Programación Orientada a Objetos

Unidad 3: Relaciones entre clases

Teoría 04 - 08/09/2011 - Pablo Novara

Relaciones entre Clases

- **Agregación/Composición:**
  - una clase contiene como atributo una o más instancias de otra.
  - se dice que una clase está compuesta por otras
  - se identifica cuando una clase **es parte de** otra

Composición en C++

```
class A {
   private:  int x;
   public:   void CargarDatos(int _x);
};

class B {
    A a1, a2;
public:
    B(int x1, int x2) {
        a1.x=x1; a2.x=x2;
        a1.CargarDatos(x1);
        a2.CargarDatos(x2);
    }
    ...
```

Una o más instancias de una clase (A) son atributos de otra (B)

La clase compuesta debe utilizar la interfaz pública de las clases que la componen.

Composición en C++

```
class A {
   private:  int x;
   public:  A(int _x) { x=_x; }
};
```

Si la clase componente no tiene constructor por defecto obliga a la clase compuesta a inicializar las instancias

```
class B {
     A a1, a2;
public:
     B(int x1, int x2) { a1.x=x1; … }
     B(int x1, int x2) : a1(x1),a2(x2) {
          …
     }
     …
};
```

La invocación de los constructores se realiza antes de la llave que da inicio al método, precedida por dos puntos

Ejemplos Composición

– La clase Auto contiene cuatro instancias de la clase Rueda, y una de la clase Motor.

```
class Motor { … };

class Rueda { … };

class Auto {
    Rueda r[4];
    Motor m;
    float velocidad, acelaracion;
    char color[10];
    …
};
```

Ejemplos Composición

– La clase Curso contiene N instancias de la clase Alumno.

```
class Alumno {
    …
};
class Curso {
    char materia[50];
    Alumno alumnos[100];
    int carga_horaria;
    …
};
```

Ejemplos Composición

– Un triangulo se construye con 3 puntos

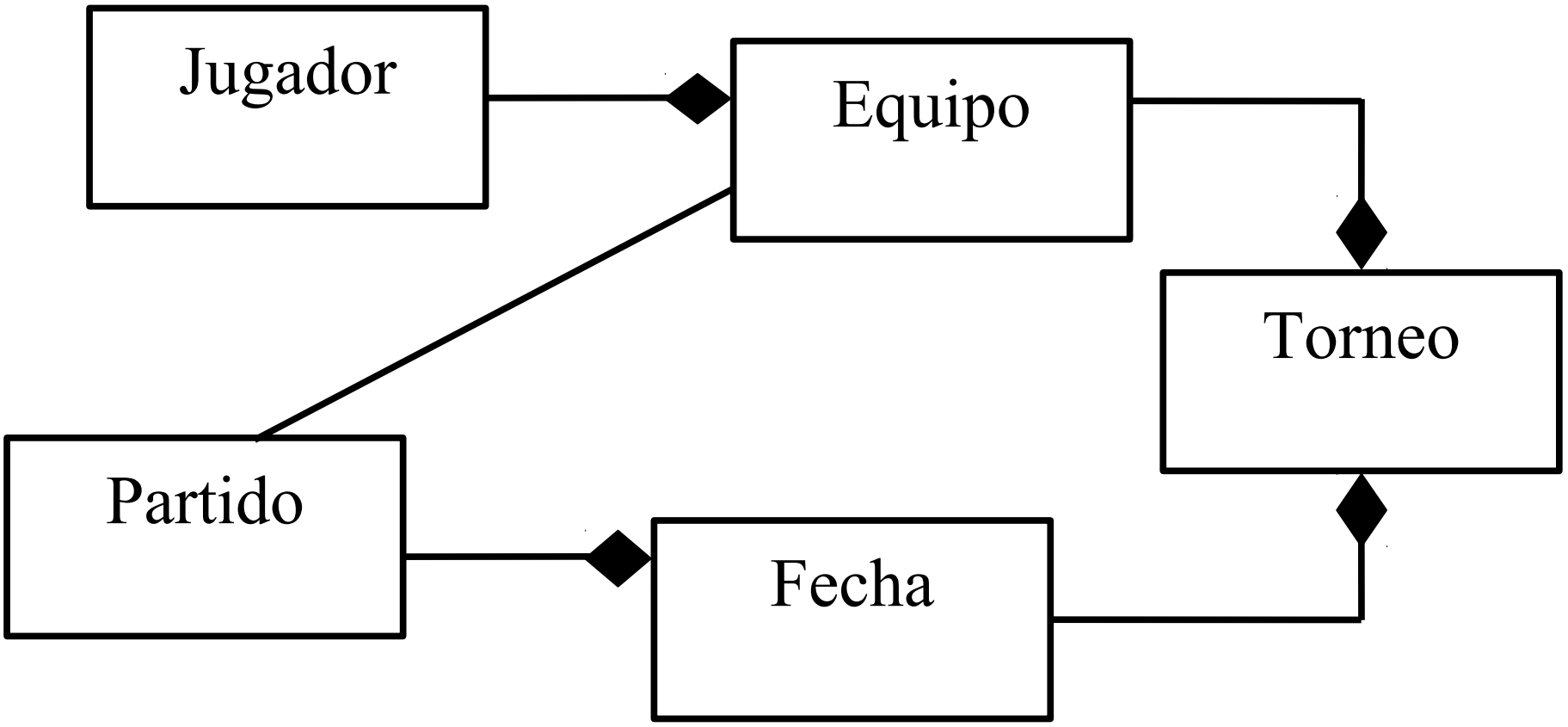
```
class Punto { float x,y; public: float VerX(); float VerY(); … };

class Triangulo {
     Punto p1,p2,p3;
     float area;
public:
   Triangulo(Punto a1, Punto a2, Punto a3) {p1=a1; p2=a2; p3=a3;}
   void CalcularArea() {
     float dx1=p2.VerX()-p1.VerX(), dy1=p2.VerY()-p1.VerY();
     float dx2=p3.VerX()-p1.VerX(), dy2=p3.VerY()-p1.VerY();
     area = abs(dx1*dy1-dx2*dy2)/2;
   }
   float VerArea() { return area; } …
};
```

