

Contenido Gral. del Curso

- Conceptos básicos de Orientación a Objetos
- Análisis Orientado a Objetos:
 - Modelado del dominio
 - Comportamiento del sistema
- Diseño Orientado a Objetos:
 - Diseño de interacciones
 - Diseño de estructura
- Implementación Orientada a Objetos:
 - Codificación

Bibliografía Básica

- Applying UML and patterns - Craig Larman - Prentice Hall (2^a Ed. 2001) - ISBN 978–013–092–569–5. **Nota: este libro utiliza una versión 1.x de UML**
- UML Distilled - Martin Fowler - Addison Wesley (3^a Ed. 2003) - ISBN 978–032–119–368–1. **Nota: este libro utiliza la versión 2.0 de UML**
- Design Patterns - Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides - Addison-Wesley (1995) - ISBN 0–201–63361–2
- Cómo programar en C/C++ - H.M. Deitel y P.J. Deitel - Prentice Hall (1995) - ISBN 968–880–471–1

NOTA: UML LENGUAJE DE MODELADO

INTRODUCCION AL PROCESO DE DESARROLLO

Marco general.

DESARROLLO

El desarrollo es un proceso de software de que describe un enfoque para construir, instalar y mantener sistemas de software.

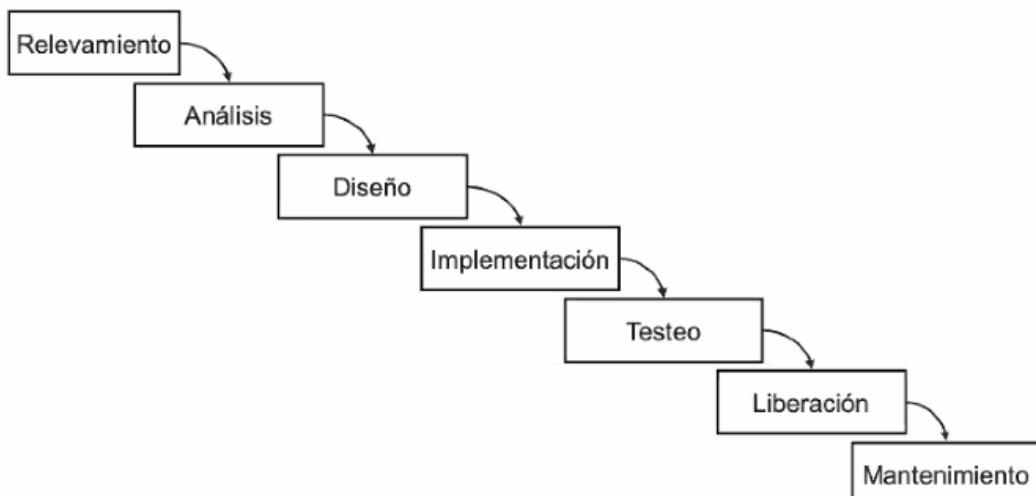
Lo necesitamos para conocer de antemano que actividades debemos realizar.

ALGUNAS ACTIVIDADES DEL PROCESO DE DESARROLLO

- Hablar con el cliente
- Obtener una descripción de lo que se espera
- Comprender
- Determinar
- Hacerlo
- Probar
- Entregar
- Hacerle retoques varios
- Mantenerlo
- Realizar estimaciones de tiempo, de costo, de recursos
- Planificar
- Asegurarse que las cosas se hagan:
 - En el tiempo resisto
 - De la forma establecida
- Administrar las diferentes versiones de lo que se va producido
- Montar y mantener los ambientes de desarrollo y prueba.

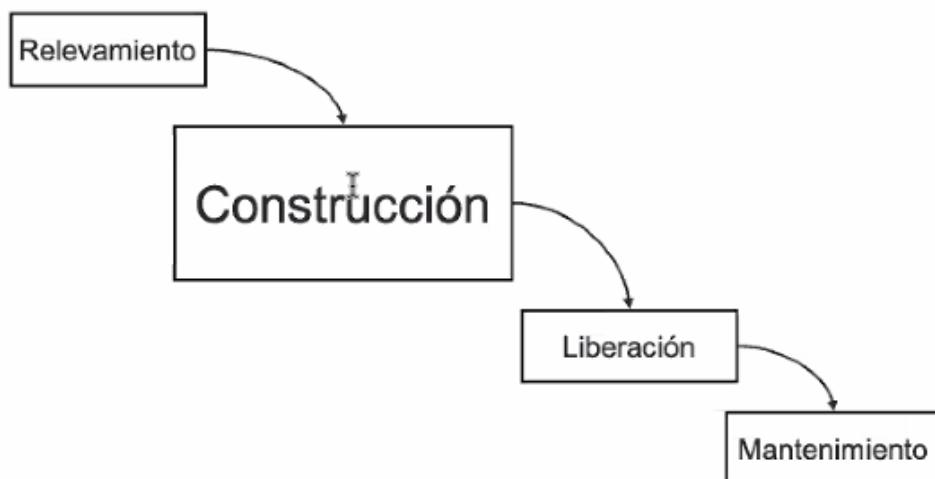
Un Modelo de Proceso

Cascada:



Otro Modelo

Iterativo e Incremental (I&I):



Características

Se divide el problema en varios subproblemas
Las iteraciones se producen en “Construcción”
Se itera sobre una “mini cascada” donde se resuelve cada subproblema:

```
for each (sp:Subproblema) {  
    análisis(sp);  
    diseño(sp);  
    implementación(sp);  
    testeo(sp);  
}
```

En la iteración i se resuelve sp_i , llevándose resueltos los subproblemas: $sp_1, sp_2, \dots, sp_{i-1}$

Nuestro Proceso

Para poder realizar un proceso I&I es necesario conocer un proceso en cascada
Nos concentraremos en algunas actividades dentro de la “cascada” de Construcción:

- Análisis
- Diseño
- Implementación

Los pasos concretos a realizar en estas actividades depende del paradigma de desarrollo a seguir

11/03/2021

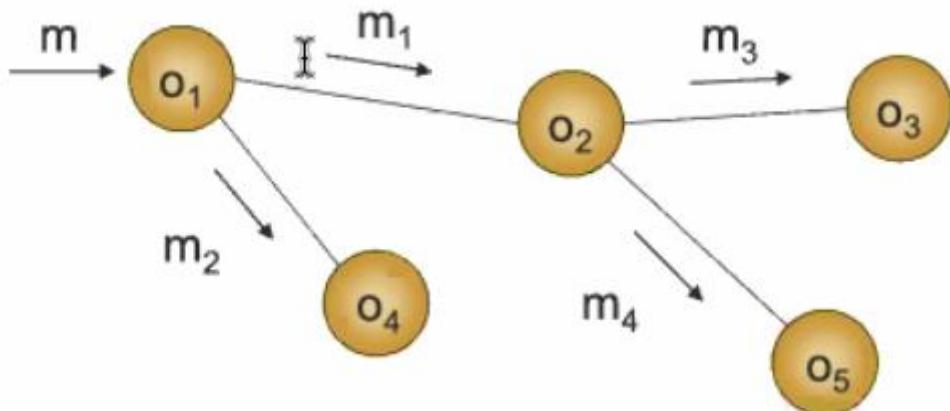
INTRODUCCION AL PARADIGMA DE ORIENTACION A OBJETOS

ENFOQUE DIFERENTE AL TRADICIONAL: Puede ser entendida como:

Una forma de pensar basada en atracción de conceptos existentes en el mundo real.
Organizar el software como una colaboración de objetos.

Ahora vamos a enfocarnos el orientado a objetos:

Una aplicación orienta a objetos es el resultado de la codificación en un lenguaje de programación orientada a objetos del esquema:



Objeto 1 (O1) depende de objeto 2 (O2). Todo lo vamos a ver en el enfoque de objetos.

Desarrollo Orientado a Objetos

Los pasos generales de desarrollo se mantienen en el enfoque orientado a objetos

Pero las actividades que constituyen algunos de ellos son particulares:

Análisis Análisis Orientado a Objetos

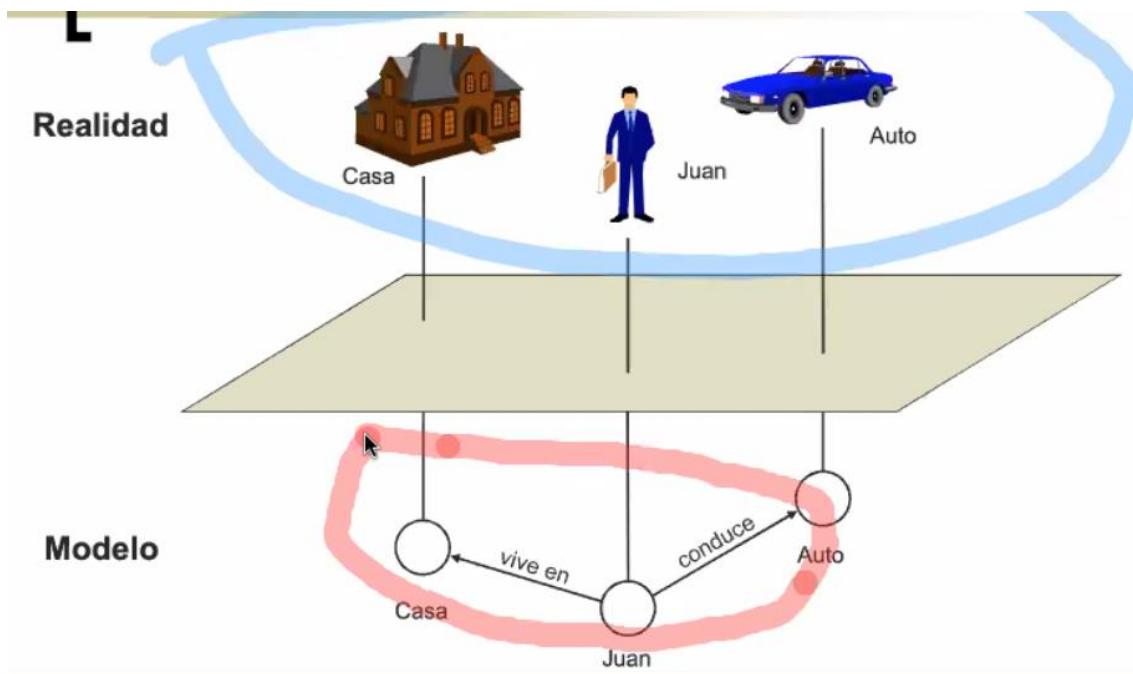
Diseño Diseño Orientado a Objetos

Implem. Implem. Orientada a Objetos

Pasa por las 3 etapas, cada una de ellas tiene un objetivo:

Análisis su objetivo es atracción de conceptos principales o clave.

La parte azul es la realidad y la parte rosa es el modelo entendido



Diseño es como hacer las cosas.

Objetivo: definir objetos lógicos (de software) y la forma de comunicación entre ellos para una posterior programación

En base a los “conceptos candidatos” encontrados durante el análisis y por medio de ciertos principios y técnicas, se debe decidir:

Cuáles de éstos serán los objetos que participarán en la solución

Cómo se comunican entre ellos para obtener el resultado deseado

Concepto clave: *responsabilidades*

En esta transición:

No todos los conceptos necesariamente participarán de la solución

Puede ser necesario “reflotar” conceptos inicialmente dejados de lado

Será necesario fabricar “ayudantes” (también objetos) para que los objetos puedan llevar a cabo su tarea

Implementación, ahora es plasmar lo que quedo de las partes anteriores.

Objetivo: codificar en un lenguaje de programación orientado a objetos las construcciones definidas en el diseño

La definición de los objetos y el intercambio de mensajes requieren construcciones particulares en el lenguaje a utilizar

EJEMPLO JUEGO DE DADOS

PROBLEMA SENCILLO PARA ILUSTRAR CONCEPTOS CLAVES

Actividades a realizar:

- Modelado del dominio
- Definición de interacción
- Definición de estructura
- Codificación

DOMINIO



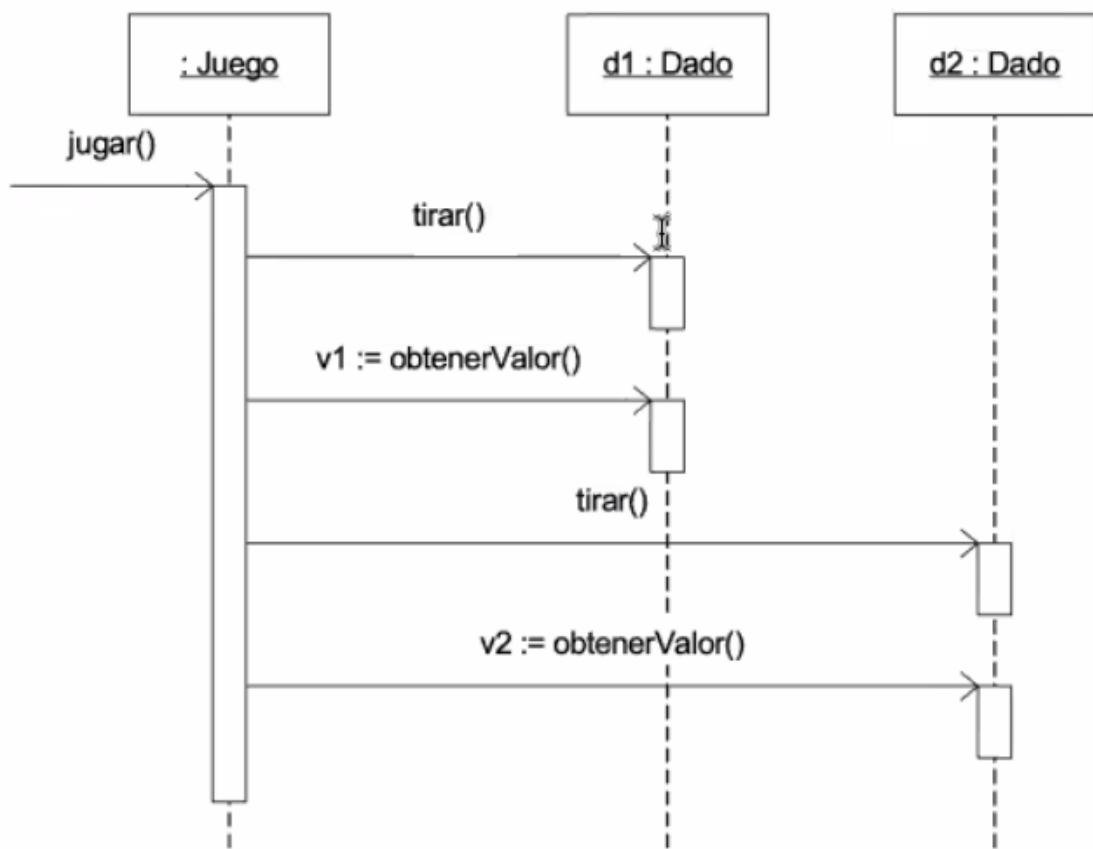
Líneas son las asociaciones.

