesor se comunica con los estudiantes.*

La *asociación* es una relación en la que un objeto utiliza o interactúa con otro.

# Asociación (cont.)

- **Asociación:**

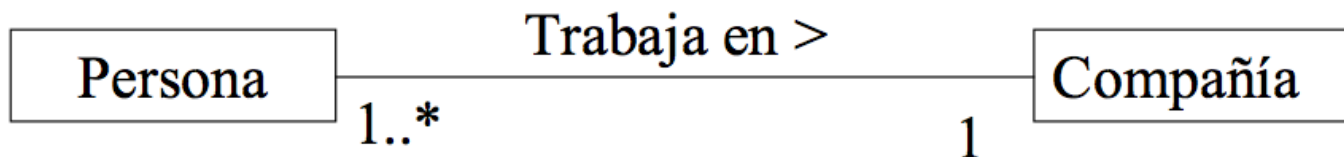
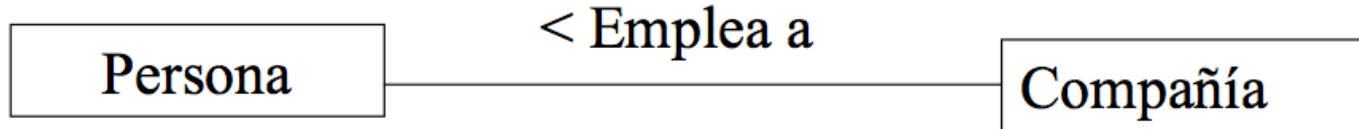
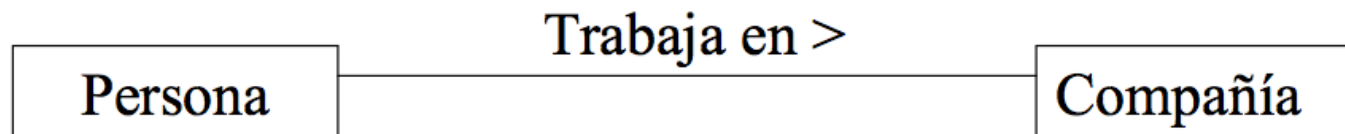
- Expresa una conexión semántica entre clases.
- Es una abstracción de los enlaces que existen entre los objetos instancias de esas clases.
- Se representan como los enlaces y se diferencian por el contexto del diagrama.