Ejemplos Composición

– Modelar un torneo de fútbol, considerando jugadores, equipos, partidos, fechas, etc.

```
class Jugador { … };

class Equipo { int id; Jugador *j;  … };

class Partido { int id_e1, id_e2; … };

class Fecha { Partido p[19]; … };

class Torneo { Equipo e[20]; Fecha[19]; …  }
```

Notación UML (Composición)



Relaciones entre Clases

- **Amistad**:
  - una clase/función **permite a otras clases/funciones acceder a sus atributos/miembros privados**
  - atenta contra el principio de ocultación

Amistad en C++

```
class A { int x; };

class B {
  A a1;
  public:
    B(int _x) {
        a1.x= _x;
    }
};

void func(A a1) {
    int d=2*a1.x;
    ...
}
```

x es un atributo privado

otras clases o funciones no pueden acceder a los atributos privados

Amistad en C++

```
class A {
    int x;
    friend class B;
    friend void func(A a1);
};

class B {
 public: B(int _x) {
        a1.x=_x;
    }
};

void func(A a1) {
    int d=2*a1.x;
    ...
```

para declarar amistad
se utiliza la palabra clave
friend seguida del prototipo
de la función o clase

las cases y funciones
amigas tienen acceso
a miembros y atributos
privados

Relaciones entre Clases

- **Herencia**:
  - una clase (sub-clase o clase derivada) hereda todos los atributos y métodos de otra (super-clase o clase base)
  - se identifica cuando **una clase es un caso particular de otra**
  - permite la creación de clasificaciones jerárquicas, dividiendo el problema en diferentes niveles de detalle
  - evita repetir código común a varias clases

Herencia en C++

```
class A {
   public:   void MetodoA() { ... }
};

class B : public A {
  public:
     void MetodoB() { ... }
};

int main() {
   SubClase s;
   s.MetodoA();
   s.MetodoB();
   ...
```

Si B hereda de A,
A es la *super-clase*
y B la *sub-clase*

Una instancia de
la sub-clase "hereda"
métodos y atributos
de la super-clase

Herencia y Visibilidad

```
class SuperClase {
  private:  void SuperMetodoPrivado() { ... }
  public:   void SuperMetodoPublico() { ... }
};

class SubClase : public SuperClase {
  public:
     void Prueba() {
          SuperMetodoPrivado();
          SuperMetodoPublico();
     }
};
```

