

Programación Orientada a Objetos

Unidad 3: Relaciones Entre Clases

© Pablo Novara

2024

RELACIONES ENTRE CLASES

- ▶ **Amistad:** otra clase/función puede acceder a los métodos y atributos privados. Atenta contra el principio de ocultación.
- ▶ **Composición:** una clase contiene como atributo una o más instancias de otra.
- ▶ **Herencia:** una clase (hija) hereda todos los atributos y métodos de otra. Se identifica cuando una clase es un caso particular (o extensión) de otra.
 - ▶ **Polimorfismo:** un uso avanzado de herencia para separar el ¿qué hace una clase? del ¿cómo lo hace?

AMISTAD

Una clase **permite** a otras clases y/o funciones acceder a sus miembros privados.



AMISTAD

```
class A {  
    int x; // x es privado  
  
};  
  
void foo(A a1) {  
    int d = 2*a1.x; // ERROR  
    ...  
}  
  
class B {  
public:  
    void bar(A &a1) {  
        a1.x = 42; // ERROR  
        ...  
    }  
};
```

AMISTAD

```
class A {  
    int x; // x es privado  
    friend void foo(A a1);  
  
};
```

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

```
class B {  
public:  
    void bar(A &a1) {  
        a1.x = 42; // ERROR  
        ...  
    }  
};
```

AMISTAD

```
class A {  
    int x; // x es privado  
    friend void foo(A a1);  
    friend class B;  
};  
  
void foo(A a1) {  
    int d = 2*a1.x; // OK  
    ...  
}  
  
class B {  
public:  
    void bar(A &a1) {  
        a1.x = 42; // OK  
        ...  
    }  
};
```

AMISTAD

Una clase **permite** a otras clases y/o funciones acceder a sus miembros privados.

Es una declaración *unidireccional*.

Es una forma de romper el principio de ocultación para casos *excepcionales*.

⚠ *Usar solo como último recurso*

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: EJEMPLO 1

```
class Rueda {  
    float m_tamano;  
public:  
    void CargarDatos(float tamano);  
    ...  
};  
  
class Bicicleta {  
    Rueda r_delantera, r_trasera;  
public:  
    Bicicleta(float rodado) { ... }  
    ...  
};  
  
int main() {  
    Bicicleta b(29);  
}
```

⚠ *Desde afuera de la clase compuesta,
la composición no se percibe*

COMPOSICIÓN: EJEMPLO 1

```
class Rueda {  
    float m_tamano;  
public:  
    void CargarDatos(float tamano);  
    ...  
};  
  
class Bicicleta {  
    Rueda r_delantera, r_trasera;  
public:  
    Bicicleta(float rodado) {  
        r_delantera.m_tamano = rodado;  
        r_trasera.m_tamano = rodado;  
    }  
    ...  
};
```

⊗ *m_tamano es privado*

COMPOSICIÓN: EJEMPLO 1

```
class Rueda {  
    float m_tamano;  
public:  
    void CargarDatos(float tamano);  
    ...  
};  
  
class Bicicleta {  
    Rueda r_delantera, r_trasera;  
public:  
    Bicicleta(float rodado) {  
        r_delantera.CargarDatos(rodado);  
        r_trasera.CargarDatos(rodado);  
    }  
    ...  
};
```

COMPOSICIÓN Y CONSTRUCTORES

```
class Rueda {  
    float m_tamano;  
public:  
    Rueda(float tamanio);  
    ...  
};
```

```
class Bicicleta {  
    Rueda r_delantera, r_trasera;  
public:  
    Bicicleta(float rodado) {  
        ...
```



ERROR: en este punto las ruedas ya deberían estar construidas

```
    }  
    ...  
};
```

COMPOSICIÓN Y CONSTRUCTORES

```
class Rueda {  
    float m_tamano;  
public:  
    Rueda(float tamanio);  
    ...  
};
```

```
class Bicicleta {  
    Rueda r_delantera, r_trasera;  
public:  
    Bicicleta(float rodado) : r_delantera(rodado), r_trasera(rodado) {  
        ...  
    }  
    ...  
};
```