Lo leemos (se puede leer en todos sentidos):

- Un jugador tira dos dados.
- Un jugador juega un juego.
- Un juego incluye dos dados.

ITERACIONES

El juego interactúa con el mundo externo, puede ser un usuario de afuera. Me innovador jugar tira un dado y le pide el valor que dio y lo guarda en v1. Tira nuevamente y lo guarda en v2.



Hasta aquí llega la parte del análisis. No va mas profundo que esto.

ESTRUCTURA



Tenemos una navegabilidad, Juego navega hacia dado, por eso la flecha (Si la flecha tiene para los dos lados es de doble navegabilidad para tener una idea). También tenemos las operaciones que son jugar (), tirar ()...

CODIFICACION

```
// Dado.h
class Dado {
    public:
        int obtenerValor();
        void tirar();
    private:
        int valor;
};
```

```
// Dado.cpp
#include "Dado.h"
int Dado::obtenerValor() {
    return valor;
}
void Dado::tirar() {
    valor = ... //random
}
```

```
// Juego.h
class Juego {
    public:
        void jugar();
    private:
        Dado* d1, d2;
};
```

```
// Juego.cpp
#include "Juego.h"
void Juego::jugar() {
    int v1,v2;
    ...
    d1->tirar();
    v1 = d1->obtenerValor();
    d2->tirar();
    v2 = d2->obtenerValor();
    ... // algo con v1 y v2
}
```

```
main(){
    ...
    Dado d1: new Dado();
    Dado d2: new Dado();
    Juego j = new Juego(d1,d2);
    j.jugar();
}
```

NOTAS:

Cuando los definimos no tiene ningún valor hasta que pasa por el main.

Las clases son declaraciones.

.h la especificación

.cpp la implementación

HERRAMIENTAS DE MODELADO

Sirve de ayuda para el desarrollo de la tarea, visualizar lo echo hasta el momento, etc.

UML es el estándar para modelado de software.

De estructura estática: Muchos tipos de diagramas están comprendidos en UML.

Clases (*)

Objetos (*)

De interacción:

Secuencia (*)

Comunicación (*)

De implementación:

Componentes (*)

Distribución

De casos de uso

De actividad

De estados

Mecanismos de propósito general (*)

Notas

Paquetes

Mecanismos de extensión (*)

Estereotipos

Restricciones

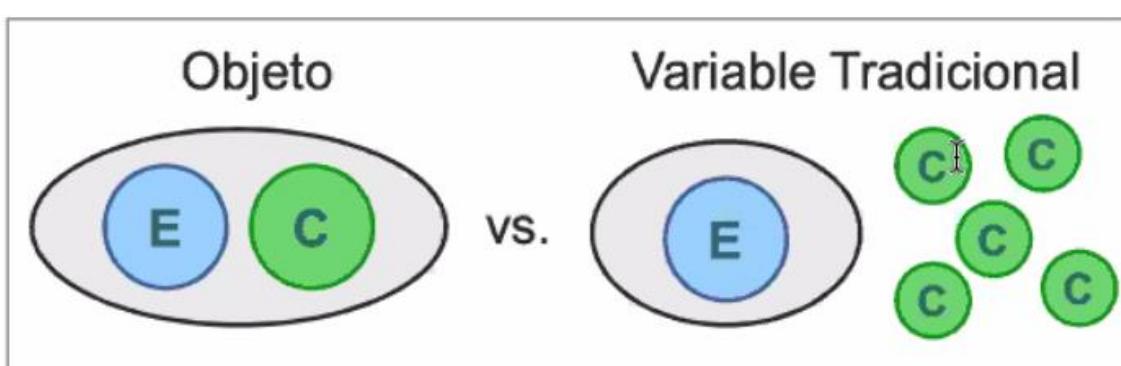
Tagged values

Object Constraint Language (*)

CONCEPTOS BASICOS DE ORIENTACION A OBJETOS

CONSTRUCCION BASICA

Un objeto es una entidad discreta como límites e identidad bien definidos. Encapsula **estado** y comportamientos (es el hecho de como fluctúa en el tiempo) (ambas conforman la clase):



Es una instancia de clase.

Identidad



Es una propiedad inherente de los objetos de ser distinguible de todos los demás

Dos objetos son distintos aunque tengan exactamente los mismos valores en sus propiedades

Conceptualmente un objeto no necesita de ningún mecanismo para identificarse

La identidad puede ser realizada mediante direcciones de memoria o claves (pero formando parte de la infraestructura subyacente de los lenguajes)

Clase

Una clase es un descriptor de objetos que comparten los mismos atributos, operaciones, métodos, relaciones y comportamiento

Una clase representa un concepto en el sistema que se está modelando

Dependiendo del modelo en el que aparezca, puede ser un concepto del mundo real (modelo de análisis) o puede ser una entidad de software (modelo de diseño)

```
class CEjemplo {
    ... // definicion de
    ... // las propiedades
    ... // de la clase Ejemplo
}
```

```
CEjemplo *e = new CEjemplo;
delete e;
```

Lo mismo que hacíamos después de definir una struct.

Para crear un objeto se definen constructores

```
CEjemplo(); //por defecto
CEjemplo(params); //común
CEjemplo(CEjemplo *); //por copia
```

Para destruir un objeto se define un destructor

```
~CEjemplo();
```

Por defecto no le pasamos nada, si le pasa algo entre los paréntesis seria común y si se le pasa la misma clase se copia.

Si ponemos delete e; eso realiza:

Atributo

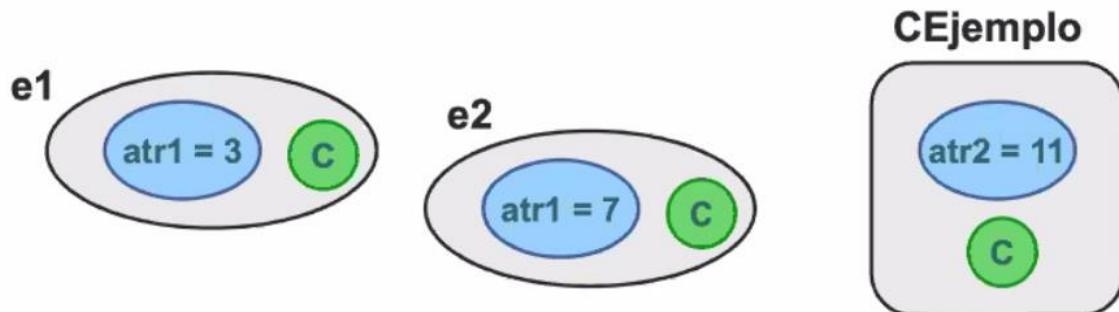
Es una descripción de un compartimiento de un tipo especificado dentro de una clase

Puede ser:

De Instancia: Cada objeto de esa clase mantiene un valor de ese tipo en forma independiente

De Clase: Todos los objetos de esa clase comparten un mismo valor de ese tipo

```
class CEjemplo {  
    int atr1;           // Atributo de instancia  
    static int atr2;   // Atributo de clase  
    ...                // Otras propiedades  
}
```



La operación es una especificación de una transformación o consulta que puede ser llamada a ejecutar. Tiene asociada un nombre, una lista de parámetros y un tipo de retorno.

Operación y Método

```
class CEjemplo {  
    int atr1;  
    static int atr2;  
}  
  
void oper(char c)  
{  
    ... // un cierto algoritmo  
}
```

Operación

Método para **operO** en CEjemplo

Estado

El estado de una instancia almacena los efectos de las operaciones

Está implementado por

 Su conjunto de atributos

 Su conjunto de [links](#)

Es el valor de todos los atributos y links de un objeto en un instante dado

Comportamiento

Es el efecto observable de una operación, incluyendo su resultado

Acceso a Propiedades



Las propiedades de una clase tienen aplicadas calificadores de acceso

Una propiedad de un objeto calificada con:

public: puede ser accedida desde cualquier punto desde el cual se tenga visibilidad sobre el objeto

private: puede ser accedida solamente desde los métodos de la propia clase

Existen otros calificadores (i.e. **protected**) pero su semántica depende del lenguaje de implementación

Acceso a Propiedades (2)

Por defecto, los atributos deben ser privados y las operaciones públicas:

```
class CEjemplo {
    private:
        int atr1;

    public:
        void operO {
            this.atr1 = 2;      // código valido
        }
}

e.atr1      // código no valido
e->operO   // código valido
```

Polimorfismo

Es la capacidad de asociar diferentes métodos a la misma operación

¡No alcanza con que tengan el mismo nombre, para ser polimorfismo deben ser realmente la misma operación!

```
class A {
    void operO {
        // un método
    }
}
```

```
class B {
    void operO {
        // otro método
    }
}
```

En este caso no se trata de la misma operación (aunque tengan la misma firma) dado que las dos clases no están relacionadas entre sí

Data Type

Es un descriptor de un conjunto de valores que carecen de identidad

Data types pueden ser tipos primitivos predefinidos como:

Strings

Números

I

Fechas

También tipos definidos por el usuario, como enumerados

Muchos lenguajes de programación no tienen una construcción específica para data types

En esos casos se implementan como clases:

Sus instancias serían formalmente objetos

Sin embargo, la identidad de esas instancias es ignorada

```
class Racional {  
    private: int numerador, denominador;  
    public:  
        Racional (int = 0, int = 1);  
        int getNumerador();  
        int getDenominador();  
  
        Racional operator+ (Racional);  
        ...  
}
```

Para que sea un datatype, las operaciones no pueden modificar el estado interno del objeto sino retornar uno nuevo

Data Value

Es un valor único que carece de identidad, una instancia de un data type

Un data value no puede cambiar su estado:

Eso quiere decir que todas las operaciones aplicables son “funciones puras” o consultas

Los data values son usados típicamente como valores de atributos

Set y get lectura y escritura

Valores y Cambios de Estado

El valor “4” no puede ser convertido en el valor “5”

Se le aplica la operación suma con argumento “1” y el resultado es el valor “5”

A un objeto persona se le puede cambiar la edad:

Reemplazando el valor de su atributo “edad” por otro valor nuevo

El resultado es la misma persona con otra edad

Identidad o no Identidad

¿Cómo saber si un elemento tiene o no identidad?:

Dos objetos separados que sean idénticos lucen iguales pero no son lo mismo (son distinguibles por su identidad)

Dos data values separados que sean idénticos son considerados lo mismo (no son distinguibles por no tener identidad)

RELACIONES

Asociación

Una asociación describe una relación semántica entre clasificadores (clases o data types)

Las instancias de una asociación ([links](#)) son el conjunto de tuplas que relacionan las instancias de dichos clasificadores

Cada tupla puede aparecer como máximo una sola vez en el conjunto

Instancia de las asociaciones, eso son los link.

