Programación Avanzada

Conceptos Básicos de Orientación a Objetos

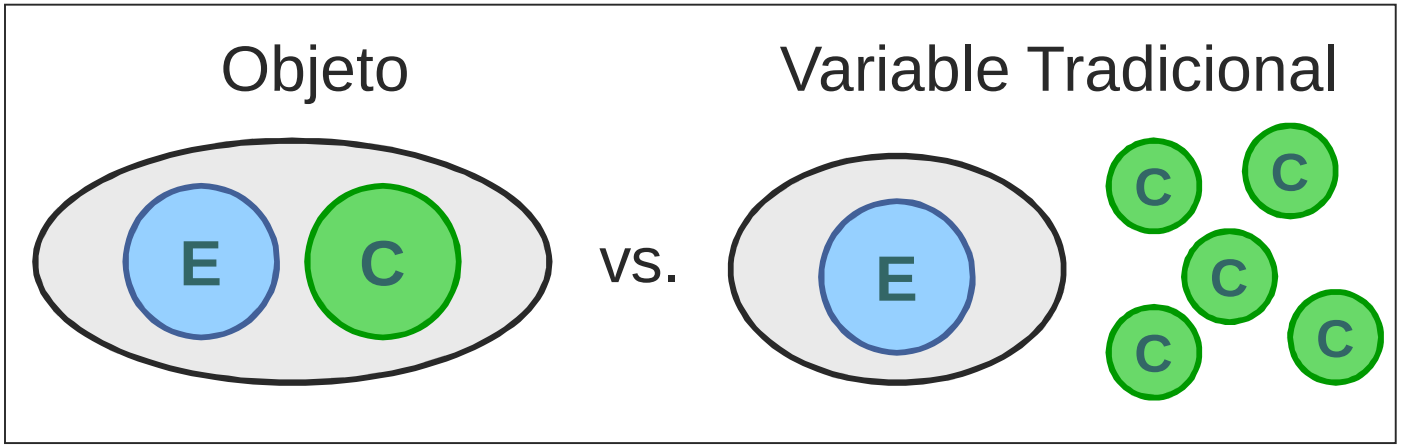
Contenido

- Construcciones Básicas
- Relaciones

Construcciones Básicas

Objeto

- Un objeto es una entidad discreta con límites e identidad bien definidos
- Encapsula estado y comportamiento:



- Es una instancia de una 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)

Clase (2)

Definición de una clase

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


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

Instancia de una clase (i.e. un objeto)

Clase (3)

- 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

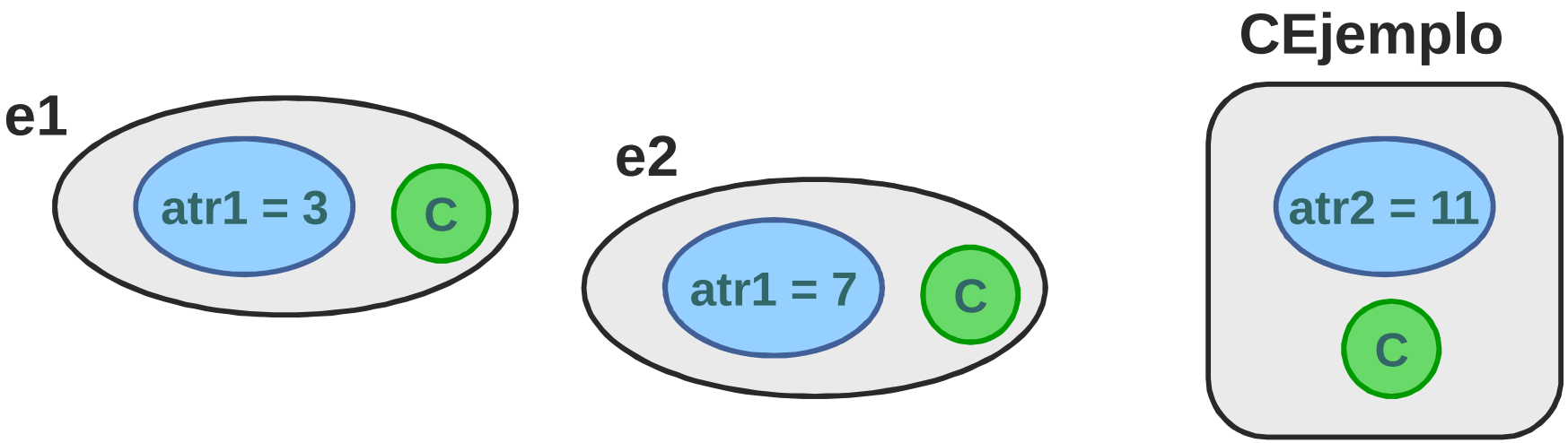
```
~CEjemplo();
```

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

Atributo (2)