👍 *invocar a los constructores de las partes, antes de abrir las llaves*

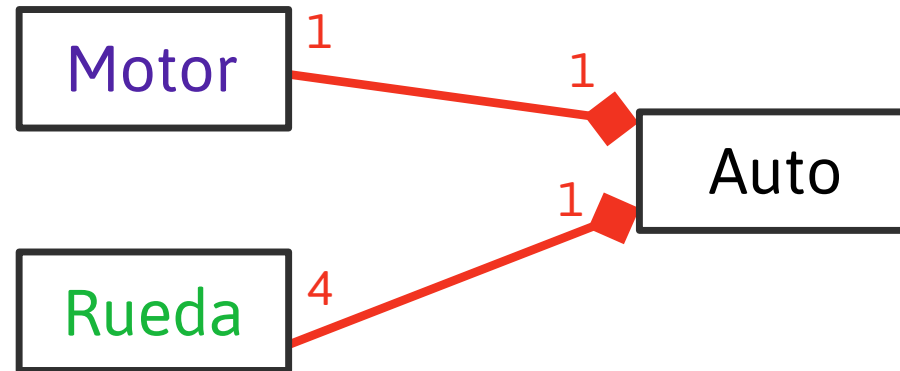
COMPOSICIÓN: EJEMPLO 2

```
class Motor { ... };
class Rueda { ... };

class Auto {
    Motor m_motor;
    Rueda m_ruedas[4];
    string m_color;
    float m_velocidad, m_aceleracion;
public:
    ...
    void EncenderMotor() {
        m_motor.Encender();
    }
    int VerRPMs() const {
        return m_motor.VerRPMs();
    }
    ...
};
```

COMPOSICIÓN EN UML: EJEMPLO 2

```
class Motor { ... };  
class Rueda { ... };  
  
class Auto {  
    Motor m_motor;  
    Rueda m_ruedas[4];  
    ...  
};
```