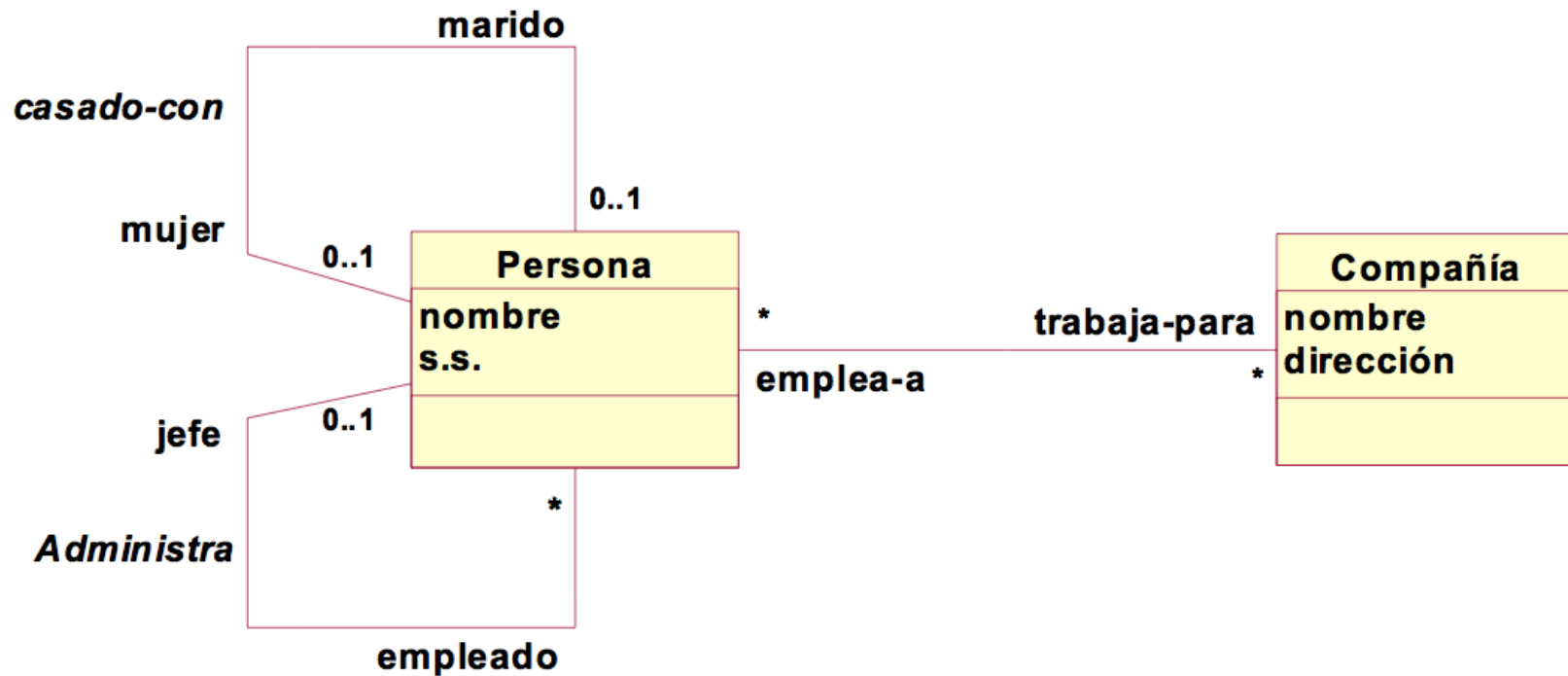
# Relaciones – Asociación - 2

- **Asociación:** [dirección y multiplicidad]



# Relaciones – Asociación - 3

- Ejemplo de **Asociaciones**:



# Relaciones – Asociación - 4

- **Multiplicidad** de **Asociaciones**

- **Mínima .. Máxima**

- 1 Uno y sólo uno
    - 0..1 Cero o uno
    - M..N Desde M hasta N (enteros naturales)
    - \* Cualquiera (cero o varios)
    - 0..\* Cualquiera (entre cero y varios)
    - 1..\* Uno o muchos (al menos uno)

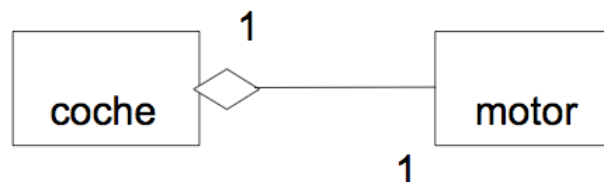
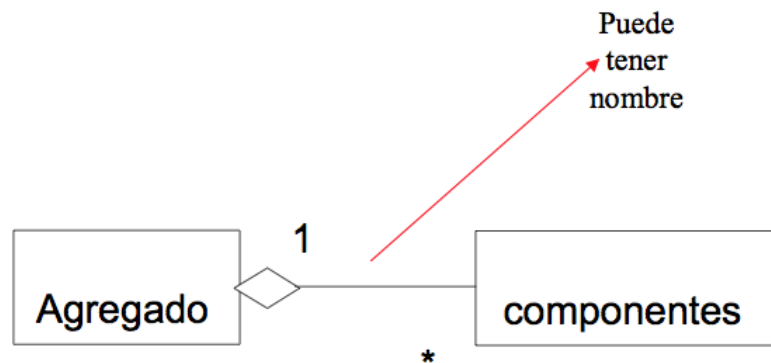
- Sólo suponen restricciones los casos:

- Mínimo mayor de cero (1 o  $>1$ )
    - Máximo menor que varios (0,1 u otro  $n \neq *$ )

# Relaciones - Agregación

## ● Agregación:

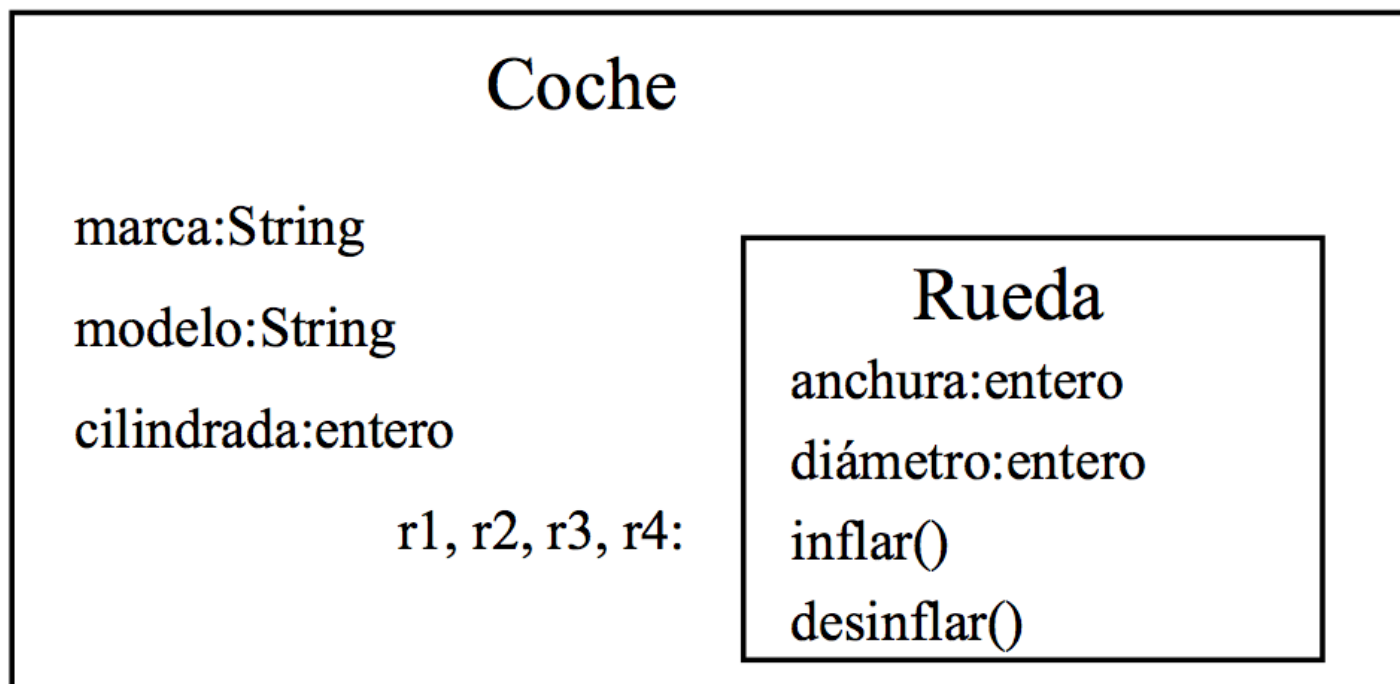
- Forma particular de asociación con acoplamiento **fuerte y asimétrico**
- Representa una relación **parte\_de** entre objetos
- Una de las clases cumple una función más importante que la otra.
- Permite representar asociaciones amo/esclavo, todo/partes, compuestos / componentes.