Desde la subclase no se puede acceder a métodos privados de la super-clase

Herencia y Visibilidad

Nueva etiqueta:
protected (protegido)

```
class Clase {
  private:  void MetodoPrivado() { ... }
  protected: void MetodoProtegido() { … }
  public:   void MetodoPublico() { ... }
};

int main () {
    Clase c1;
    c1.MetodoPrivado();
    c1.MetodoProtegido();
    c1.MetodoPublico();
}
```

Desde el programa cliente no se puede acceder a métodos o atributos privado ni protegidos, solo públicos

Herencia y Visibilidad

```
class SuperClase {
  private:  void SuperMetodoPrivado() { ... }
  protected:  void SuperMetodoProtegido() { ... }
  public:  void SuperMetodoPublico() { ... }
};

class SubClase : public SuperClase {
  public:
     void Prueba() {
          SuperMetodoPrivado();
          SuperMetodoProtegido();
          SuperMetodoPublico();
     }
};
```

Desde la subclase no se puede acceder a métodos privados, pero sí a métodos públicos y protegidos

Herencia y Visibilidad

```
class A {
    … public: MetodoA(); …
};

class B : public A { … } ;

class C : private A { … } ;

int main() {
    B b1;
    C c1;
    b1.MetodoA();
    c1.MetodoA();
```

La etiqueta que precede a la super-clase limita la visibilidad de los métodos y atributos heredados

Herencia y Constructores

```
class A {
    int x;
    public: A(int _x) {x=_x;}
};

class B : public A {
        ...
    public:
        B() { ... }
        B(int x) : A(x) { … };
        ...
} ;
```

Si la super-clase no tiene constructor nulo, obliga a la subclase a invocar el constructor con la misma notación que en el caso de composición

Ejemplos Herencia

Clase base, la más general para cualquier deporte

```
class Equipo { … };
```

Clases hijas, casos particulares de equipo

```
class EquipoFutbol : public Equipo {…};
class EquipoBasquet : public Equipo {…};
class EquipoVoley: public Equipo {…};
```

Instancias de cada subclase, equipos específicos de cada deporte

```
EquipoFutbol Boca;
EquipoBasquet Atenas;
EquipoVoley Bolivar;
```