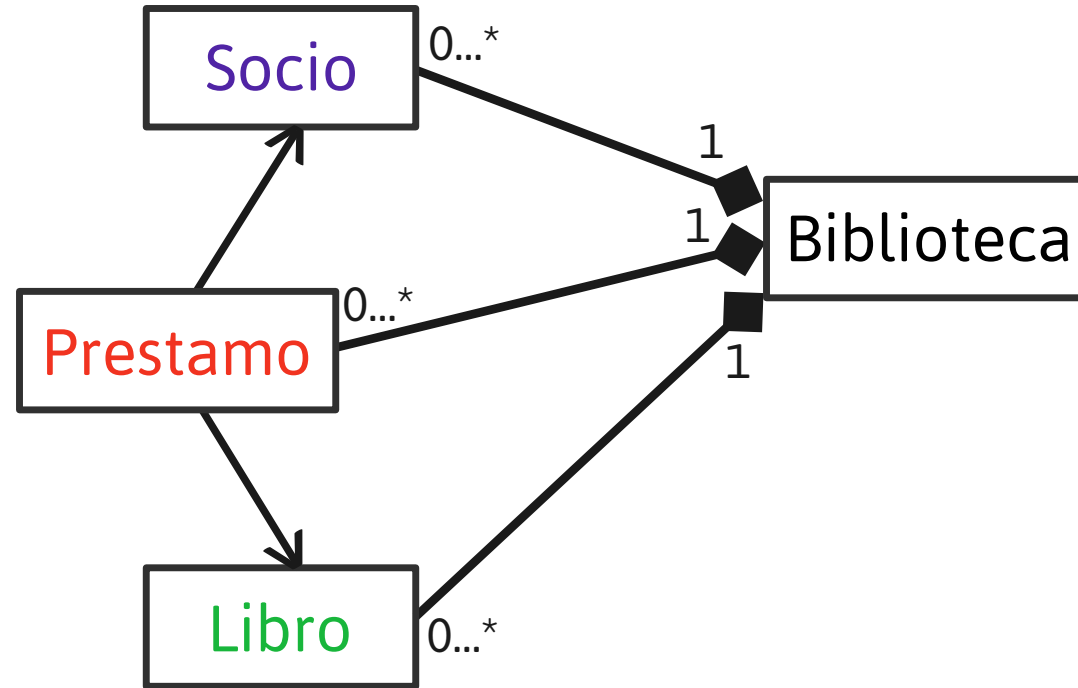


COMPOSICIÓN: EJEMPLO 3

```
class Socio { int m_dni; ... };
class Libro { int m_codigo; ... };
class Prestamo {
    int m_socio; // dni
    int m_libro; // codigo
    ...
public:
    ...
};
class Biblioteca {
    vector<Socio> m_socios;
    vector<Libro> m_libros;
    vector<Prestamo> m_prestamos;
    ...
public:
    ...
}
```

⚠ *No todo es composición*

COMPOSICIÓN EN UML: EJEMPLO 3



COMPOSICIÓN: EJEMPLO 4

```
class Jugador { ... };

class Juego {
    vector<Jugador> m_jugadores;
    ...
public:
    ...
    void AgregarJugador(const Jugador &un_jug) {
        m_jugadores.push_back(un_jug);
    }
    int CantidadJugadores() const {
        return m_jugadores.size();
    }
    Jugador VerJugador(int cual) const {
        return m_jugadores[cual];
    }
    ...
};
```

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

```
class Equipo {  
public:  
    string VerNombre() const { ... }  
    ...  
};  
  
class EquipoFutbol : public Equipo {  
public:  
    string VerArquero() const { ... }  
    ...  
};  
  
int main() {  
    EquipoFutbol obj;  
    cout << obj.VerNombre() << endl;  
    cout << obj.VerArquero() << endl;  
    ...  
}
```

HERENCIA: EJEMPLO 1

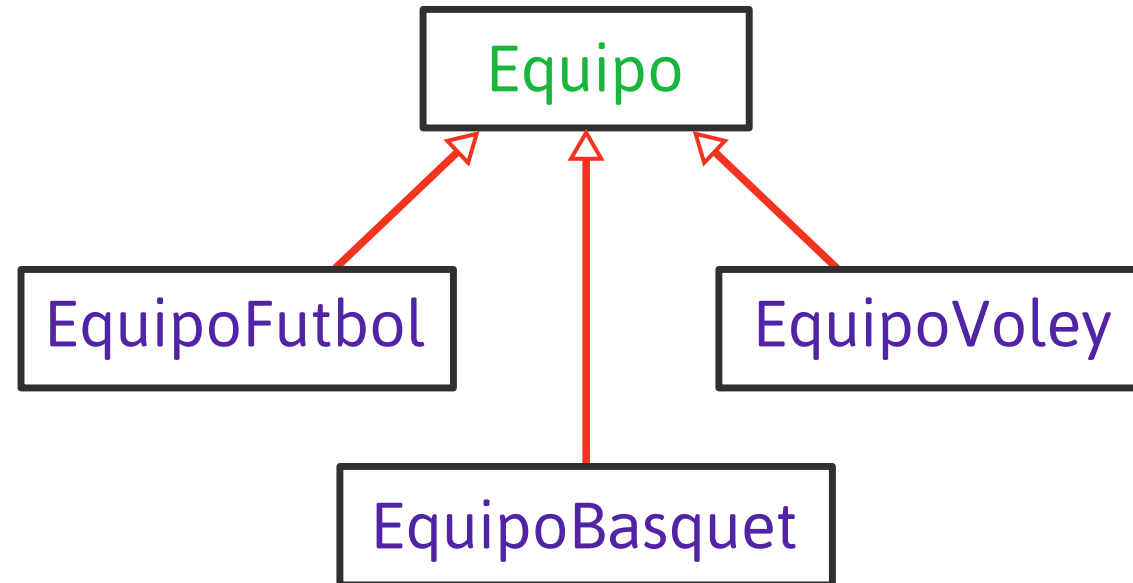
```
// clase Base  
// representa un equipo genérico, de cualquier deporte  
class Equipo { ... };
```

```
// clases Hijas  
// para representar tipos de equipos particulares  
class EquipoFutbol : public Equipo { ... };  
class EquipoBasquet : public Equipo { ... };  
class EquipoVoley : public Equipo { ... };
```