La existencia de ambos es independiente

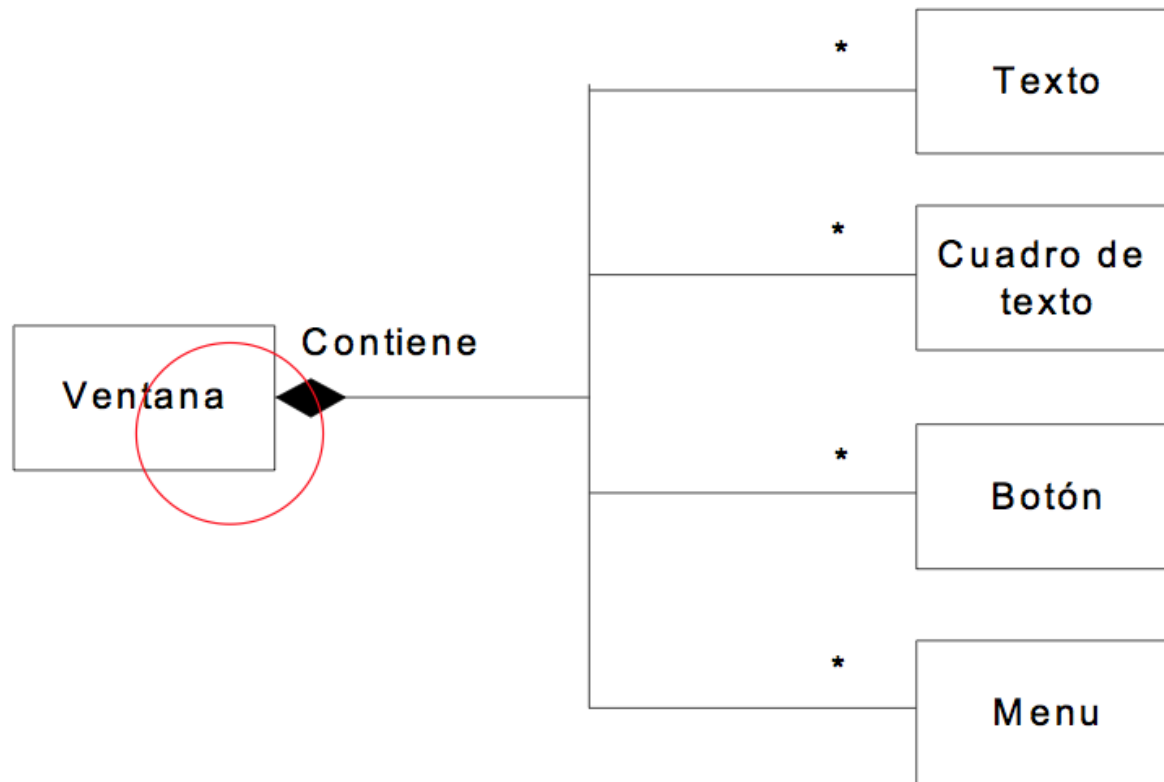
Agregación compartida ]

# Agregación - Ejemplo



# Agregación - Composición

- La **Composición** es un tipo específico de Agregación:



- La clase del "todo" debe controlar el ciclo de vida de sus clases "parte"