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



Operación

- Es una especificación de una transformación o consulta que un objeto puede ser llamado a ejecutar
- Tiene asociada un nombre, una lista de parámetros y un tipo de retorno

Método

- Es la implementación de una operación para una determinada clase
- Especifica el algoritmo o procedimiento que genera el resultado o efecto de la operación

Operación y Método

```
class CEjemplo {
      int atr1;
      static int atr2;

      void oper(char c)
      {
            ... // un cierto algoritmo
      }
}
```

Operación

Método para `oper()` 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 oper() {
            this.atr1 = 2;    // código valido
         }
}
```

```
e.atr1      // código no valido
e->oper()   // 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 oper() {
            // un método
      }
}
```

```
class B {
      void oper() {
            // 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
  - Fechas
- También tipos definidos por el usuario, como enumerados

Data Type (2)

- 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

Data Type (3)

```
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

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

Asociación (2)

- 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

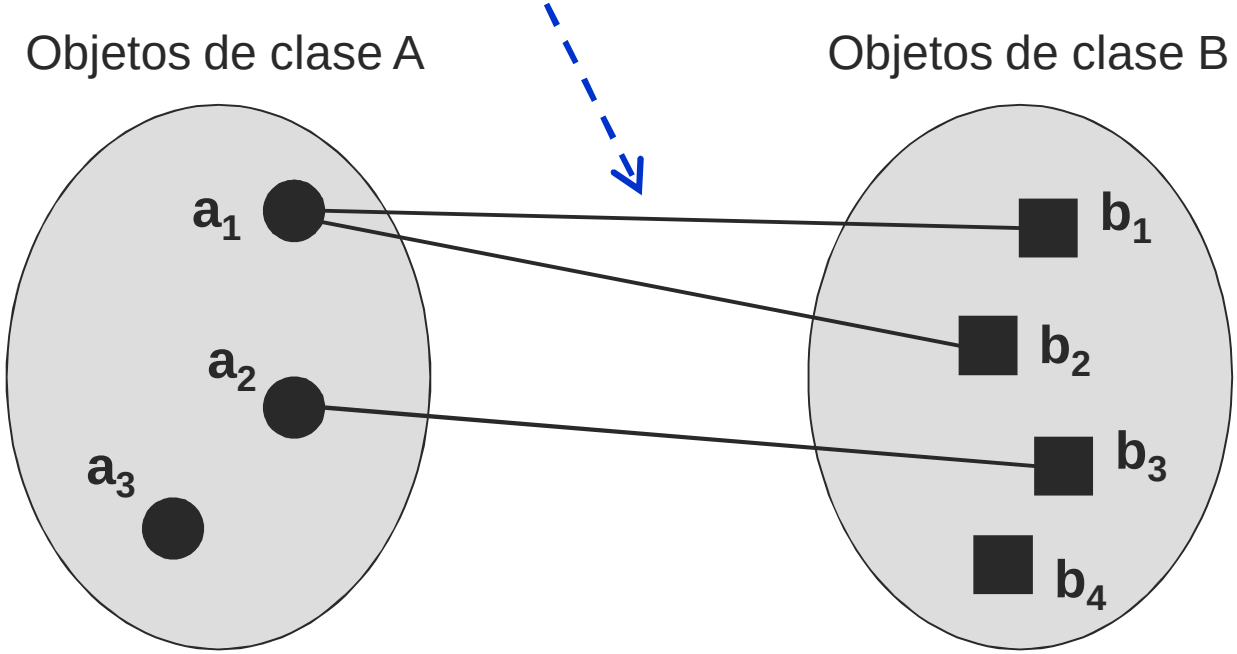
Link

- Es una tupla de referencias a instancias (objetos o data values)
- Es una instancia de una asociación
- Permite visibilidad entre todos las instancias participantes

Link (2)

- Ejemplo: asociación R entre clases A y B

$$R = \{<a_1,b_1>, <a_1,b_2>, <a_2,b_3>\}$$

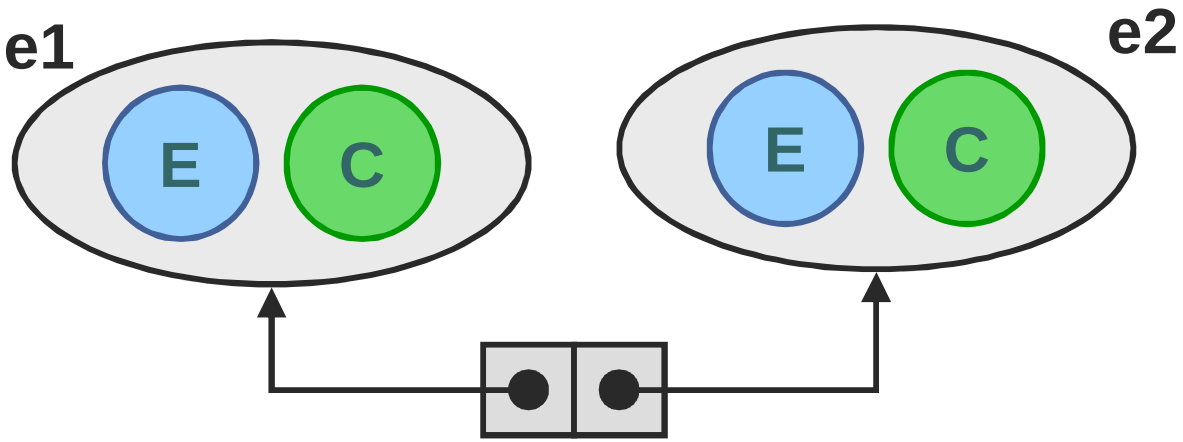


Representación de Asociaciones

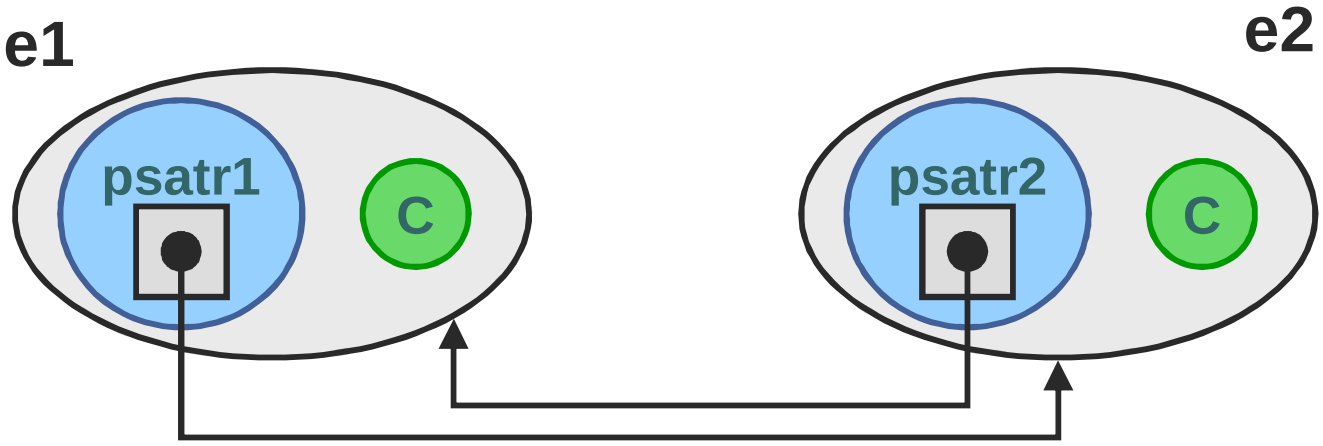
- Casi ningún lenguaje provee construcciones específicas para implementar asociaciones
- Para ello se suelen introducir “pseudoatributos” en las clases involucradas