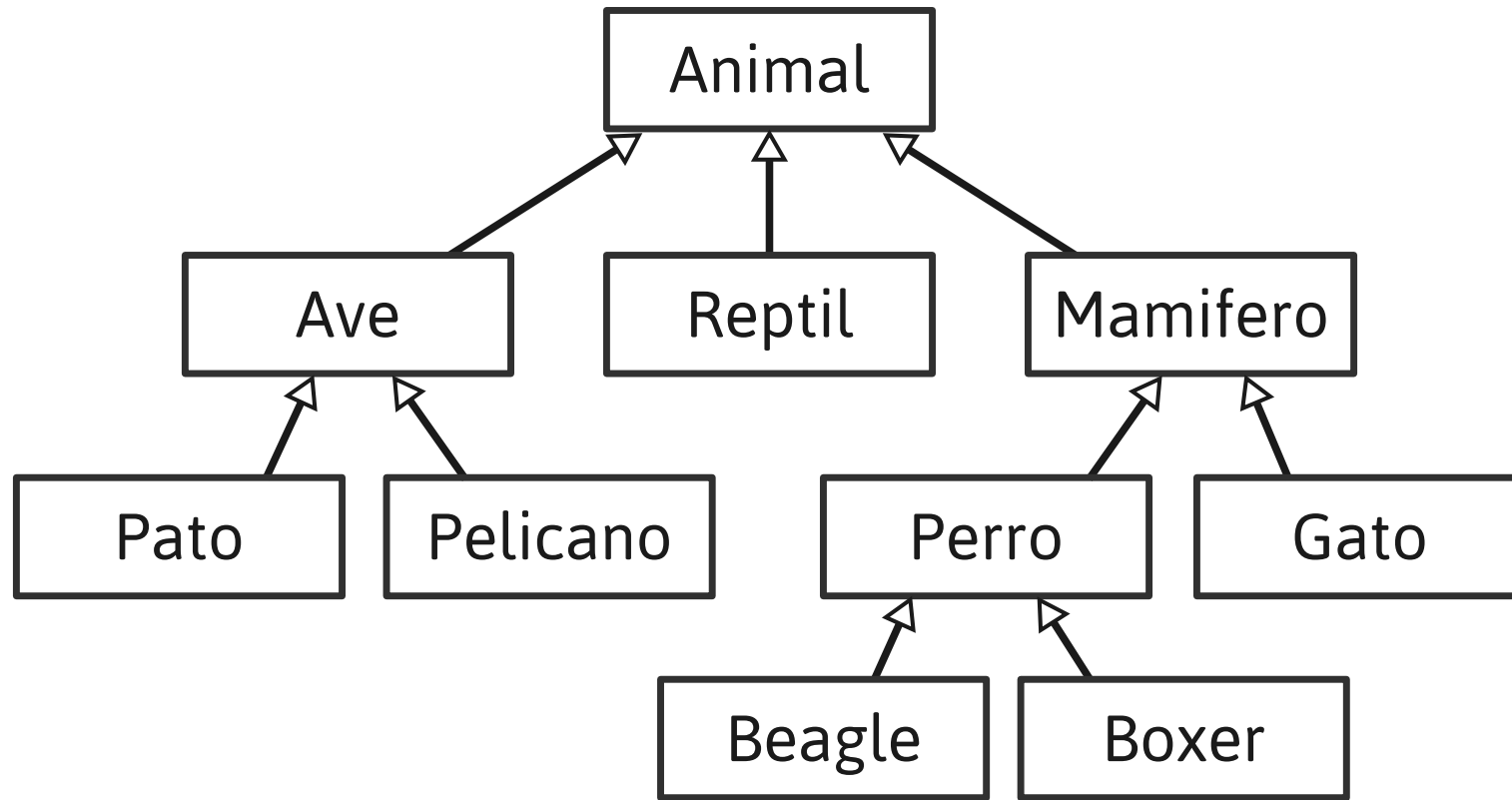
```
int main() {  
    // Instancias/Objetos: equipos específicos  
    EquipoFutbol boca("C. A. Boca Juniors");  
    EquipoBasquet spurs("San Antion Spurs");  
    EquipoVoley bolivar("Pers3onal Bolívar");  
    ...  
    Equipo sky(9, "ciclismo", "Team Sky");  
    ...  
}
```