# Relaciones - Jerarquías

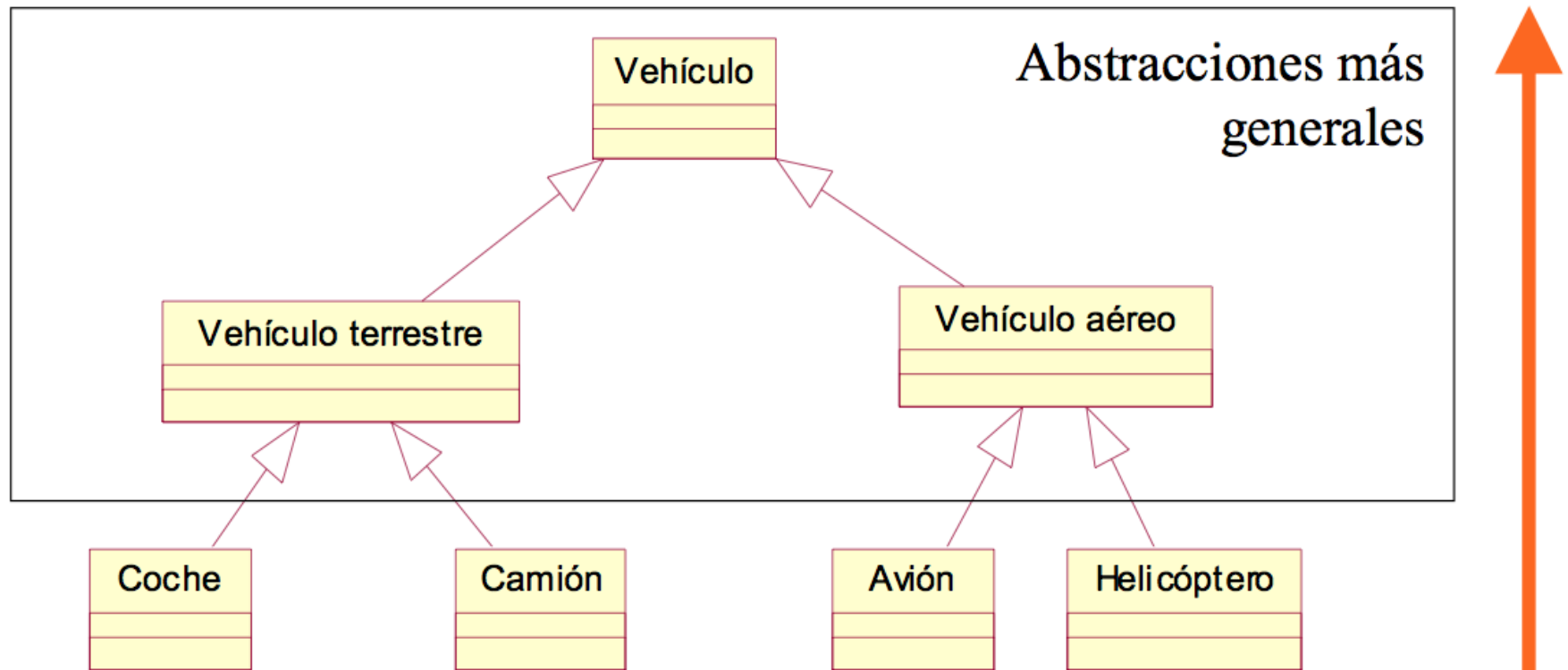
- Las jerarquías de clases o clasificaciones permiten gestionar la complejidad ordenando los objetos dentro de árboles de clases
  - **Generalización:**
    - ❖ Consiste en factorizar los elementos comunes (atributos, operaciones y restricciones) de un conjunto de clases en una clase más general llamada superclase
  - **Especialización:**
    - ❖ Permite capturar particularidades de un conjunto de objetos no discriminados por las clases ya identificadas. Las nuevas características se representan por una nueva clase, subclase de una de las clases existentes.
- La Generalización y Especialización son equivalentes en cuanto al resultado: la jerarquía y herencia establecidas