- De esta manera un link no resulta implementado exactamente igual a su representación conceptual
- Una tupla es dividida y un componente es ubicado en el objeto referenciado por el otro componente de la tupla

Representación de Asocs. (2)

Representación conceptual



Implementación usual



Representación de Asocs. (3)

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

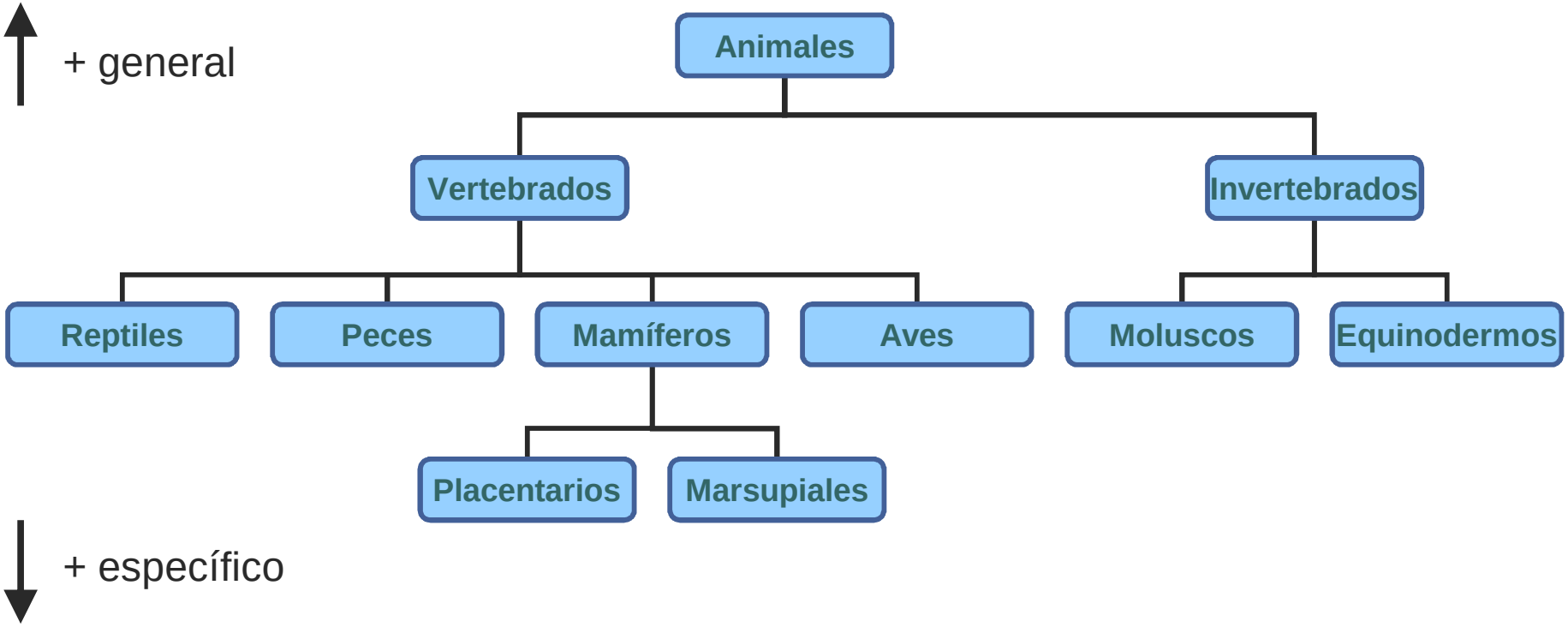
```
class Persona {
      private:
            String nombre;    // atributo
            Empresa *miEmp;   // pseudoatributo
      ...
}
```

- 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

Taxonomía


+ general
Animales
Vertebrados
Invertebrados
Reptiles
Peces
Mamíferos
Aves
Moluscos
Equinodermos
Placentarios
Marsupiales
+ específico

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*

Clase Base y Clase Derivada (2)

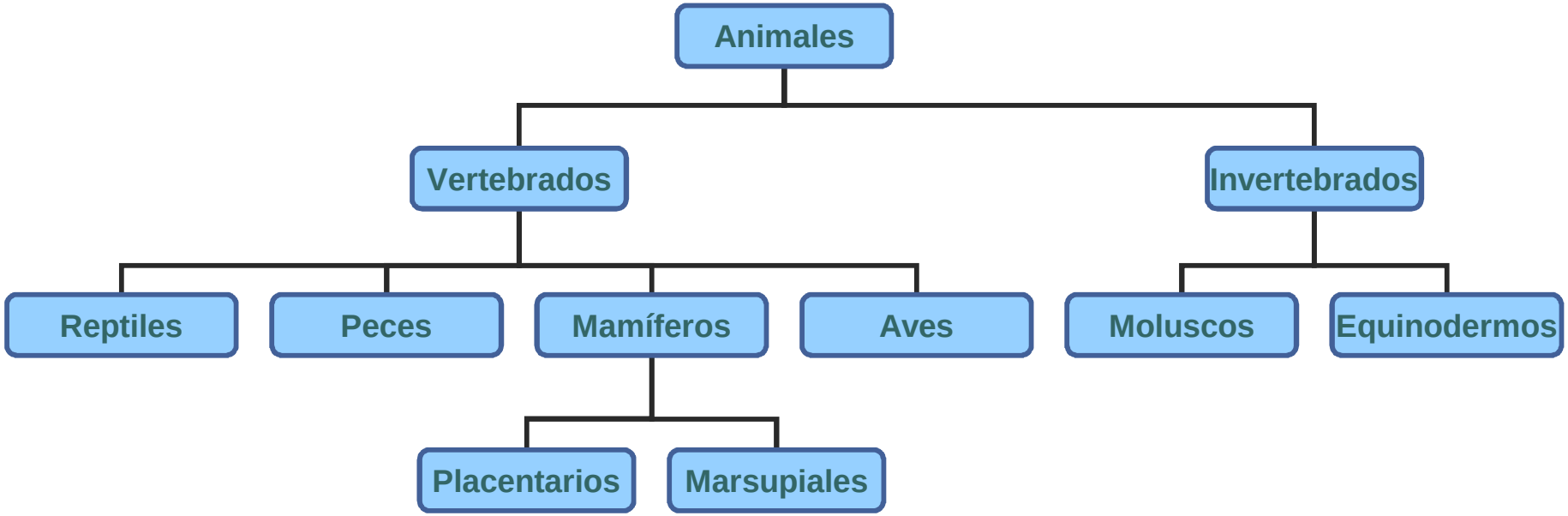
- Una clase puede tener cualquier cantidad de clases base, y también cualquier cantidad de clases derivadas.