HERENCIA EN UML: EJEMPLO 1

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



HERENCIA EN UML: EJEMPLO 2



```
class Animal { ... };  
class Mamifero : public Animal { ... };  
class Perro : public Mamifero { ... };  
class Boxer : public Perro { ... };  
Boxer mi_perro("Mordelón");
```

HERENCIA Y CONSTRUCTORES

```
class A {  
    ...  
public:  
    A(int x) { ... }  
    ...  
};  
  
class B : public A {  
public:  
    B() { ... } // ERROR  
    B() : A(0) { ... } // OK  
    B(int x) : A(x) { ... }; // OK  
    ...  
} ;  
  
int main() {  
    B b1, b2(10);  
    ...  
}
```


HERENCIA Y CONTROL DE ACCESO

```
class SuperClase {  
    private:    void SuperMetodoPrivado();  
    public:    void SuperMetodoPublico();  
};  
  
void foo(SubClase sub1) {  
    sub1.MetodoPrivado(); // ERROR  
    sub1.MetodoPublico(); // OK  
}  
  
class SubClase : public SuperClase {  
    ...  
    void Prueba() {  
        SuperMetodoPrivado(); // ERROR  
        SuperMetodoPublico(); // OK  
    }  
    ...  
};
```

HERENCIA Y CONTROL DE ACCESO

```
class SuperClase {  
    private:    void SuperMetodoPrivado();  
    protected: void SuperMetodoProtegido();  
    public:     void SuperMetodoPublico();  
};  
void foo(SubClase sub1) {  
    sub1.MetodoPrivado(); // ERROR  
    SuperMetodoProtedido(); // ERROR  
    sub1.MetodoPublico(); // OK  
}  
class SubClase : public SuperClase {  
    ...  
    void Prueba() {  
        SuperMetodoPrivado(); // ERROR  
        SuperMetodoProtedido(); // OK  
        SuperMetodoPublico(); // OK  
    }  
};
```

CONTROL DE ACCESO - RESUMEN

- ▶ **private:**

- ▶ función global: **NO**
- ▶ función miembro de cualquier otra clase: **NO**

- ▶ **protected:**

- ▶ función global: **NO**
- ▶ función miembro de clase *no relacionada*: **NO**
- ▶ función miembro de clase **hija**: **SI**