Una asociación entre clases indica que es posible “conectar” entre sí instancias de dichas clases

Cuando se desea poder conectar objetos de ciertas clases, éstas deben estar relacionadas por una asociación

Una asociación R entre clases A y B puede entenderse como $R \subseteq A \times B$

Los elementos en R pueden variar con el tiempo

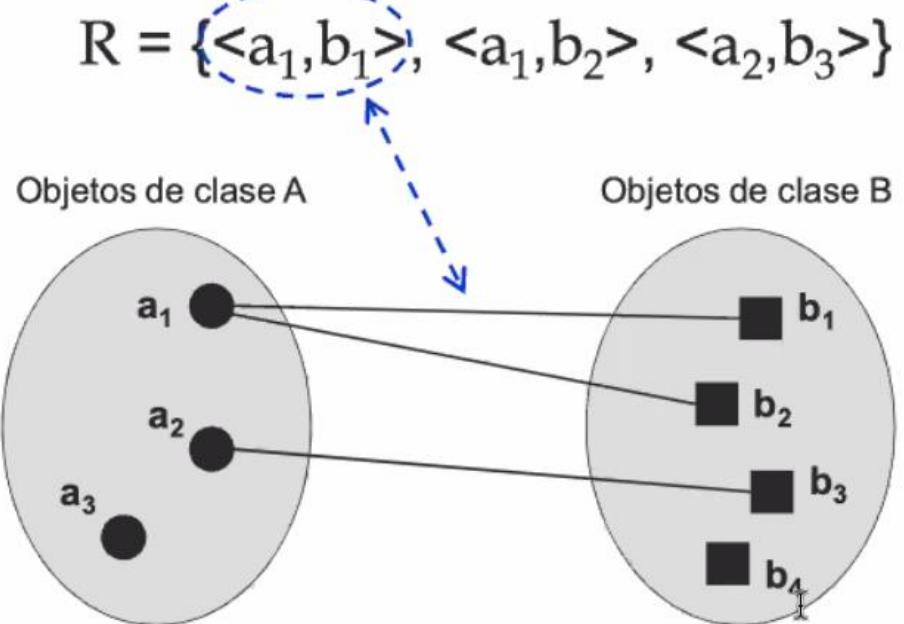
Link

Es una tupla de referencias a instancias (objetos o data values)

Es una instancia de una asociación

Permite visibilidad entre todos las instancias participantes

Ejemplo: asociación R entre clases A y B



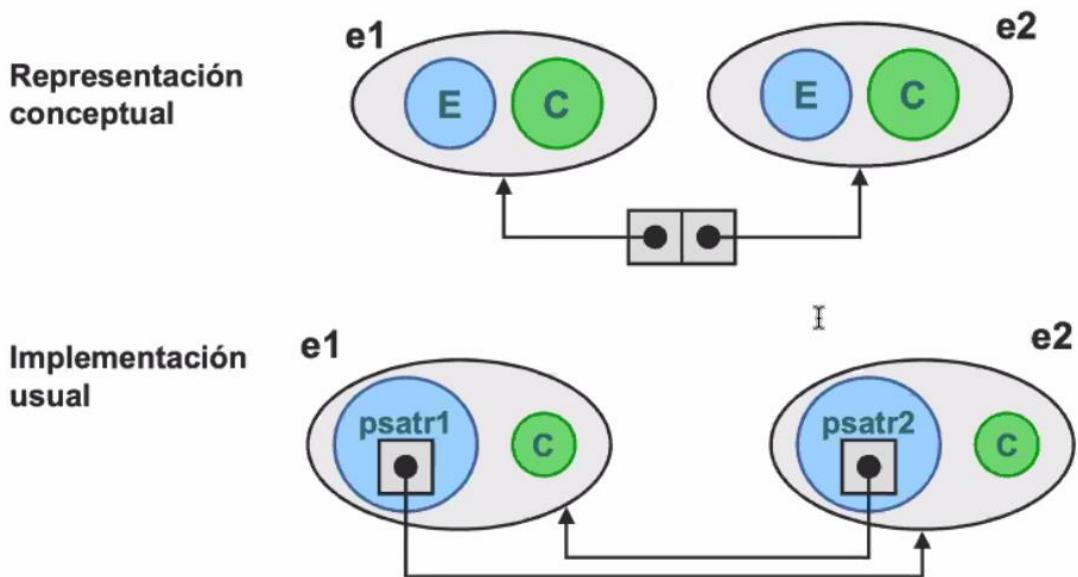
Representación de Asociaciones

Casi ningún lenguaje provee construcciones específicas para implementar asociaciones

Para ello se suelen introducir “pseudoatributos” en las clases involucradas

De esta manera un link no resulta implementado exactamente igual a su representación conceptual

Una tupla es dividida y un componente es ubicado en el objeto referenciado por el otro componente de la tupla



Representación de Asocs. (3)

Ejemplo: Asociación entre **Persona** y **Empresa**

```
class Persona {
    private:
        String nombre;      // atributo
        Empresa *miEmp;   // pseudoatributo
        ...
}
```

El tipo de un **pseudoatributo** suele ser una clase, pero el de un **atributo** debe ser un data type

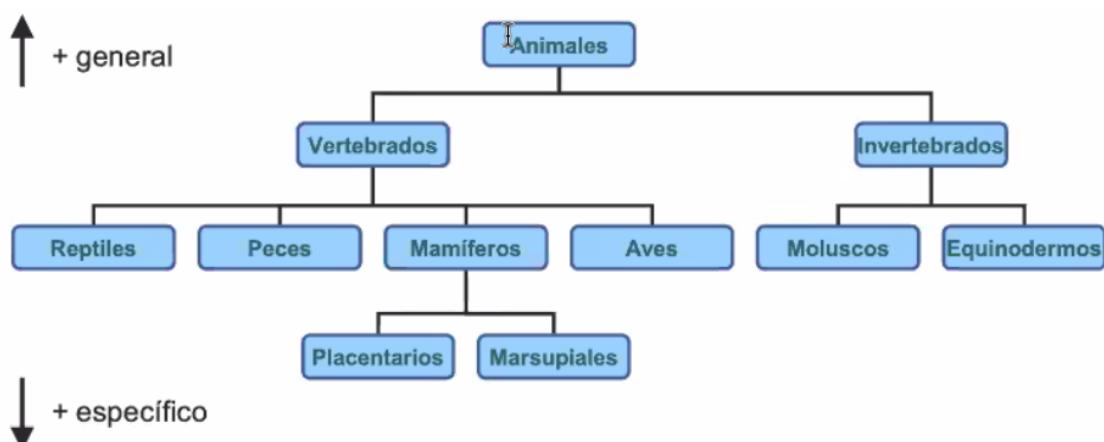
Por cuestiones de costo si una de las visibilidades no es necesaria usualmente no se implementa

Generalización

Una generalización es una relación taxonómica entre un elemento (clase, data type, interfaz) más general y entre un elemento más específico

El elemento más específico es consistente (tiene todas sus propiedades y relaciones) con el más general, y puede contener información adicional

Es una jerarquía, de algo más general a algo más específico



Se conserva las propiedades y se van agregando otras.

Clase Base y Clase Derivada



Cuando dos clases están relacionadas según una generalización, a la clase más general se la denomina *clase base* y a la más específica *clase derivada* de la más general

A una clase base se la denomina también *superclase* o *padre*

A una clase derivada se la denomina también *subclase* o *hijo*

Una clase puede tener cualquier cantidad de clases base, y también cualquier cantidad de clases derivadas.

```
class Vehiculo {  
    ... // propiedades de vehículo  
}  
  
class Auto : public Vehiculo {  
    ... // props específicas de auto  
}  
  
class Moto : public Vehiculo {  
    ... // props específicas de moto  
}
```

Auto es una subclase de vehículo, hereda todas las propiedades de vehículo.

Ancestros y Descendientes

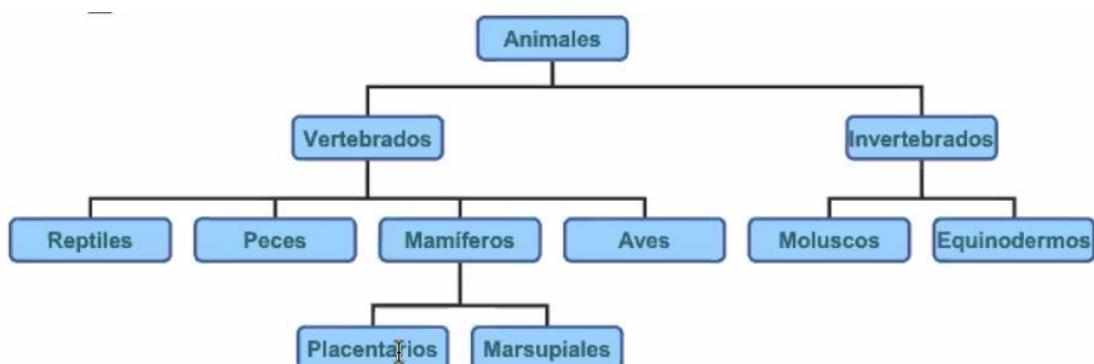


Los ancestros de una clase son sus padres (si existen), junto con los ancestros de éstos

Los descendientes de una clase son sus hijos (si existen), junto con los descendientes de éstos

Una clase es clase base directa de sus hijos, y una clase es clase derivada directa de sus padres

Una clase es clase base indirecta de los descendientes de sus hijos, y una clase es clase derivada indirecta de los ancestros de sus padres



- Ancestros de Marsupiales son {Mamíferos, Vertebrados, Animales}
- Descendientes de Invertebrados son {Moluscos, Equinodermos}
- Ave es clase derivada directa de Vertebrados e indirecta de Animales
- Vertebrados es clase base directa de Mamíferos e indirecta de Marsupiales

Subclassing



Se define la relación entre clases:

$(\lessdot) : \text{Clase} \times \text{Clase} \rightarrow \text{Prop}$

donde,

$(B, A) \in (\lessdot) \Leftrightarrow B \text{ es clase derivada de } A$

Observación: La relación \lessdot define un orden parcial entre clases y es idéntica, transitiva y antisimétrica.

Es una propiedad que deben cumplir todos los objetos, también conocida como *intercambiabilidad*

Un objeto de clase base puede ser sustituido por un objeto de clase derivada (directa o indirecta)

Por lo tanto: $b : B \wedge B \lessdot A \Rightarrow b : A$

Esto se puede leer como: “un objeto instancia de una clase derivada es también instancia de cualquier clase base”

Ejemplo: “Todo auto es un vehículo”

```
class Auto : public Vehiculo {  
    ... // props específicas de auto  
}
```

Descriptors

Un *full descriptor* es la descripción completa que es necesaria para describir a un objeto

Contiene la descripción de todos los atributos, operaciones y asociaciones que el objeto contiene

Un *segment descriptor* son los elementos que efectivamente se declaran en un modelo o en el código (por ejemplo, clases) y contienen las propiedades heredables que son:

los atributos

las operaciones y los métodos

la participación en asociaciones (los pseudoatributos)

```
class Empleado {  
    private: string nombre;  
    Empresa miEmp;  
    public: string getNombre() {  
        return nombre;  
    }  
}
```

```
class Fijo : public Empleado {  
    private: float sueldo;  
    public: float getSueldo() {  
        return sueldo;  
    }  
}
```

SD_{Empleado}

ATT_{nombre}

PSA_{miEmp}

OP_{getNombre}

MET_{Empleado::getNombre}

SD_{Fijo}

ATT_{sueldo}

OP_{getSueldo}

MET_{Fijo::getSueldo}

SD segment descriptor.

En un lenguaje orientado a objetos, la descripción de un objeto es construida incrementalmente a partir de segmentos

Los segmentos son combinados mediante **herencia** para producir el descriptor completo de un objeto

El mecanismo de herencia define cómo full descriptors son producidos a partir de un conjunto de segment descriptors conectados entre sí por generalización

Los full descriptors son implícitos pero son quienes definen la estructura de objetos concretos

Herencia

Es el mecanismo por el cual se permite compartir propiedades entre una clase y sus descendientes

Define la forma en que el full descriptor de una clase es generado

Si una clase no tiene ningún parente entonces su full descriptor coincide con su segment descriptor

Si tiene uno o más padres, entonces su full descriptor se construye como la unión de su propio segment descriptor con los de todos sus ancestros

La clase para la cual se genera el full descriptor hereda las propiedades especificadas en los segmentos de sus ancestros

I

Para una clase, no es posible declarar un atributo u operación con el mismo prototipo en más de uno de los segmentos:

Si eso ocurriera el modelo estaría mal formado

$$FD_{\text{Empleado}} = SD_{\text{Empleado}}^I$$

$$FD_{\text{Fijo}} = SD_{\text{Empleado}} \oplus SD_{\text{Fijo}}$$

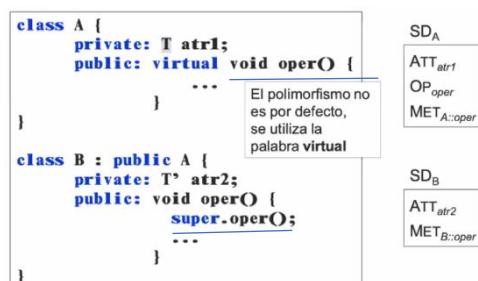
FD_{Empleado}	FD_{Fijo}
ATT_{nombre} PSA_{miEmp} $OP_{\text{getNombre}}$ $MET_{\text{Empleado}::\text{getNombre}}$	$ATT_{\text{nombre}}, ATT_{\text{sueldo}}$ PSA_{miEmp} $OP_{\text{getNombre}}, OP_{\text{getSueldo}}$ $MET_{\text{Empleado}::\text{getNombre}}, MET_{\text{Fijo}::\text{getSueldo}}$

CLASE 16/3

Redefinición de Operaciones

Cuando en la generación de un full descriptor se encuentra más de un método asociado a la misma operación, se dice que dicha operación está redefinida

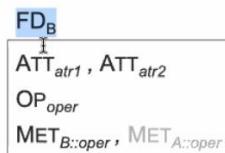
El método asociado a dicha operación será aquel que se encuentre en el segmento más próximo (en la jerarquía de generalizaciones) a la clase para la cual se está generando el full descriptor



- B es un hijo de A.
- El hijo puede acceder a los métodos del padre y a su vez agregar sus métodos.

- Si tengo un objeto A y se me ocupe invocar la operación **oper**. Pero si rememos un objeto de tipo B y queremos invocar **oper**, no sabe si es el **oper** del padre o el de él. Siempre termina eligiendo la mas cercana al objeto.
- Se utiliza la palabra virtual para ...
- y si la clase B solo utiliza el método del padre? Ahí no tendría un método para B?
CORRECTO

En el full descriptor de *B* el método asociado a *oper()* es el de la clase *B* (ocultando al heredado desde la clase *A*)



Sin embargo, el método heredado puede ser considerado en el full descriptor porque puede ser utilizado en el método que lo redefine

Sobrecarga

Es la capacidad que tiene un lenguaje de permitir que varias operaciones tengan el mismo nombre sintáctico, pero recibiendo diferente cantidad/tipo de parámetros

Ejemplos de sobrecarga:

```

void oper(int a, int b)   ↗
void oper(float a, float b)

```

La sobrecarga no es un concepto exclusivo de la orientación a objetos

Sobrecarga vs. Redefinición

La redefinición trata de la misma operación, con diferentes métodos

La sobrecarga trata de diferentes operaciones, con diferentes métodos

No son la misma operación

Importante concepto

Operación Abstracta

En una clase, una operación es abstracta si no tiene un método asociado

Tener una operación abstracta es condición suficiente para que una clase sea abstracta

Una clase puede ser abstracta aún sin tener operaciones abstractas

¿Para qué nos interesa tener una operación abstracta?