```
class Vehiculo {
      ... // propiedades de vehiculo
}

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

class Moto : public Vehiculo {
      ... // props especificas de moto
}
```

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 y Descendientes (2)



- 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:

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

  donde,

  $(B,A) \in (<:) \Leftrightarrow B$ es clase derivada de A

- **Observación:** La relación <: define un orden parcial entre clases y es idéntica, transitiva y antisimétrica.

Subsumption

- 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: $b : B \wedge B <: A \Rightarrow b : A$
- 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”

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)

Descriptors (2)

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


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

$SD_{Empleado}$

ATT_{nombre}
PSA_{miEmp}
$OP_{getNombre}$
$MET_{Empleado::getNombre}$

SD_{Fjio}

ATT_{sueldo}
$OP_{getSueldo}$
$MET_{Fijo::getSueldo}$

Descriptors (3)

- 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 padre 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

Herencia (2)

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

Herencia (3)

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

$FD_{Empleado}$

ATT_{nombre}

PSA_{miEmp}

$OP_{getNombre}$

$MET_{Empleado::getNombre}$

FD_{Fijo}

ATT_{nombre} , ATT_{sueldo}

PSA_{miEmp}

$OP_{getNombre}$, $OP_{getSueldo}$

$MET_{Empleado::getNombre}$, $MET_{Fijo::getSueldo}$

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

Redefinición de Operaciones (2)

```
class A {
      private: T atr1;
      public: virtual void oper() {
                  ...
              }
}

class B : public A {
      private: T' atr2;
      public: void oper() {
                 super.oper();
                 ...
              }
}
```

El polimorfismo no es por defecto, se utiliza la palabra **virtual**

SD_A

ATT_{atr1}

OP_{oper}

$MET_{A::oper}$

SD_B

ATT_{atr2}

$MET_{B::oper}$

Redefinición de Operaciones (3)

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

FD_B

ATT_{atr1} , ATT_{atr2}
OP_{oper}
$MET_{B::oper}$, $MET_{A::oper}$

- 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

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

Operación Abstracta (2)

```
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

Operación Abstracta (3)