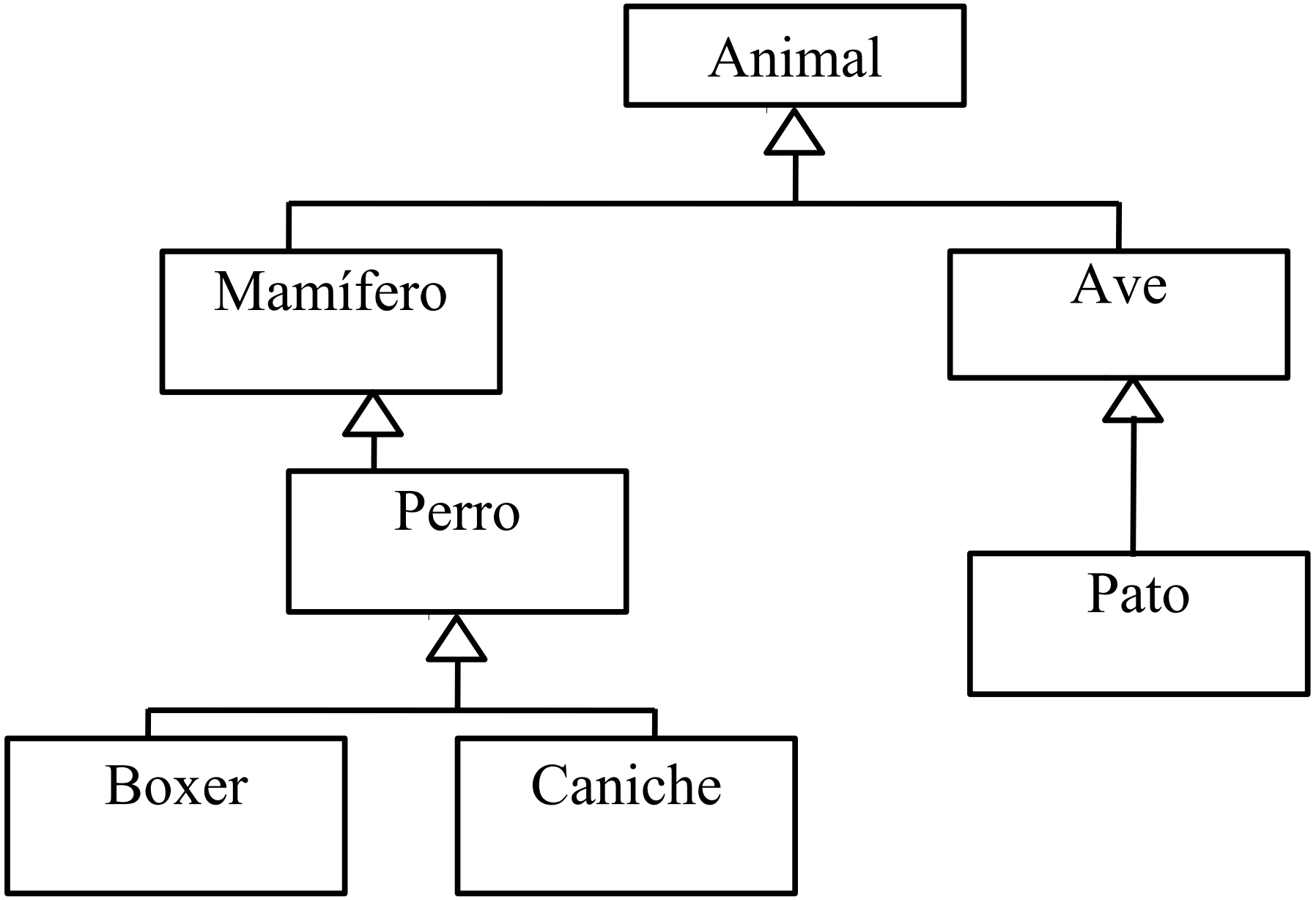
Ejemplos Herencia

```
class Animal {
    int patas;
    char especie[50];
    bool vuela;
    int edad;

  public:
    Animal(char esp[], int p, bool v) { edad=0; }
    const char *VerEspecie() { return especie; }
    void Envejecer() { edad++; }
    int VerEdad() { return edad; }
  };
```

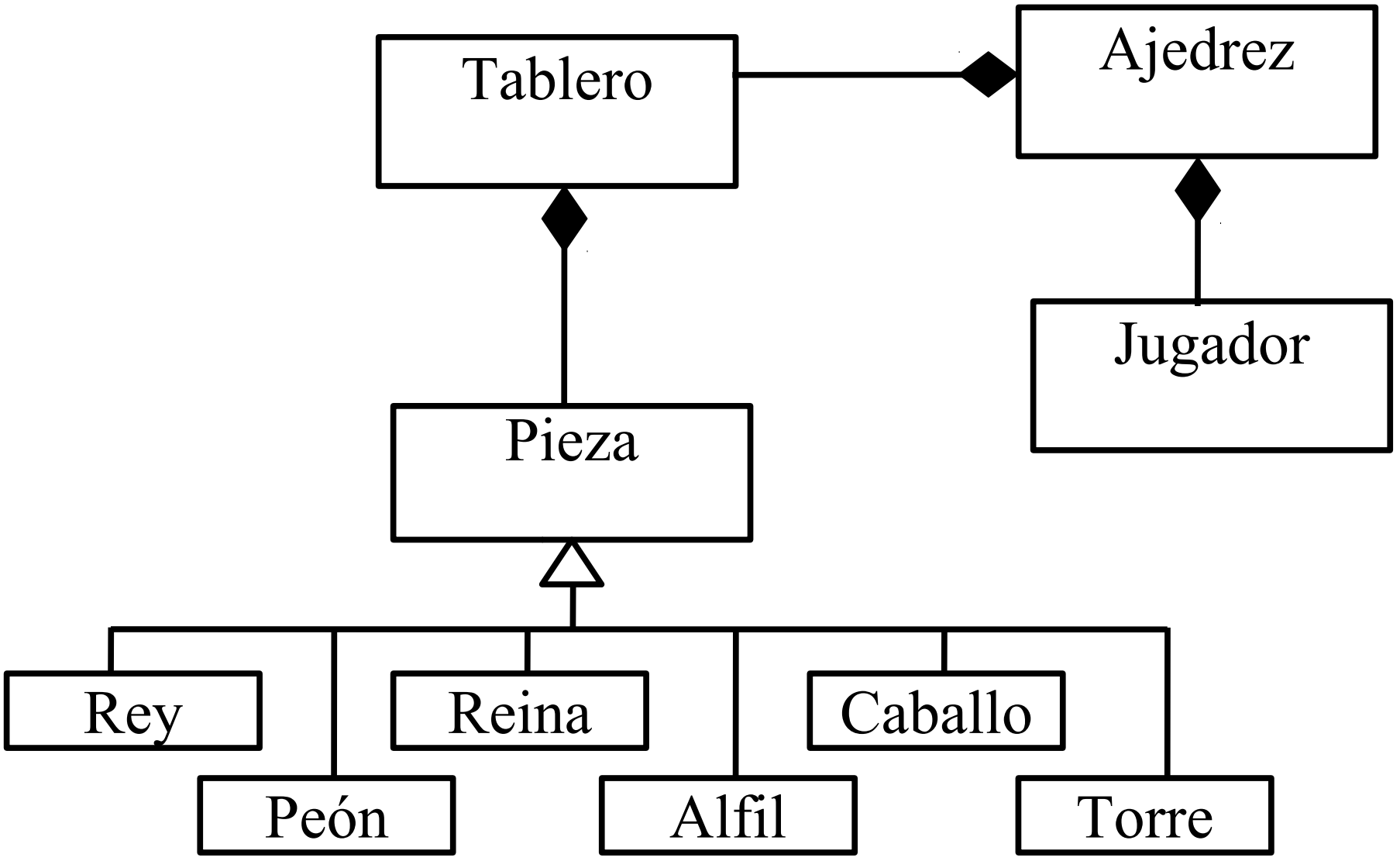
Ejemplos Herencia

```
class Perro : public Animal {
 public:
     Perro():Animal(“perro”,4,false) {}
     void Ladrar() { cout<<”Guau!”<<endl; }
};

class Pato : public Animal {
 public:
     Pato():Animal(“pato”,2,true) {};
     void Graznar() { cout<<”Cuack!”<<endl; }
};
```