Nos interesa para que cada hijo le dé el método que quiera.

```

class Empleado {
    private: string nombre;
    public: virtual float getSueldo() = 0;
}

class Fijo : public Empleado {
    private: float sueldo;
    public: float getSueldo(){
        return sueldo;
    }
}

class Jornalero : public Empleado {
    private: float valorHora;
    int cantHoras;
    public: float getSueldo(){
        return valorHora*cantHoras;
}

```

Gracias a la **herencia**
es posible que una
operación esté en
más de una clase

Gracias al **polimorfismo**
es posible asociarle
métodos diferentes en
cada una de ellas

getSueldo() =0; Una operación abstracta y en este ejemplo lo tenemos así porque tenemos 2 tipos de empleados. Los dos hijos responden al a operación getSueldo.

Operación abstracta es como un descriptor, lo pone en un padre y sus hijos lo operan como ellos quieran.

Polimorfismo, la capacidad de redefinirla. Tenemos que ponerle virtual si o si para que se pueda redefinir, reescribir.

El =0; quiere decir que no da método.

Clase Abstracta

Algunas clases pueden ser abstractas:

Ningún objeto puede ser creado
directamente a partir de ellas

No son instanciables

Las clases abstractas existen
solamente para que otras hereden las
propiedades declaradas por ellas

```

class Empleado {
    private: string nombre;
    public: virtual float getSueldo() = 0;
}

class Fijo : public Empleado {
    private: float sueldo;
    public: float getSueldo(){...}
}

class Jornalero : public Empleado {
    private: float valorHora;
    int cantHoras;
    public: float getSueldo(){...}
}

```

Observación:
Empleado es una
clase abstracta por
tener todos sus
operaciones abstractas.
Debido a esto, todo
empleado es fijo ó
jornalero

INTERFAZ

Una interfaz "es un conjunto de operaciones al que se le aplica un nombre"

Una interfaz no define un estado para las instancias de estos elementos, ni tampoco asocia un método a sus operaciones

Es conceptualmente equivalente a un Tipo Abstracto de Datos

Este conjunto de operaciones caracteriza el (o parte del) comportamiento de instancias de clases:

De manera similar en la que un TAD caracteriza el comportamiento de instancias de sus implementaciones

Es un conjunto de operaciones genéricas que sirve para tener distintas clases.

Va a especificar el comportamiento de las subclases, pero no da métodos.

Una clase **realiza** una interfaz en forma análoga a cómo un tipo implementa un TAD

Cuando **C** realiza **I**, puede decirse que una instancia de **C**:

"Es de **C**" o "es un **C**" pero tambien que,

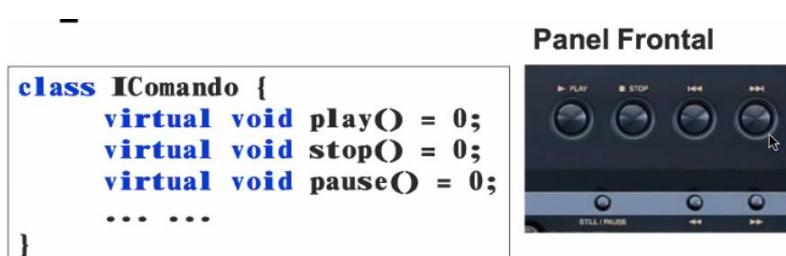
"Es de **I**" o "es un **I**"

Esto permite quebrar las dependencias hacia "las implementaciones" cambiándolas por una sola dependencia hacia "la especificación" (la interfaz)

NOTAS:

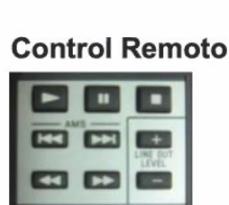
- Controladores se le llama a los que realizan la interfaz.
- Es un conjunto de operaciones abstracta.

EJEMPLO:



Tanto un "panel frontal" como un "control remoto" son "comandos"

Sabiendo operar un "comando" se puede operar tanto a un "panel frontal" como a un "control remoto"



Tanto el panel como el control, tiene las mismas operaciones.

Instancia Directa e Indirecta

Si un objeto es creado a partir del full descriptor generado para una cierta clase, C, entonces se dice que ese objeto es *instancia directa* de C

Además se dice que el objeto es *instancia indirecta* de todas las clases ancestras de C

Ejemplo: **Jornalero *j = new JornaleroO;**

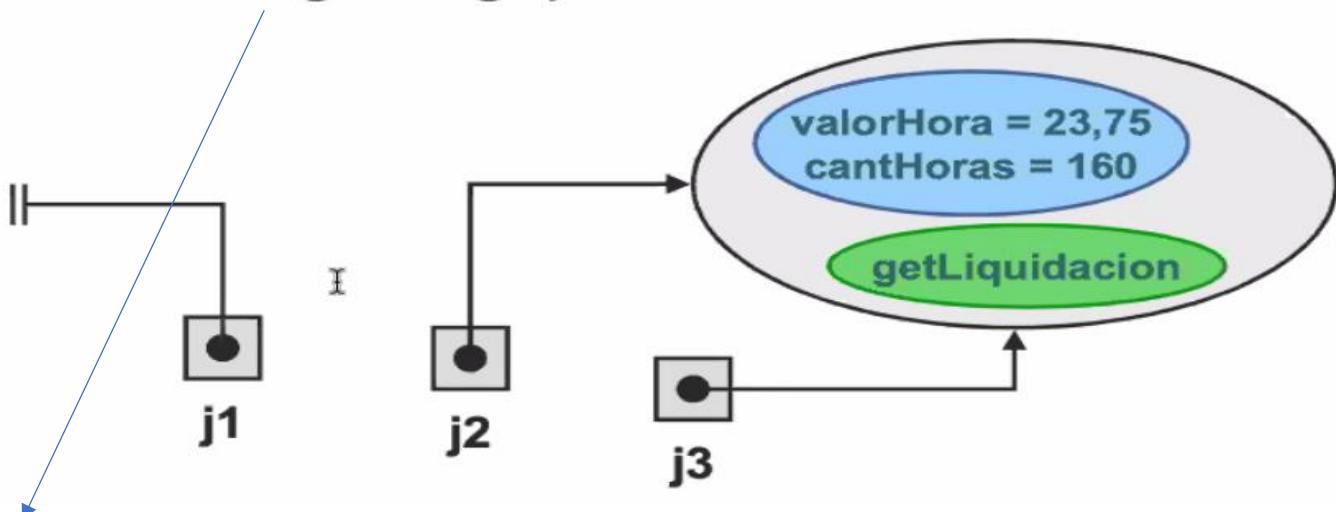
j es instancia directa de Jornalero

j es instancia indirecta de Empleado

Jornalero *j1 = null;

Jornalero *j2 = new JornaleroO; // attached

Jornalero *j3 = *j2; // attached



El primero dice que es void, es un método pero todavía no tiene nada, las otras dos si tienen un lugar en memoria.

El tipo estatico se define en tiempo de ejecución, es como lo creo. Y el tipo dinamico es el que

Invocación

Una invocación se produce al acceder a una propiedad de una instancia que sea una operación

El resultado es la ejecución del método que la clase de dicha instancia le asocia a la operación accedida (despacho)

Tipo Estático y Dinámico (2)



En situaciones especiales, el tipo dinámico difiere del tipo estático y se conoce en tiempo de ejecución

Este tipo de situación es en la que la referencia a un objeto es declarada como de una clase ancestral del tipo del objeto:

Lo cual es permitido por subsumption

Se cumple la siguiente relación entre los tipos de *obj*:

TipoDinamico(obj) <: TipoEstatico(obj)

EJEMPLO

```
Empleado *e = new Jornalero();
```

TipoEstatico(e) = Empleado

TipoDinamico(e) = Jornalero

```
Empleado *e;  
if (cond)  
    e = new Fijo();  
else  
    e = new Jornalero();
```

¿Cuál es el tipo dinámico de e?

Empleado *e todavía no pide memoria, después hace un código. el tipo estático es empleado. El tipo dinámico depende, no vamos a saber hasta el tiempo de ejecución. Lo pedimos genérico y después, lo instanciamos

Jornalero	Fijo	Jornalero
-----------	------	-----------	-----	-----	-----	-----	-----	-----	------

La idea es unificar, porque tiene un comportamiento similar.

Definimos las clases

```
Empleado * emps [100];
int cantEmpleados = 0;

Class Empleado {
    public:
        void virtual obtenerSueldo() = 0;
}

Class Fijo : public Empleado {
    public:
        void obtenerSueldo(){
            ...
        }
}

Class Jornalero : public Empleado {
    public:
        void obtenerSueldo(){
            ...
        }
}
```

Ahora hacemos un main

```
main (){
}
...
int sueldo;
for (int i = 0; i < cantEmpleados; i++){
    sueldo = emps[i]->obtenerSueldo();
    ...
}
```

Yo puedo tener un int cantidad de empleados que es un arreglo. Eso tiene que ir preguntando si el sueldo es de jornalero y fijo. Cada uno resuelve a su manera,

La operación **getLiquidacion()** declarada en **Empleado** es polimórfica porque es redefinida en **Fijo** y en **Jornalero**

```
Empleado *e;
if (cond)
    e = new Fijo();
else
    e = new Jornalero();
e->getLiquidacion();
```

Se está invocando a una operación polimórfica sobre un objeto (que será de clase **Fijo** ó **Jornalero**) mediante una referencia declarada como de tipo **Empleado** (clase ancestral de las anteriores)

En esta invocación debería despacharse $\text{MET}_{\text{Fijo}::\text{getLiquidacion}}$ ó $\text{MET}_{\text{Jornalero}::\text{getLiquidacion}}$

Utilizando la información estática se intentaría despachar $\text{MET}_{\text{Empleado}::\text{getLiquidacion}}$ (que en este ejemplo no existe!)

Para que en este tipo de casos el despacho sea realizado en forma correcta es necesario esperar a contar con la información del tipo real del objeto (tipo dinámico):

Eso se obtiene en tiempo de ejecución

El *despacho dinámico* es la capacidad de aplicar un método basándose en la información dinámica del objeto y no en la información estática de la referencia a él

La decisión de qué tipo de despacho emplear para una operación puede estar preestablecida en el propio lenguaje o definida estáticamente en el código fuente

En algunos lenguajes de programación el despacho es dinámico para cualquier operación (sea polimórfica o no)

En otros lenguajes:

Las invocaciones a operaciones polimórficas son siempre despachadas dinámicamente

Las invocaciones a operaciones no polimórficas son siempre despachadas estáticamente

Caso: Empresa asociada con Empleados

Responsabilidad: calcular el total de la liquidación de todos los empleados de la empresa

Responsable: la Empresa porque es quien dispone de la información necesaria para cumplir con la responsabilidad

```
class Empresa {  
    private: String ruc;  
    Set<Empleado> misEmps; //pseudotributo  
  
    public: float getLiquidacionTotal() {  
        float total = 0;  
  
        foreach(Empleado *e in misEmps) {  
            total = total + e->getLiquidacion();  
        }  
        return total;  
    }  
}
```

No existen las construcciones **Foreach**, **in** ni **Set** en C++

Despacha dinámicamente al método correcto, devolviendo el valor correcto

REALIZACION

Es una relación entre una especificación y su implementación
Una forma posible de realización se produce entre una interfaz y una clase

Se dice que una clase C realiza una interfaz I si C implementa todas las operaciones declaradas en I, es decir provee un método para cada una

Es posible tipar a un objeto (además de como es usual mediante la clase de la cual es instancia) también mediante *una* de las interfaces que su clase realiza

Por lo que si un objeto es declarado como de tipo I (en una lista de parámetros, como atributo, etc.), siendo I una interfaz, significa que ese objeto no es una instancia de I (lo cual no tiene sentido) sino que es instancia de una clase que realiza la interfaz I

```
class ControlRemoto : public IComando {  
    ... // algun atributo y pseudoatributo  
    // que defina el estado del CR  
    ... // alguna operacion adicional
```

public:

```
void playO {  
    ...  
}
```

```
void stopO {  
    ...  
}
```

... // implementacion del resto

Una interfaz puede ser entendida como la especificación de un *rol* que algún *objeto* debe desempeñar en un sistema

Un objeto puede desempeñar más de un rol:
Una clase puede realizar cualquier cantidad de interfaces

Un rol puede ser desempeñado por objetos de características diferentes:
Una interfaz puede ser realizada por cualquier cantidad de clases

```
class ControladorAudio {  
    ...  
  
    public: void controlarAudio(IComando *c) {  
        c->playO;  
        void operacion (IComando *);  
        ...  
        s->operacion(c1);  
        ...  
    }  
}  
  
IComando *c1 = new ControlRemotoO;  
IComando *c2 = new PanelFrontalO;  
  
ca->controlarAudio(c1); // invocación valida  
ca->controlarAudio(c2); // tambien valida
```

TENEMOS una operación que recibe un IComando, es algo muy genérico. Solo conoce IComando que es una interfaz.

```

Class Sistema {
public:
    void oper (IComando *);
...
};

Class IComando { // Interfaz
public:
    virtual void play() = 0;
    virtual void stop() = 0;
    virtual void pause() = 0;
}

Class ControlRemoto : public IComando { //Controlador, o sea la clase que realiza la Interfaz IComando
private:
    int x;
...
public:
    void play() {...};
    void stop() {...};
    void pause() {...};
...
}

```

- IComando es una interfaz, las iguala a cero.
- Controlador, es la clase que realiza la interfaz Icomando. Tiene que dar si o si una implementación de play, stop u pause. Lo resuelve de la manera que quiera.

```

main() {
    Sistema * s = new Sistema();
    IComando * c = new ControlRemoto();
    s->oper(c);

}

```

--o sea que usas el tipo dinamico para crear una instancia de una clase abstracta? CORRECTO

Este mecanismo permite abstraerse de la implementación concreta del objeto declarado

En lugar de exigir que dicho objeto presente una implementación determinada (es decir, que sea instancia de una determinada clase), se exige que presente un determinado comportamiento parcial (las operaciones declaradas en /)

Este comportamiento es implementado por una clase que realice la interfaz, y de la cual el objeto en cuestión es efectivamente instancia

Notar que en la definición previa se asume que la clase que realiza la interfaz es concreta. Es posible sin embargo que una interfaz sea realizada por una clase abstracta

En cuyo caso debe declarar todas las operaciones de la interfaz aunque no esté obligada a implementarlas a todas

Si C es abstracta y realiza la interfaz /, entonces un objeto declarado como de tipo / debe ser instancia de alguna subclase concreta de C (o de otra clase que realice la interfaz /)

Una clase para realizar siempre tiene que dar el código y si no lo hace si es abstracta tiene que venir una clase abajo que lo especifique y del código.