$FD_{Empleado}$

ATT_{nombre}

$OP_{getLiquidacion}$

FD_{Fijo}

ATT_{nombre} , ATT_{sueldo}

$OP_{getLiquidacion}$

$MET_{Fijo::getLiquidacion}$

$FD_{Jornalero}$

ATT_{nombre} , $ATT_{valorHora}$, $ATT_{cantHoras}$

$OP_{getLiquidacion}$

$MET_{Jornalero::getLiquidacion}$

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

Clase Abstracta (2)

```
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

Interfaz (2)

- 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)

Interfaz (3)

```
class IComando {
      virtual void play() = 0;
      virtual void stop() = 0;
      virtual void pause() = 0;
      ... ...
}
```

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”

Panel Frontal

Control Remoto

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 Jornalero();`
  - `j` es instancia directa de Jornalero
  - `j` es instancia indirecta de Empleado

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)

Despacho

- Con la introducción de subsumption es necesario reexaminar el significado de la invocación de operaciones
- Suponiendo *b* : *B* y *B* <: *A* es necesario determinar el significado de *b*.*f*() cuando *B* y *A* asocian métodos distintos a la operación *f*()
- Por tratarse de *b* : *B* resultaría natural que el método despachado por la invocación sea el de la clase *B*
- Sin embargo por subsumption también *b* : *A*, por lo que sería posible que el método a despachar sea el de la clase *A*

Despacho (2)

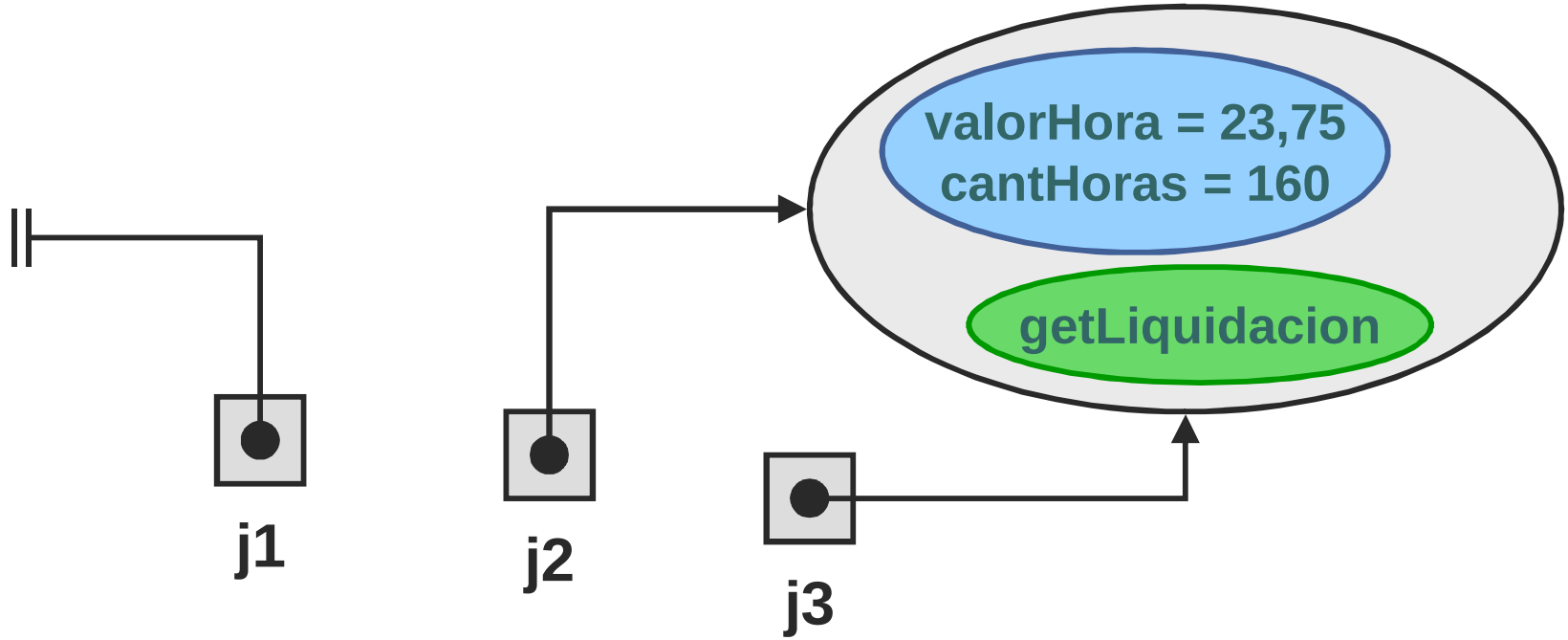
- En este tipo de casos lo deseable es que el método a despachar sea el asociado a la clase de la cual el objeto al que le es aplicada la operación es instancia directa
  - En el ejemplo anterior, *B*
- Siempre que es posible el despacho se realiza en tiempo de compilación denominándose despacho estático

Referencia

- Una referencia es un valor en tiempo de ejecución que es: void ó attached
- Si es attached la referencia identifica a un único objeto (se dice que la referencia está adjunta a ese objeto particular)
- Si es void la referencia no identifica a ningún objeto

Referencia (2)