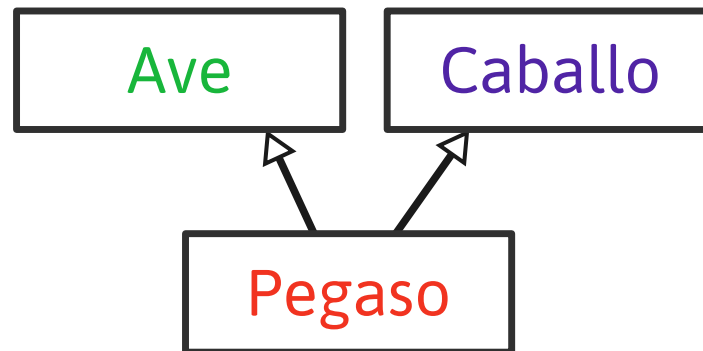
- ▶ **public:**

- ▶ función global o miembro de cualquier clase: **SI**

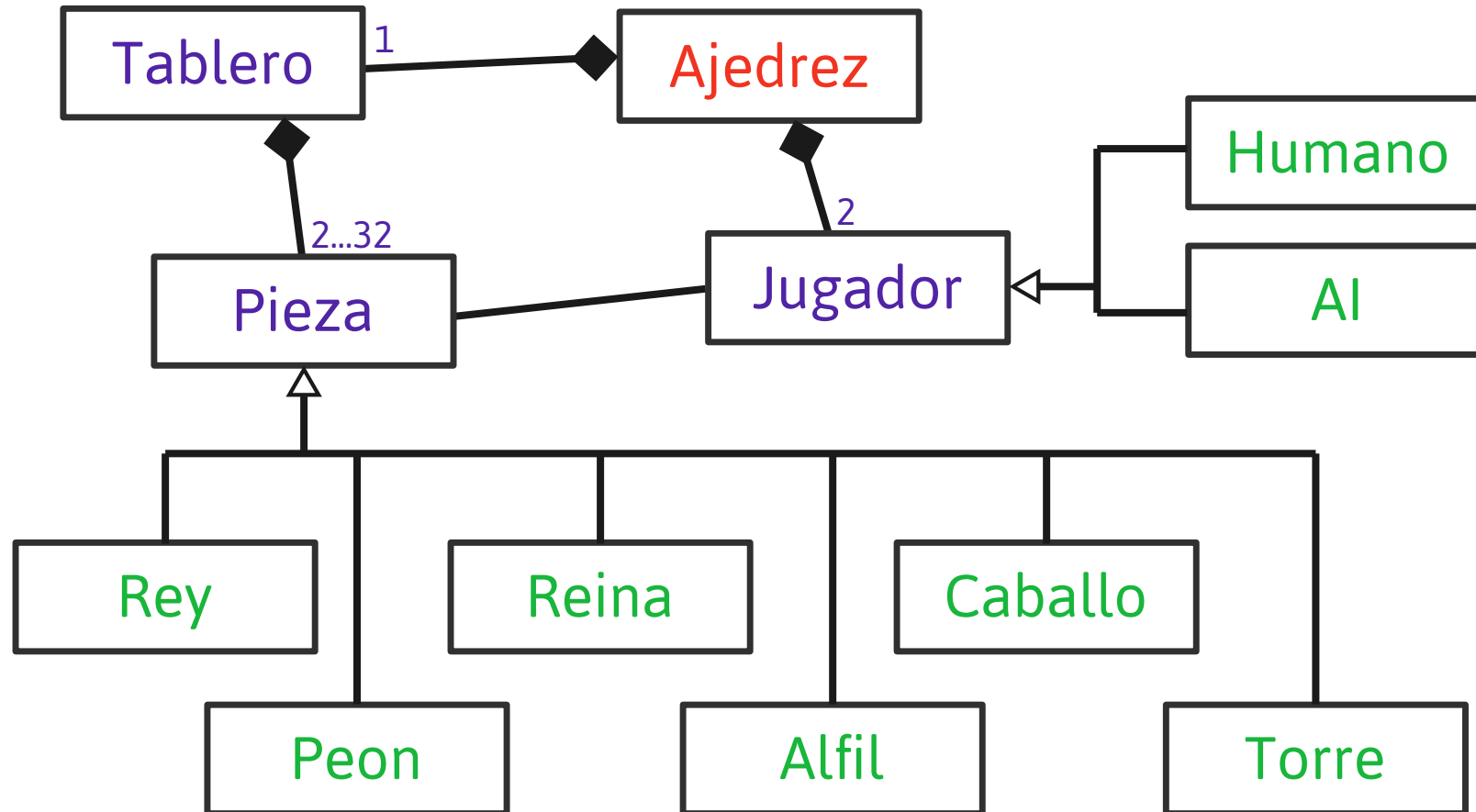
⚠ *pueden agregar excepciones mediante amistad*