# Relaciones - Generalización

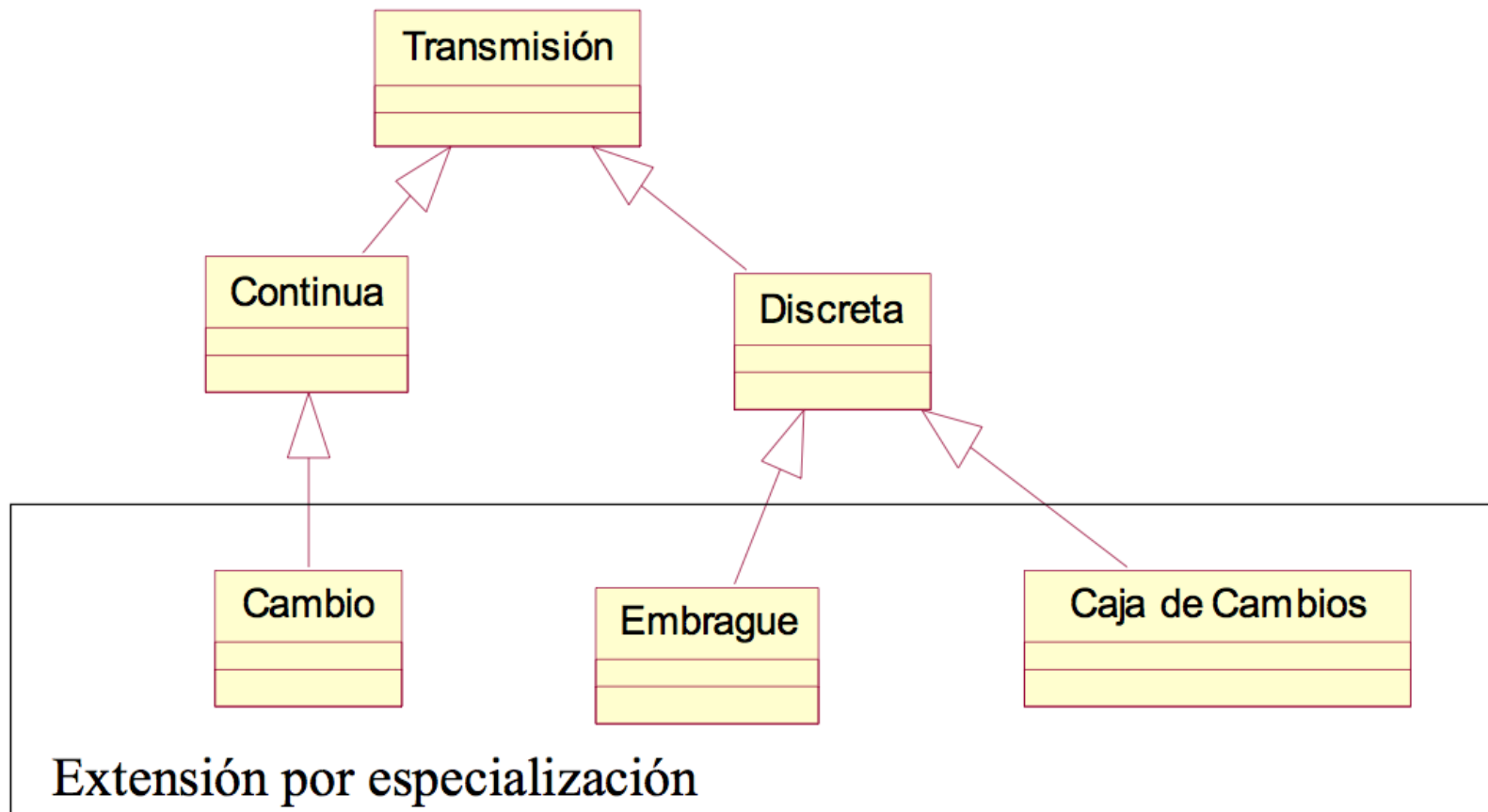
- Nomenclatura:
  - clase padre - clase hija
  - superclase – subclase
  - clase base - clase derivada
- Las subclases **heredan** propiedades de sus clases padre:
  - Los atributos, operaciones y asociaciones de la clase padre están disponibles en sus clases hijas.



# Generalización



# Especialización



# Generalización – Herencia entre clases

- Al utilizar Herencia entre clases debe tenerse siempre en cuenta el **Principio de Sustitución de Liskow**, que afirma que:

*"Debe ser posible utilizar cualquier objeto instancia de una subclase en el lugar de cualquier objeto instancia de su superclase sin que la semántica del programa escrito en los términos de la superclase se vea afectado."*

# Acoplamiento – Programación OO

El acoplamiento es la fuerza con la que dos clases o componentes están unidos, dicho de otra manera, indica el nivel de dependencia que tiene uno del otro.