DEPENDENCIA

Es una relación asimétrica entre un par de elementos donde el elemento independiente se denomina *destino* y el dependiente se denomina *origen*

En una dependencia, un cambio en el elemento destino puede afectar al elemento origen

Las asociaciones, generalizaciones y realizaciones caen dentro de esta definición general

Pero son una forma más fuerte de dependencia

En esos casos la dependencia se considera asumida y no se expresa explícitamente

Asociación es el más fuerte, tipo de relación fuerte. Generalización y Realización relativamente iguales y Dependencia es la débil.

PRACTICO

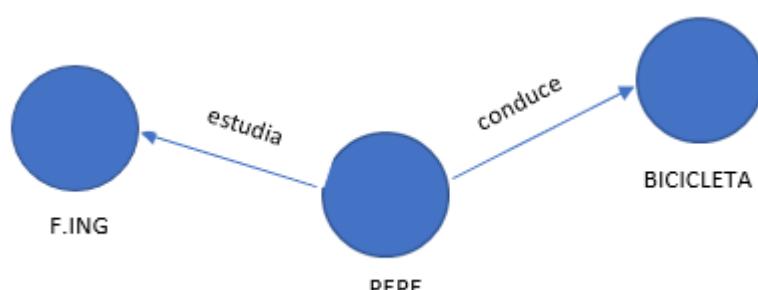
Ejercicio 1 (básico, imprescindible)

Exponer la metodología por usted utilizada hasta el momento para el desarrollo de software. Identificar los pasos seguidos en el proceso de desarrollo desde la especificación del problema hasta la obtención del programa; ¿qué herramientas utiliza para realizar una especificación formal de la realidad?, ¿y para especificar el diseño de la solución?, ¿y para realizar el pasaje de una a otra? ¿Con qué herramientas cuenta para gestionar la mantenibilidad, reusabilidad y evolutividad, cualidades deseables en los sistemas de software?

Ejercicio 2 (básico, imprescindible)

Intentar identificar objetos y links en la siguiente realidad sobre Pepe y su vida cotidiana. ¿Qué abstracciones pueden realizarse para dicha realidad?

Debido al aumento del crudo y la consecuente suba del boleto montevideano, Pepe ha tenido que tomar la decisión de conducir su bicicleta montaña para ir a estudiar a la Facultad de Ingeniería.



Ejercicio 3 (básico, imprescindible)

Recientemente los restaurantes han implementado un servicio de atención al cliente directamente en su vehículo. El mecanismo de funcionamiento de este servicio es el siguiente.

El cliente se aproxima a la ventanilla y le dice al recepcionista su pedido. El recepcionista le indica al cocinero que准备 la comida solicitada. Le indica al cajero que准备 la cuenta y le pide al cliente que pase a la siguiente ventanilla. El cliente llega a la segunda ventanilla y le paga al cajero, quién, luego de cobrar, le indica que siga a la siguiente ventanilla en donde el cocinero le entrega el pedido.

Modelar la realidad como una interacción de objetos. Abstraer los conceptos presentes en dicha realidad.

Hoy seguimos con el práctico 1

Ejercicio 3 (medio, imprescindible)

- a) Implementar en C++ una clase que represente a los conjuntos de enteros, utilizando como estructura de datos arreglos dinámicos. La clase deberá proveer al menos las siguientes operaciones:

```
agregar      : SetInt x int -> SetInt
remover      : SetInt x int -> SetInt
union        : SetInt x SetInt -> SetInt
diferencia   : SetInt x SetInt -> SetInt
interseccion : SetInt x SetInt -> SetInt
pertenece    : SetInt x int -> bool
esVacio      : SetInt -> bool
cantidadElem : SetInt -> int
esIgual       : SetInt x SetInt -> bool
```

```
class SetInt {
    private:
        int enteros [MAX_ENTEROS];
        int tope;
    public:
        SetInt();
        void agregar(int);
        void remover(int);
        SetInt union_ (SetInt);
        SetInt diferencia (SetInt);
        SetInt interseccion (SetInt);
        bool pertenece (int);
        bool esVacio ();
        unsigned cantidadElem ();
        bool esIgual(SetInt);

};
```

```
void SetInt::remover(int n) {
    if (this->pertenece(n)){
        int i;
        for (i = 0; i < this->tope; i++){
            if (this->enteros[i] == n){
                this->enteros[i] = this->enteros[this->tope-1];
                this->tope--;
                break;
            }
        }
    }
}
```

```
SetInt SetInt::diferencia (SetInt b){
    SetInt res();
    int temp;
    for (int i = 0; i < this.tope; i++){
        temp = this.enteros[i];
        if (!b.pertenece(temp))
            res.agregar(temp);
    }
    return res;
}
```

```
bool SetInt::esVacio (){
    return this->tope == 0;
}
```

```
SetInt SetInt::union_ (SetInt b) {
    SetInt res();
    // Como agregar ya está contemplando que no se repitan, le mandamos palo y palo
    for (int i=0; i < this.tope; i++)
        res.agregar(this.enteros[i]);
    for (int j=0; j < b.cantidadElem; j++)
        res.agregar(b.enteros[j]);
    return res;
}
```

```
bool esIgual(SetInt b){
    SetInt a_b = this.diferencia(b);
    SetInt b_a = b.diferencia(this);
    SetInt la_union = a_b.union_(b_a);
    return la_union.esVacio();}
}
```

Especificación de Requerimientos

La Especificación de Requerimientos es un insumo fundamental en el desarrollo de software:

Es la principal fuente de información a partir de la cual se diseña, implementa y testea el sistema

Es uno de los aspectos más delicados de un proyecto:

Es algo complejo de obtener

De su correctitud depende el éxito del proyecto

Representa un “contrato” con el usuario

No se genera por completo al inicio del proyecto, sino incrementalmente

Suele presentarse como la agregación de diferentes artefactos

Tipos de Requerimientos

Un requerimiento es una condición o capacidad que un sistema debe cumplir

Requerimiento No Funcional:

Expresa una propiedad o calidad que el sistema debe presentar

También restricciones físicas sobre los funcionales

Requerimiento Funcional:

Expresa una acción que debe ser capaz de realizar el sistema

Especifica comportamiento de entrada/salida

Requerimientos No Funcionales

Los requerimientos no funcionales suelen referir a:

Usabilidad: factores humanos, ayuda, documentación

Confiabilidad: frecuencia de fallas, tiempo de recuperación

Performance: tiempo de respuesta, tasa de procesamiento, precisión, capacidad de carga

Soportabilidad: adaptabilidad, mantenibilidad, configurabilidad, internacionalización

Interfaces: restricciones en la comunicación con sistemas externos

Restricciones: en el uso de

Sistemas o paquetes existentes

Plataformas

Lenguajes de programación

Ambientes de desarrollo

Herramientas (sistemas de bases de datos, middleware, etc.)

Requerimientos Funcionales

Los requerimientos funcionales se expresaban en términos de "funciones del sistema"

Una función del sistema es algo puntual que el sistema debe hacer

Técnica básica: Si X es una función del sistema, entonces la frase "*El sistema debe hacer X*" tiene que tener sentido

Casos de Uso

El enfoque de casos de uso está basado en la noción de **actor**

Un actor es un agente externo (humano o no) que interactúa directamente con el sistema

Un caso de uso narra la historia completa (junto a todas sus variantes) de un conjunto de actores mientras usan el sistema:

La historia termina cuando uno de los actores (el principal) logra su objetivo y obtiene un resultado de valor

Casos de Uso (3)

Un caso de uso se compone de:

Nombre que identifica al caso de uso

Actores participantes en el caso de uso

Sinopsis que describe brevemente su objetivo

Curso típico de eventos que narra la "historia" más común de los actores durante el uso del sistema

Cursos alternativos de eventos que narran las variantes de uso del sistema

Casos de Uso (5)

Método básico (variable según el avance):

1. Detectar actores
2. Identificar algunos casos de uso (detectando objetivos y necesidades de actores)
3. Especificarlos en alto nivel
4. Examinarlos y expandir algunos de ellos
5. ...

A medida que se avanza en el desarrollo se detectan nuevos casos de uso y se especifican otros ya detectados

Casos de Uso (2)

Los casos de uso son la herramienta más aplicada para la especificación de requerimientos funcionales

Por ser expresados textualmente resultan simples de comprender (hasta para personal no-técnico)

Por estar orientados a los objetivos de los actores (y al camino hacia su obtención):

Son intuitivos

Propician la completitud de especificación

Casos de Uso (4)

Los casos de uso no suelen especificarse con todo detalle de una sola vez

Esto se realiza en forma gradual y posterior a la identificación de actores:

Los actores son más fáciles de identificar y sus necesidades son las que dan lugar a los casos de uso

Formas posibles de un caso de uso:

Identificado o detectado: solo su nombre y actores participantes

Especificado en alto nivel: se incorpora una sinopsis

Especificado en forma expandida: se incorpora la "historia" de uso y sus variantes

Casos de Uso (6)

Sobre el principio del proyecto se tiende a buscar y especificar los casos de uso más importantes

Los casos de uso se usan además como criterio de partición del problema en un proceso iterativo e incremental:

En una iteración se desarrolla "uno a la vez"

Los incrementos no refieren a "partes" físicas sino a conjuntos de funcionalidades

Ejemplo (Formato Expandido)

Caso de Uso: Realizar una compra

Actores: Cajero

Sinopsis: Un cliente llega a la caja con artículos para comprar. El cajero registra los artículos y recibe el pago. Al finalizar, el Cliente se retira con los artículos.

Escenario Típico:

1. El Cliente llega a la caja con artículos para comprar.
2. El Cajero comienza un nueva venta.
3. El Cajero ingresa el identificador del artículo.
4. El Sistema registra el artículo y presenta su descripción, precio y subtotal.
El Cajero repite los pasos 3 y 4 hasta terminar los artículos.
5. El Sistema presenta el total con los impuestos incluidos.
6. ...

¿Qué sigue después?