```
Jornalero *j1 = null;                // void
Jornalero *j2 = new Jornalero(); // attached
Jornalero *j3 = *j2;                 // attached
```



Tipo Estático y Dinámico

- El *tipo estático* de un objeto es el tipo del cual fue declarada la referencia adjunta a él:
  - Se conoce en tiempo de compilación
- El *tipo dinámico* de un objeto es el tipo del cual es instancia directa
- En ciertas situaciones ambos tipos coinciden por lo que pierde el sentido realizar tal distinción

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 ancestra del tipo del objeto:
  - Lo cual es permitido por subsumption
- Se cumple la siguiente relación entre los tipos de *obj* :
  - *TipoDinamico*(*obj*) <: *TipoEstatico*(*obj*)

Tipo Estático y Dinámico (3)

```
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*?

Despacho Dinámico

- Los lenguajes de programación orientados a objetos permiten que el tipo dinámico de un objeto difiera del tipo estático
- Cuando se realiza una invocación a una operación polimórfica (que está redefinida) sobre un objeto utilizando una referencia a él declarada como de una de sus clases ancestras puede no ser correcto realizar el despacho en tiempo de compilación

Despacho Dinámico (2)

- De realizarse el despacho en forma estática se utilizaría para ello la única información disponible de él en ese momento:
  - La basada en el tipo estático
- Por lo que se despacharía (eventualmente) el método equivocado:
  - En particular, cuando la operación invocada es abstracta en la clase del tipo estático no hay método que despachar

Despacho Dinámico (3)

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

e->getLiquidación();
```

- La operación `getLiquidacion()` declarada en `Empleado` es polimórfica porque es redefinida en `Fijo` y en `Jornalero`
- 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 ancestra de las anteriores)
- En esta invocación debería despacharse $\text{MET}_{\textit{Fijo::getLiquidacion}}$ ó $\text{MET}_{\textit{Jornalero::getLiquidacion}}$
- Utilizando la información estática se intentaría despachar $\text{MET}_{\textit{Empleado::getLiquidacion}}$ (que en este ejemplo no existe!)

Despacho Dinámico (4)

- 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

Despacho Dinámico (5)

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

e->getLiquidación();
```

- En tiempo de compilación: al pasar por la invocación el compilador NO despacha método alguno
- En tiempo de ejecución: al pasar por la invocación el ambiente de ejecución del lenguaje se ocupa de averiguar el tipo dinámico de *e* y despachar al método correcto, es decir a: $\text{MET}_{Fijo::getLiquidacion}$ ó $\text{MET}_{Jornalero::getLiquidacion}$

Despacho Dinámico (6)

- 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

Ejemplo

- 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

Ejemplo (2)

```
class Empresa {
      private: String ruc;
               Set(Empleado) misEmps; //pseudoatributo

      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

Realización

- 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

Realización (2)

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

        public:

        void play() {
                ...
        }

        void stop() {
                ...
        }
        ... // implementacion del resto
            // de las operaciones
}
```

Realización (3)

- 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

Realización (4)

- 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*

Realización (5)

```
class ControladorAudio {
      ...

      public: void controlarAudio(IComando *c) {
            c->play();
            ...
      }
}
```

```
IComando *c1 = new ControlRemoto();
IComando *c2 = new PanelFrontal();

ca->controlarAudio(c1); // invocación valida
ca->controlarAudio(c2); // tambien valida
```

Realización (6)

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

Realización (7)

- 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 esta obligada a implementarlas a todas
- Si *C* es abstracta y realiza la interfaz *I*, entonces un objeto declarado como de tipo *I* debe ser instancia de alguna subclase concreta de *C* (o de otra clase que realice la interfaz *I*)

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