Analicemos el siguiente ejemplo:

```
public class Printer {
    private ConsolePrinter salida;

    public Printer(ConsolePrinter salida){
        this.salida = salida;
    }
    public void generateMessage(){
        salida.process("Hello world!");
    }
    public static void main(String[] args) {
        Printer a = new Printer(new ConsolePrinter());
        a.generateMessage();
    }
}

public class ConsolePrinter {
    public void process(String message){
        System.out.println(message);
    }
}
```

# Acoplamiento II – Programación OO

Ahora bien, ¿qué tendríamos que hacer para desacoplar estas dos clases, de tal manera que la segunda no tenga o tenga muy poca dependencia de la *segunda* clase?

La solución es simple, lo primero que tenemos que hacer es crear una *Interface* que defina las operaciones básicas de la clase *ConsolePrinter*, seguido la clase *Printer* deberá recibir como parámetro esta interface en lugar de la clase concreta.

```
public interface IPrinterSalida {  
    public void process(String message);  
}
```

Que es implementada por:

```
public class ConsolePrinter implements IPrinterSalida{  
    @Override  
    public void process(String message){  
        System.out.println(message);  
    }  
}
```

# Acoplamiento III – Programación OO

La clase Printer nueva quedaría:

```
public class Printer {
    private IPrinterSalida salida;

    public Printer(IPrinterSalida salida){
        this.salida = salida;
    }
    public void generateMessage(){
        salida.process("Hello world!");
    }
    public static void main(String[] args) {
        Printer a = new Printer(new ConsolePrinter());
        a.generateMessage();
        Printer b = new Printer(new GUIPrinter());
        b.generateMessage();
    }
}

public class GUIPrinter implements IPrinterSalida{
    @Override
    public void process(String message){
        JOptionPane.showMessageDialog(null,message);
    }
}
```

# Cohesión – Programación OO

Así como el acoplamiento tiene que ver con cómo las clases se relacionan entre sí, la cohesión tiene que ver con la forma en que agrupamos unidades de software.

Con la cohesión buscamos que todas las unidades de software que creemos estén diseñadas para atacar un solo problema aumentando su reutilización.

Pero para ser más claro veámoslo así: Cohesión es la forma en la que agrupamos funciones en una librería, o la forma en la que agrupamos métodos en una clase, o la forma en la que agrupamos clases en un componente, o componentes en un módulo, etcétera.

Por ejemplo:

```
public interface Calculator {  
    public Number sum(Number a, Number b);  
    public Number sub(Number a, Number b);  
    public Number mult(Number a, Number b);  
    public Number div(Number a, Number b);  
    public void createGUI();  
}
```

# Cohesión II – Programación OO

Qué pasa con el método createGUI() si otro programa necesita usar esta clase para manejar sus operaciones?

Por ejemplo:

```
public interface Calculator {  
    public Number sum(Number a, Number b);  
    public Number sub(Number a, Number b);  
    public Number mult(Number a, Number b);  
    public Number div(Number a, Number b);  
}
```

Ahora si logramos una alta cohesión.

Como conclusión podemos decir que siempre deberíamos de buscar que todos los componentes de software que desarrollemos tengan alta cohesión y bajo acoplamiento.



# Bibliografía

## Obligatoria:

- LARMAN, Craig (2003): UML y patrones. 2ª edición. Prentice Hall.

## Complementaria:

- BENNETT, Simon, MCROBB, Steve y FARMER, Ray (2007): Análisis y diseño orientado a objetos con UML, 3ra edición. McGraw Hill.
- BOOCH, G., RUMBAUGH, J., JACKOBSON, I. (2006): Lenguaje Unificado de Modelado, Manual de Referencia Uml 2.0, Addison-Wesley.
- MEYER, Bertrand (2000): Construcción de Software Orientado a Objetos, Prentice Hall. 2da edición
- SCHACH, Stephen R. (2005): Análisis y Diseño Orientado a Objetos, Mc Graw Hill