Una vez detectado y especificado el conjunto inicial de casos de uso:

El equipo de desarrollo está listo para analizarlos, diseñar una solución para ellos e implementarlos

Mientras tanto el equipo de analistas avanza en la detección y especificación de otros casos de uso

Iterativo autoincrementar (Detectamos casos los realizamos)

Objetivos

Modelar el dominio del problema:

Para comprender mejor el contexto del problema

Para obtener una primera aproximación a la estructura de la solución

Especificificar el comportamiento del sistema:

Para contar con una descripción más precisa de qué es lo que se espera del sistema

Modelado del Dominio

Consiste en encontrar y describir los conceptos en el dominio de la aplicación

Durante esta actividad se construye el **Modelo de Dominio**

En él se incluyen todos los elementos que se definen durante esta actividad

Comportamiento del Sistema

Consiste en:

Entender cada caso de uso en términos de intercambios de mensajes entre los actores y el sistema

Especificar el comportamiento de cada uno de esos mensajes (pero sin decir cómo funcionan)

Durante esta actividad se completa el **Modelo de Casos de Uso**

En él se incluyen todos los elementos que se definen durante esta actividad

TEORICO MODELADO DEL DOMINO ANALISIS

Contenido

Introducción
Modelo de Dominio
Conceptos
Asociaciones
Atributos
Generalizaciones
Otros elementos
Restricciones

M.D: Es descompones el problema en conceptos

Modelo de Dominio

Está enfocado en conceptos del dominio y no en entidades de software

Contenido:

Introducción: Breve descripción que sirve como introducción al modelo

Conceptos: Clases que representan conceptos significativos presentes en el dominio

Tipos: Data types que describen propiedades de las clases que representan conceptos

Relaciones: Relaciones de asociación o generalización entre las clases que representan conceptos

Restricciones: Expresiones que restringen las posibles instancias de los conceptos del modelo

Diagramas: Representaciones (usualmente uno solo) de conceptos, tipos y relaciones presentes en el modelo

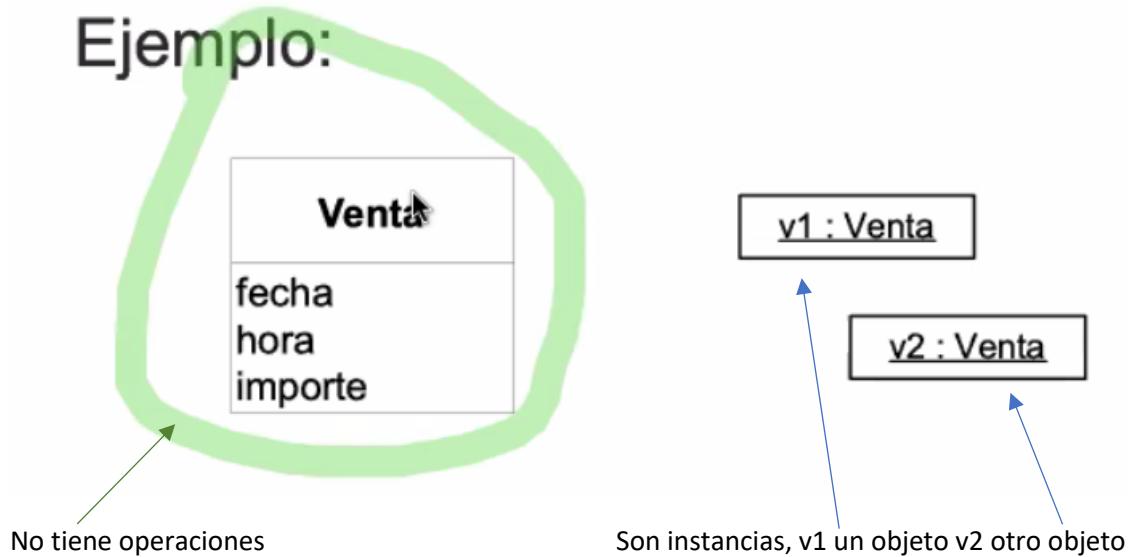
n



Conceptos

Un concepto es una idea, cosa o elemento de la realidad o problema que se está modelando.

Ejemplo:



Conceptos Identificación de Conceptos

Es muy común omitir conceptos en esta fase (identificación) que pueden ser descubiertos en una fase o etapa posterior

Al descubrirlos se los agrega al Modelo de Dominio

Es posible encontrar conceptos interesantes que no tengan atributos (que tengan un rol de comportamiento más que de información)

Lista de categorías de conceptos:

Consiste en repasar la lista de categorías de conceptos buscando los conceptos del dominio del problema que apliquen a cada categoría

Categoría	Ejemplo
Objetos físicos o tangibles	Avión
Descripciones de cosas	DescripcionVuelo
Lugares	Aeropuerto
Transacciones	Reserva
Roles	Piloto

Lista de categorías de conceptos (cont.)

La lista se puede continuar con:

contenedores de cosas

cosas contenidas en contenedores

sistemas externos

sustantivos abstractos

I

organizaciones

eventos

reglas y políticas

catálogos

registro de asuntos financieros o legales

servicios e instrumentos financieros

Identificación de sustantivos

Se identifican los sustantivos de una descripción textual del problema (visión del problema y/o casos de uso) y se los considera como conceptos o atributos candidatos

No es posible realizar esta actividad en forma totalmente automática

El lenguaje natural es ambiguo

No todo sustantivo refiere a un concepto significativo

Sugerencias

¶ Cómo crear un Modelo de Dominio:

1. Listar los conceptos candidatos usando cualquiera de las dos técnicas presentadas (o una combinación de ambas)
2. Incluirlos en el Modelo de Dominio
3. Agregar las asociaciones necesarias para registrar relaciones que necesiten ser preservadas
4. Agregar los atributos necesarios para satisfacer los requerimientos de información

Sugerencia: Generar y mantener el diagrama en paralelo

Sugerencias (2)

■ Nombres y modelado

La estrategia del cartógrafo se aplica tanto a la construcción de mapas y a la de Modelos de Dominio:

Usar nombres que existan en el territorio

Excluir características irrelevantes

No incluir cosas inexistentes

■ Sugerencias (3)

■ Granularidad de la especificación

Durante el proceso de modelado, es mejor sobre-especificar con muchos conceptos de granularidad fina, que sub-especificar

El costo de eliminar un concepto que resultó innecesario es menor que el de agregar uno que fue omitido

Siempre es posible agregar o eliminar conceptos durante el proceso de modelado

Sugerencias (4)

Error común al identificar conceptos

El error más común al crear un Modelo de Dominio es representar algo como un atributo cuando debió ser un concepto

Si no se piensa en un concepto X básicamente como un número, un texto o un booleano (o data types en general) entonces X probablemente sea un concepto

En caso de duda, representarlo como un concepto

ERROR COMUN

Tenemos dos casos

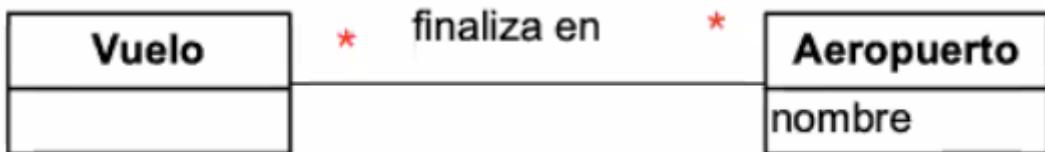
Error común al identificar conceptos (cont.)



En el primer caso tenemos, no tenemos un concepto aeropuerto. Si necesitamos saber que vuelos llegaron a MVD o algo así no podemos saberlo.

```
v1 = new Vuelo("MVD");
```

No pasa con el ejemplo de abajo si nos permite obtener más información



Sugerencias (6)

Supóngase la siguiente situación:

Una instancia de “Producto” representa a un producto físico en una tienda

Un producto tiene un número de serie, una descripción, un precio y un código, que no aparecen en ninguna otra parte

Los que trabajan en la tienda “no tienen memoria”

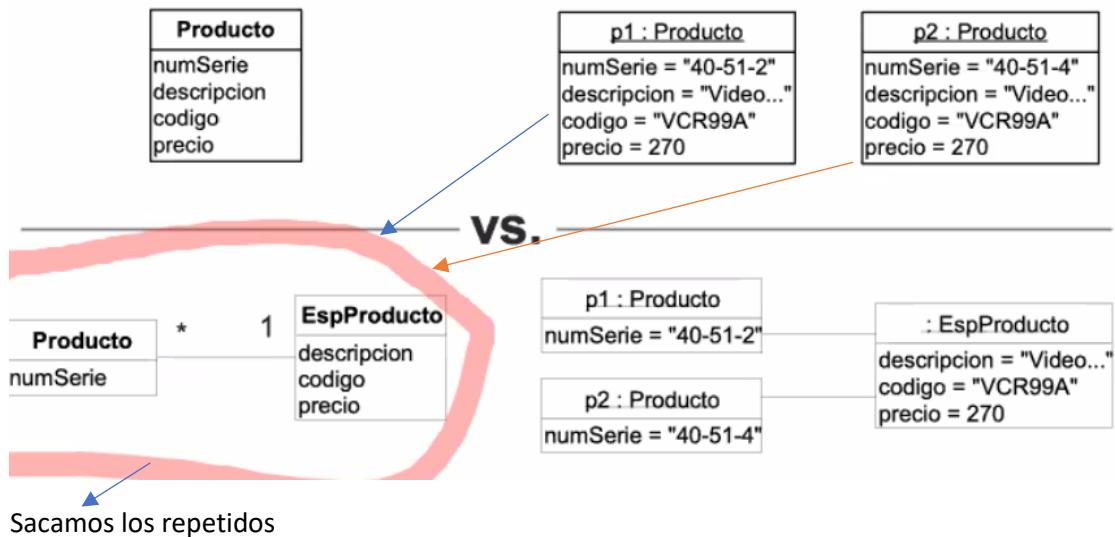
Cada vez que un producto físico es vendido, la correspondiente instancia de “Producto” es eliminada del sistema

p1: Producto
class Producto{
private:

string desc;
float precio;
int cod;

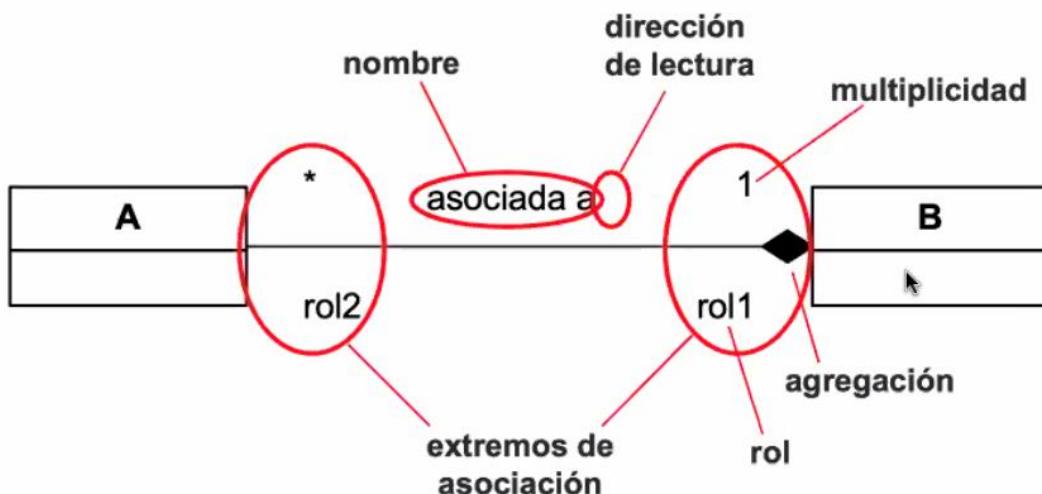
Especificaciones y descripciones (cont.)

Ejemplo:



Asociaciones Notación

La asociación se lee: “A asociada a B”



agregación: composición

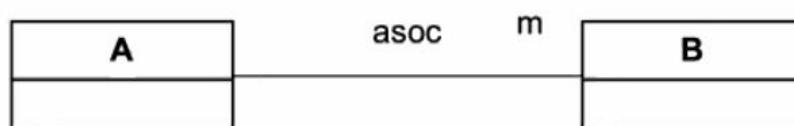
Notación - Multiplicidades

La multiplicidad limita la cantidad de veces que una instancia determinada está conectada a otras a través de una asociación

Eso se indica en el extremo de asoc. opuesto

$\text{asoc} \subseteq A \times B$

Indica la cantidad de instancias de B que pueden conectarse con un 'A' cualquiera



Se expresa como un subconjunto de los naturales (subrango o enumerado)
 $m \subseteq \mathbb{N}$ tal que $\max(m) > 0$

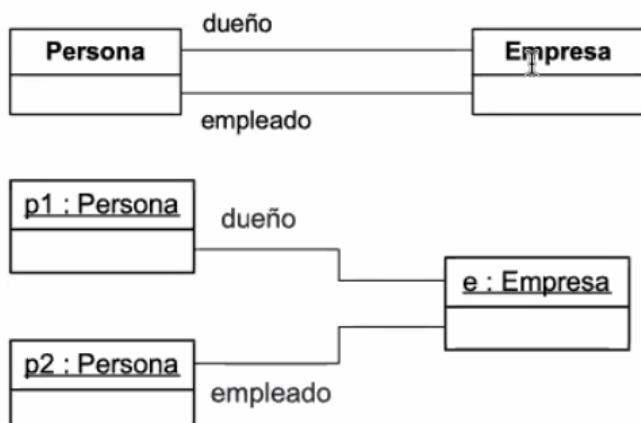
Ejemplos:

- * Cualquier cantidad (cero o más)
- 1..* Al menos uno (uno o más)
- 0..1 Opcionalmente uno (cero o uno)
- 5 Exactamente cinco
- 3,5,8 Exactamente tres, cinco u ocho

Notación - Roles

Especifican el papel que juegan las clases en una asociación

Pueden ser necesarios para eliminar ambigüedades



Notación - Restricciones

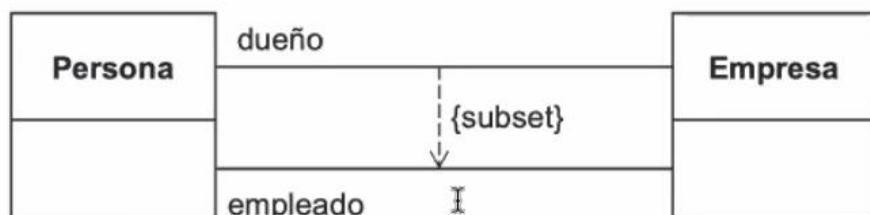
En ocasiones es necesario especificar que existe una restricción entre dos asociaciones
Por ejemplo, que un par de instancias solo estén conectadas mediante una asociación



De esta forma una persona no puede ser dueño y empleado de la misma empresa

Notación – Restricciones (2)

Otro ejemplo, si dos instancias están conectadas por una asociación, también lo deben estar por otra asociación



De esta forma una persona que sea dueña de la empresa tiene que ser empleado

Notación – Restricciones (3)

Es posible también indicar que existe un orden entre las instancias con las cuales otra instancia está relacionada



Aquí interesa el orden de los alumnos en cada curso (por ejemplo por cédula)

Notación - Agregación

Es una forma más fuerte de asociación

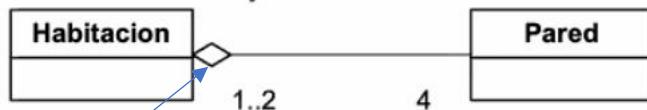
Significa que un elemento es parte de otro

Existen dos variantes

Agregación compartida (agregación)

Agregación compuesta (composición)

Agregación compartida



Débil blanca, fuerte en negro.