Notación UML (Herencia)



Notación UML



Herencia Múltiple

```
class Caballo {
    public: void Relinchar();
            void Trotar();
}

class Ave {
    public: void Volar();
};

class Pegaso : public Caballo, public Ave {

   …

}
```

Relaciones entre Clases

- **Polimorfismo**:
  - es una extensión de la relación de herencia donde **las sub-clases alteran el comportamiento de los métodos de la super-clase**
  - se utiliza mediante punteros del tipo de la clase base que en realidad apuntan a instancias de las clases derivadas

Polimorfismo

```
class A {
   public: void MetodoA() { ... }
};

class B : public A {
  public: void MetodoB() { ... }
};

int main() {
   A *x = new B;

   x->MetodoA();
   x->MetodoB();
```

Se puede utilizar un puntero a la super-clase para apuntar a la sub-clase

Si el puntero es de tipo A* solo se pueden invocar métodos de A sin importar a qué apunta

Polimorfismo

```
class A {
  public:
    void Metodo() { ... }
};

class B : public A {
  public:
    void Metodo() { ... }
};

int main() {
   A *a=new A; B *b=new B;
   a->Metodo();  // llama al método de A
   b->Metodo();  // llama al método de B
```

Ambas clases tienen métodos con el mismo nombre

Se determina a cual llamar según el tipo de puntero

Polimorfismo

```
class A {
  public:
  virtual void Metodo() { ... }
};

class B : public A {
  public:
    void Metodo() { ... }
};

int main() {
  A *s = new B;
  s.Metodo();
  ...
```

Un *método virtual* se declara anteponiendo la palabra virtual al prototipo

Para aprovechar el polimorfismo hay que utilizar punteros del tipo de la super-clase que apunten a la sub-clase

Se determina a cual llamar según el tipo del objeto apuntado, no del puntero

Polimorfismo

```
class A {
  public:
  virtual void Metodo() = 0;
};

class B : public A {
  public:
    void Metodo() { ... }
};

int main() {
  A a1;
  A*s = new B;
```

Un *método virtual puro* es aquel que no tiene implementación en la super-clase

Una clase con al menos un método virtual puro es una *clase abstracta*, y no puede ser instanciada