Notación - Agregación

Agregación compuesta

Un elemento es exclusivo del compuesto
(máximo de la multiplicidad es 1)

Generalmente una acción sobre el compuesto
se propaga a las partes (típicamente en la
destrucción)



Atributos

Es necesario identificar aquellos atributos que permitan satisfacer los requerimientos de información

Un atributo se entiende como un *data value* de un objeto

El tipo de un atributo es un *data type*

Atributos Notación

Al mostrar un atributo es necesario especificar al menos su nombre

Propiedades opcionales

Tipo, multiplicidad, valor inicial, visibilidad, etc.

Persona
nombre
telefono
edad

Representación
mínima

Persona
-nombre[1] : String
+telefono[*] : String
-edad[1] : Integer = 0

Representación
completa

Atributos Notación (2)

Alcance de atributos

Empleado
sueldo

De instancia

Empleado
<u>sueldo</u>

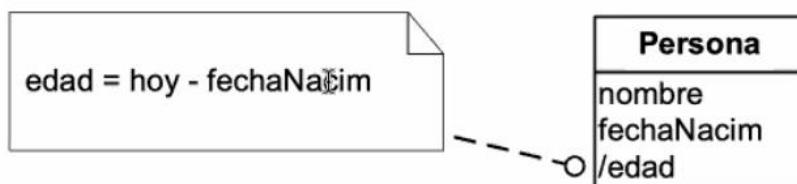
De clase

De instancia, cada instancia va a tener la suya. Y De clase no requiere que inicialices un objeto para acceder a el. Capas que no existe ninguna instancia pero el atributo subrayado si esta.

Notación (3)

Un atributo (o cualquier elemento) que sea derivable se marca con un '/'

Lo usual es adjuntarle una nota especificando la forma en que se calcula



Sugerencias

No utilizar atributos como clave foránea

Los atributos no deben ser utilizados para relacionar elementos del modelo



VS.



Sugerencias (2)

Tipos primitivos y no-primitivos

Los tipos de los atributos son en general tipos primitivos (Integer, String, Real, etc.)

De ser necesario es posible definir tipos no-primitivos para un problema



Generalizaciones

Es posible especificar variantes de un concepto cuando

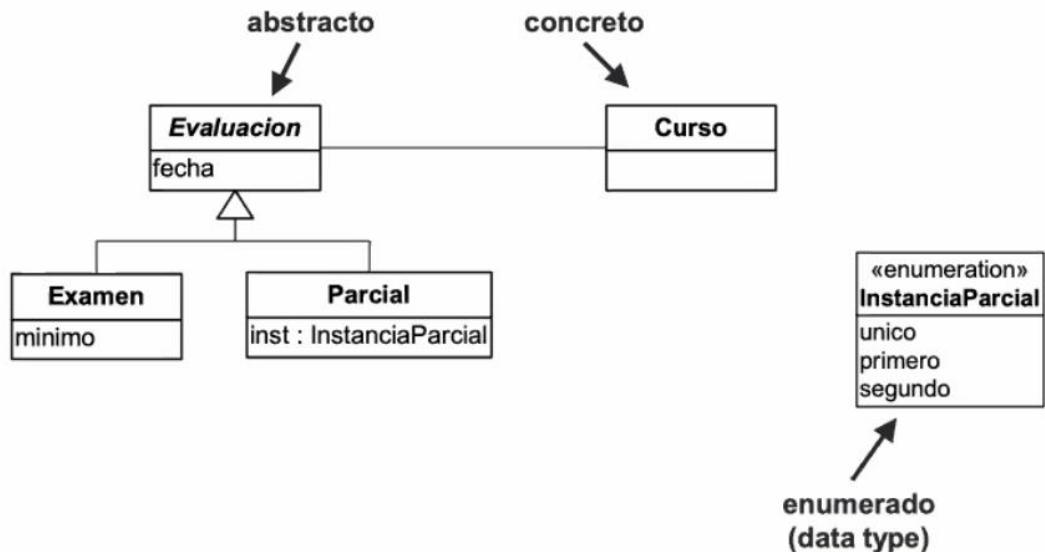
Los subtipos potenciales representan variantes interesantes de un cierto concepto

Un subtipo es consistente con su supertipo (se aplica subsumption)

Todos los subtipos tienen atributos comunes que pueden ser factorizados en el supertipo

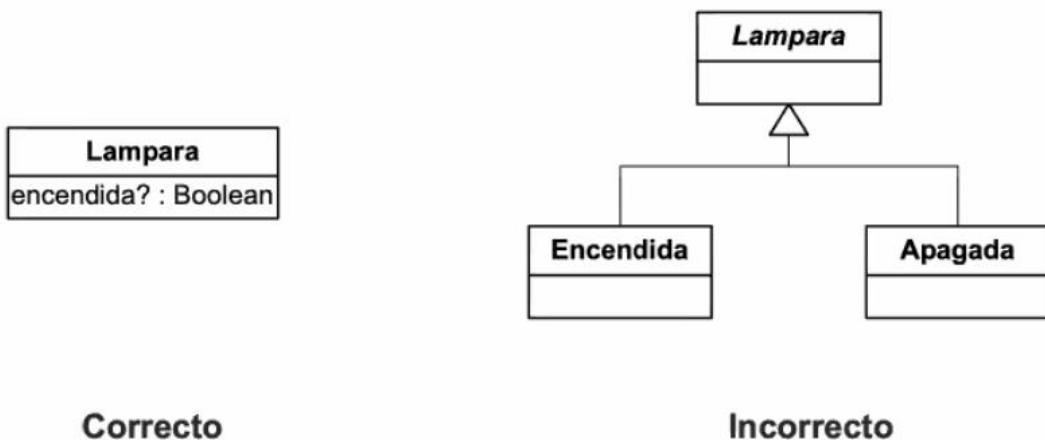
Todos los subtipos tienen asociaciones comunes que pueden ser factorizadas en el supertipo

Notación



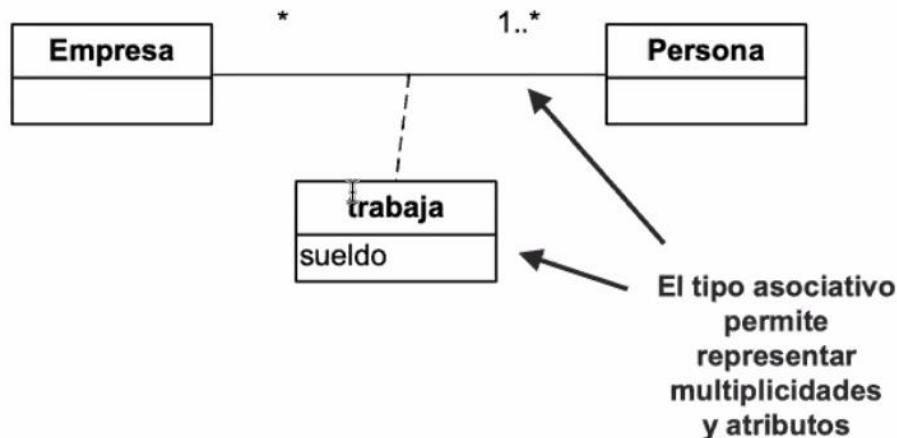
Sugerencias (2)

Modelado de estados (cont.)



Las cosas que cambiar se manejan como atributos.

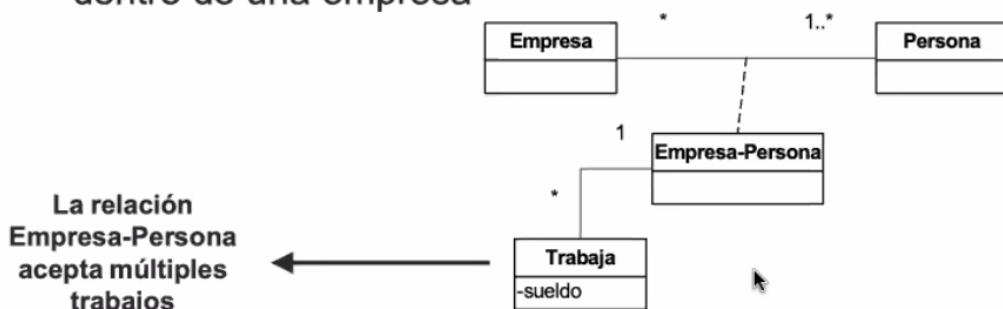
Tipos Asociativos Notación



Tipos Asociativos Modelado Avanzado

¿Cómo se modela cuando se necesitan múltiples instancias de la misma clase de asociación para un mismo par de instancias?

Ejemplo: registrar todos los sueldos de una persona dentro de una empresa



[Errores Comunes (2)]

Omisión de una *especificación* para las instancias

Incluir elementos del diseño
(interfaces, dependencia, etc.)

Representar asociaciones como atributos (uso de claves foráneas)

Redundancia y sobre especificación

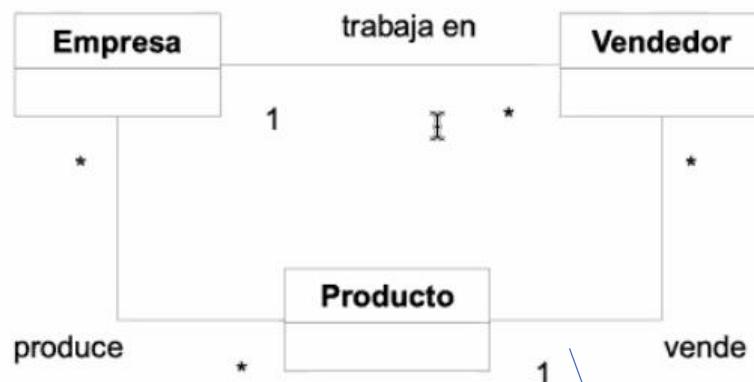
Especificar el tipo de estructura en una multiplicidad de *

[Restricciones]

Es muy común el hecho de que un Modelo de Dominio no alcance a representar exactamente la realidad planteada

Existen casos donde un modelo representa fielmente la mayoría de los aspectos de la realidad sin embargo permite otros que no son deseables

Restricciones Motivación



**El modelo representado por este
diagrama: ¿Refleja fielmente la realidad?**

p1: Producto
p2: Producto
e1: Empresa
e2: Empresa
v1: Vendedor

v1 — e1
v1 — p2
p1 — e1
p2 — e2

EJEMPLO

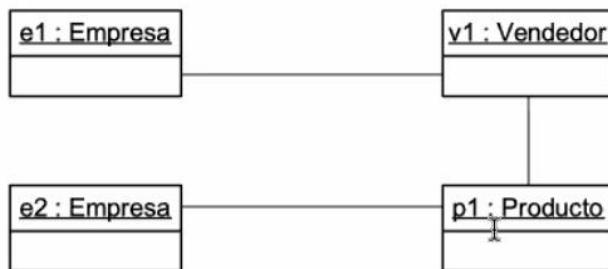
vende un producto de una
empresa en la cual no trabaja



Restricciones Motivación (2)

Permite o considera como válidos casos como:

“Un vendedor vende un producto producido por una empresa para la cual él no trabaja”



¡Todas las multiplicidades están satisfechas!
(esta configuración de objetos es válida respecto al Modelo de Dominio)

La empresa de v1 (o sea e1) debería producir el producto que él vende (o sea p1), o v1 debería trabajar en la empresa e2

PARA RESOLVERLO USAMOS RESTRICCIONES

Restricciones Adjuntar Restricciones

Otra alternativa al problema es la imposición de restricciones (en particular invariantes)

Un invariante es un predicado que expresa una condición sobre los elementos del Modelo de Dominio y que siempre debe ser verdadero

Cuando es evaluado contra una cierta configuración de objetos dando un resultado de *falso* significa que la configuración de objetos no es válida

UML no especifica el modo en que un invariante deba ser expresado

Puede utilizarse notación informal o formal

Restricciones

Invariantes - Informal

Los invariantes pueden ser expresados informalmente en lenguaje natural

Un ejemplo de esto puede ser

Invariante:

“Todo vendedor debe vender un producto que sea producido por la empresa para la cual trabaja”

Restricciones

Invariantes - Informal (2)

Ventajas

Es “entendido” por todos

Desventajas

Es ambiguo: una restricción compleja puede

Ser difícil de escribir y/o leer

Fácilmente dar lugar a confusiones

No puede ser procesado en forma automática

Restricciones Invariantes - Formal

UML contiene un lenguaje que fue diseñado específicamente para la especificación de este tipo de restricciones

Es relativamente simple e intuitivo

Este lenguaje es el *Object Constraint Language*

Ejemplo:

```
context Vendedor inv:  
    self.producto.empresa->includes(self.empresa)
```

Restricciones Invariantes - Formal (2)

Ventajas:

Una restricción tiene un significado único y preciso

Puede ser procesada en forma automática

Desventajas:

El lenguaje a utilizar puede resultar extremadamente complejo

Requiere el aprendizaje de las construcciones del lenguaje

Restricciones Habituales

Unicidad de Atributos (Identificación de Instancias)

Un atributo tiene un valor único dentro del universo de instancias de un mismo tipo (una instancia es identificada por ese valor)

Dominio de Atributos

El valor de un atributo pertenece a cierto dominio

Integridad Circular

No puede existir circularidad en la navegación

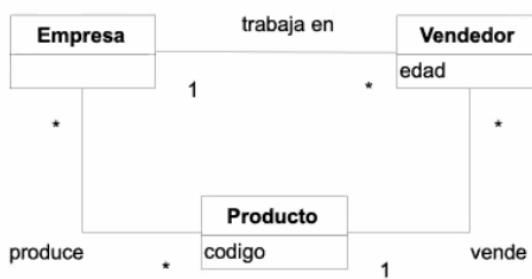
Atributos Calculados

El valor de un atributo es calculado a partir de la información contenida en el dominio

Reglas de Negocio

Invariante que restringe el dominio del problema

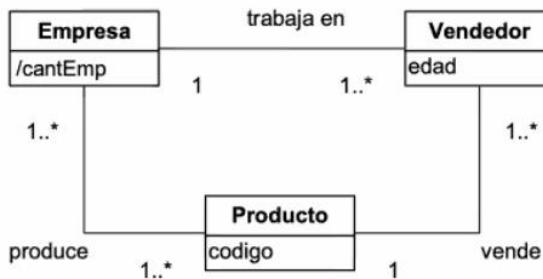
Restricciones Ejemplos



Unicidad de Atributos (Invariante)

“No hay dos productos con el mismo código (el código identifica al producto)”

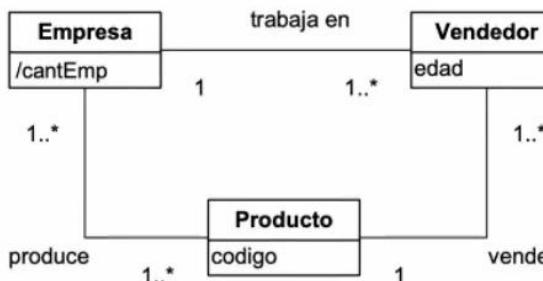
Restricciones Ejemplos (2)



Dominio de Atributos (Invariante)

“En la empresa no puede haber vendedores mayores de 65 años de edad”

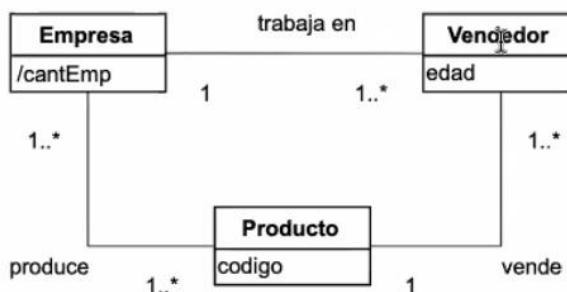
Restricciones Ejemplos (3)



Integridad Circular (Invariante)

“Un vendedor no puede vender productos de una empresa en la que no trabaja”

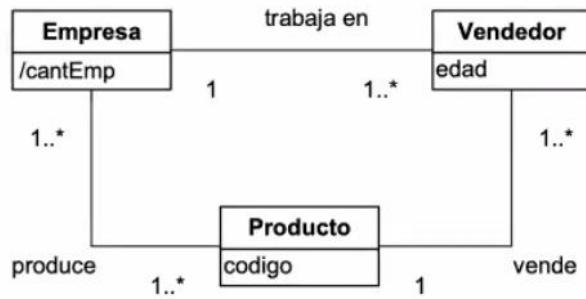
Restricciones Ejemplos (4)



Atributos Calculados (Invariante)

“El atributo cantEmp es la cantidad de empleados de la empresa”

Restricciones Ejemplos (5)



Reglas de Negocio (Invariante)

“Ningún vendedor menor de 30 años puede vender el producto de código X”

Ejercicio 6 (básico, imprescindible)

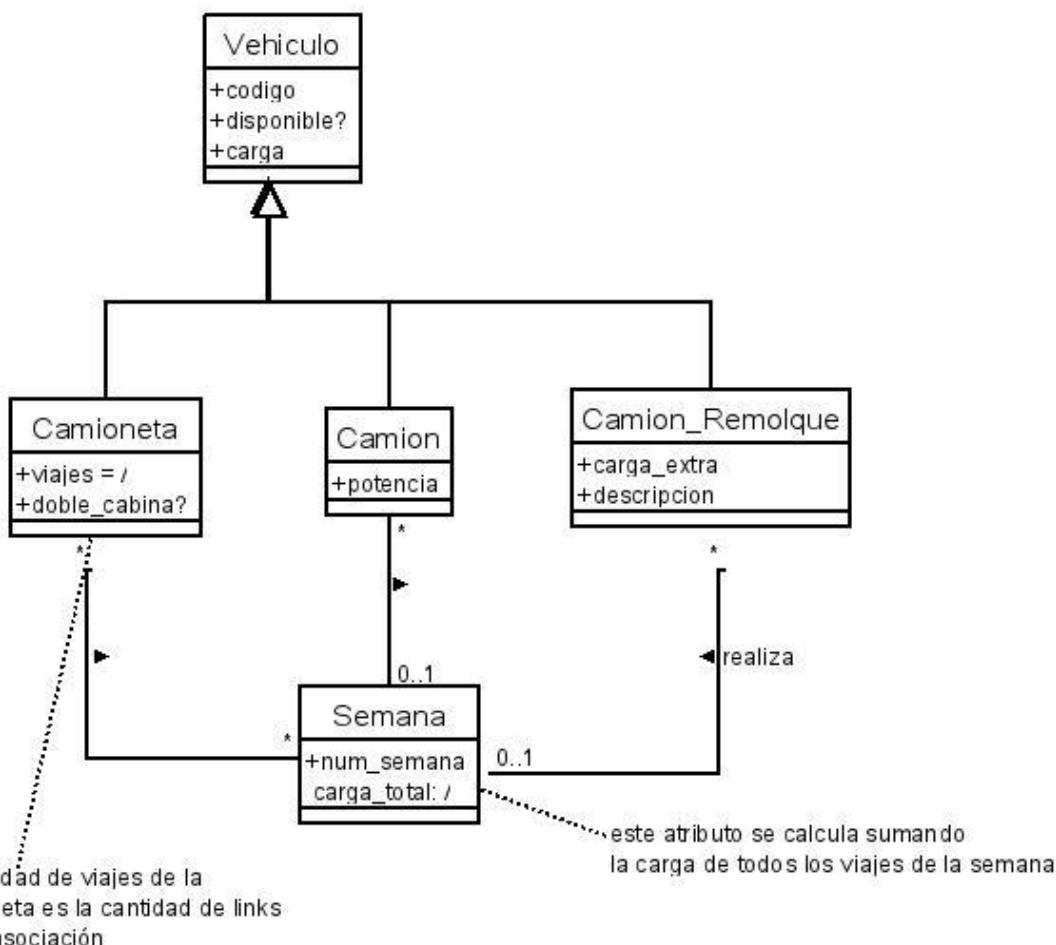
Se desea modelar el funcionamiento de una compañía de transporte de cargas la cual mantiene una flota de vehículos. De cada vehículo se sabe su código (que lo identifica), si está disponible y su capacidad de carga. Un vehículo puede ser:

- una camioneta, de las que se sabe los viajes por semana que puede hacer, y si es doble cabina o no,
- un camión, de los que se conoce su potencia, y
- un camión con remolque, de los que se sabe la capacidad de carga extra del remolque y una descripción del mismo.

Un camión (con remolque o no) puede hacer un solo viaje por semana. El modelo debe reflejar el estado de la flota en una semana particular ya que semanalmente, la compañía desea estimar la capacidad de carga total de su flota (la suma de la cantidad de carga por semana para todos sus vehículos disponibles).

Construir el Modelo de Dominio y presentarlo en un diagrama utilizando UML.

EJERCICIO 6



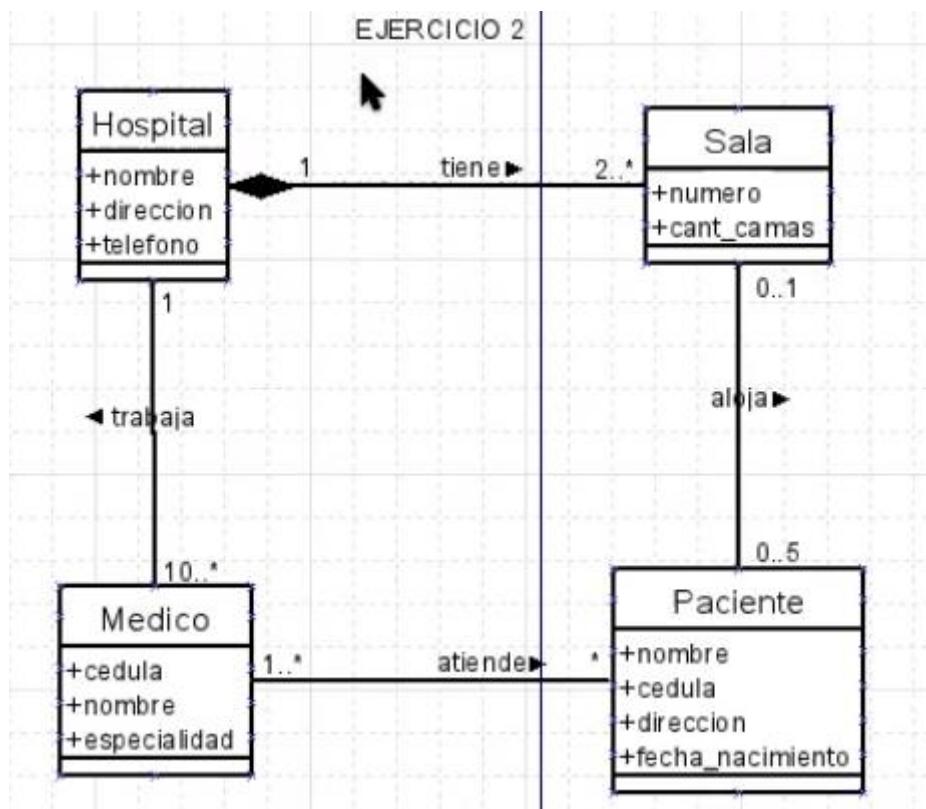
Ejercicio 2 (básico, imprescindible)

En la construcción de un sistema de información para el control hospitalario se relevaron los siguientes conceptos:

- Hospital, con los datos nombre, dirección y teléfono.
- Sala, con los datos número y cantidad de camas.
- Médico, con los datos cédula de identidad, nombre y especialidad.
- Paciente, con los datos cédula de identidad, nombre, dirección y fecha de nacimiento.

Por otra parte, las relaciones relevadas entre dichos conceptos son:

- Cada hospital tiene varias salas. Todas y cada una de ellas pertenecen a un hospital (y solo a uno).
- Cada médico trabaja en un único hospital. Todo hospital tiene al menos 10 médicos.
- Un paciente puede estar internado; si lo está, estará en una sala (y sólo en una).
- La capacidad máxima de camas que puede tener una sala es de cinco pacientes.



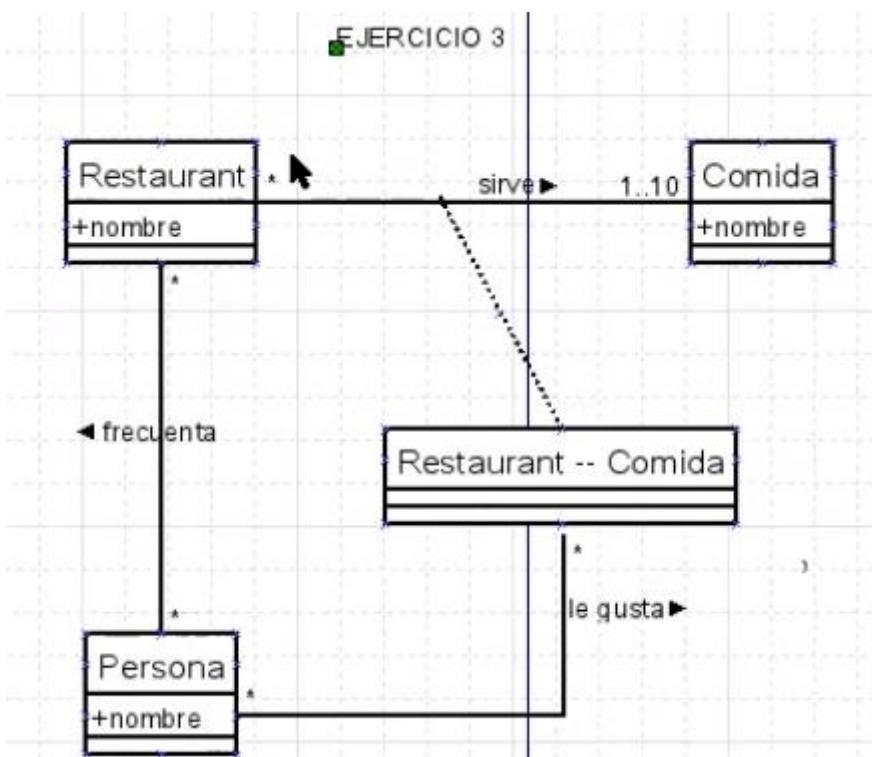
Ejercicio 3 (básico, imprescindible)

Se tiene la siguiente información:

- Las personas frecuentan algún restaurante.
- A las personas les gustan distintas comidas.
- Los restaurantes sirven comidas.

Construir el Modelo de Dominio y presentarlo en un diagrama utilizando UML, teniendo en cuenta las siguientes restricciones:

- Un restaurante no sirve más de 10 comidas.
- Una persona frecuenta varios restaurantes.
- A una persona no le gusta una comida por sí sola sino cómo la sirven en determinados restaurantes, aunque puede no gustarle ninguna.
- Una comida servida por un restaurante puede no gustarle a ninguna persona.



REPASO DE PARCIAL