HERENCIA MÚLTIPLE

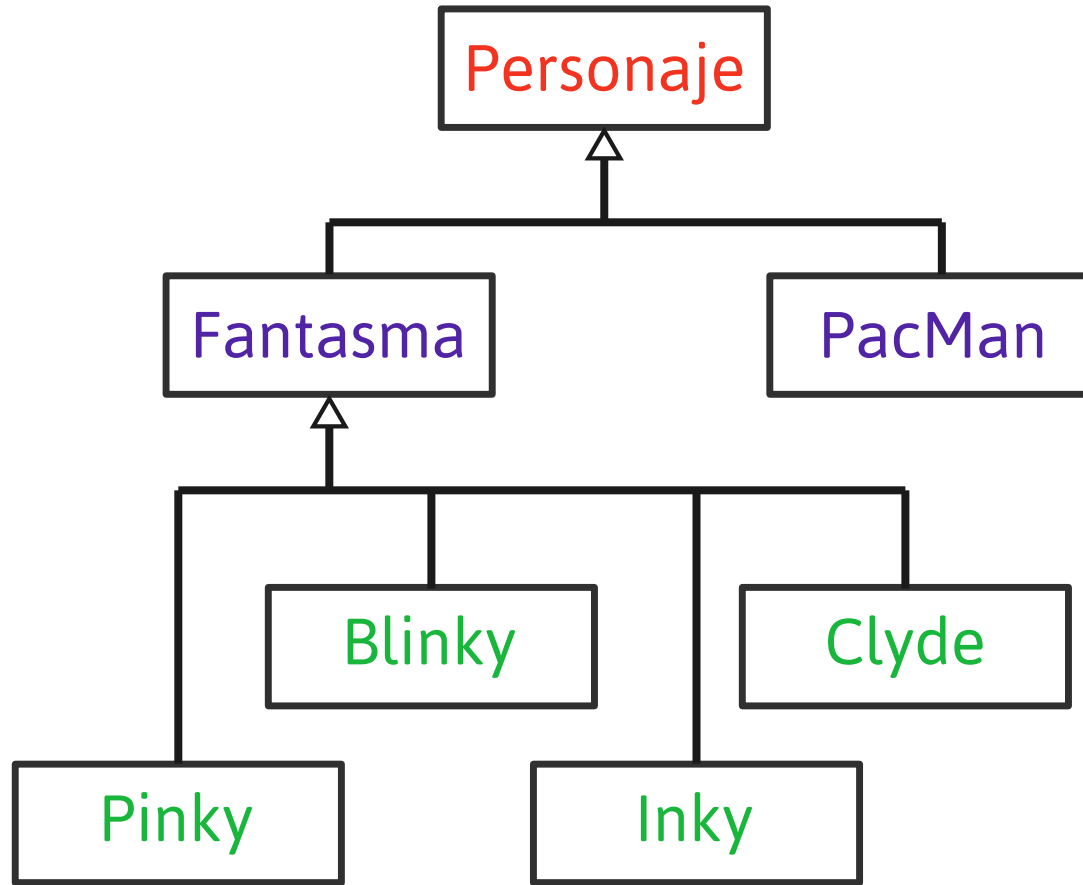
```
class Caballo { ... void Trotar(); ... };  
class Ave { ... void Volar(); ... };  
class Pegaso: public Caballo, public Ave {  
    ...  
};  
int main() {  
    Pegaso p1;  
    p1.Trotar();  
    p1.Volar();  
}
```



EJEMPLO COMPLETO



POLIMORFISMO (DINÁMICO)



POLIMORFISMO (DINÁMICO)

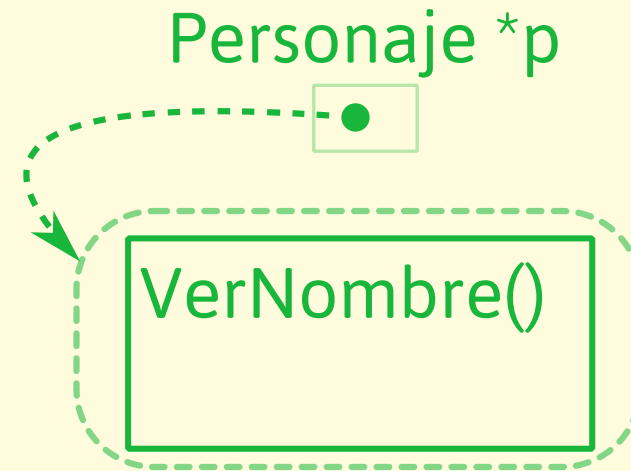
Mecanismo por el cual **sub-clases** [re]define el comportamiento de métodos de una super-clase.

Permite separar el **qué** del **cómo**,
y plantear clases y funciones **extensibles**.



POLIMORFISMO EN C++

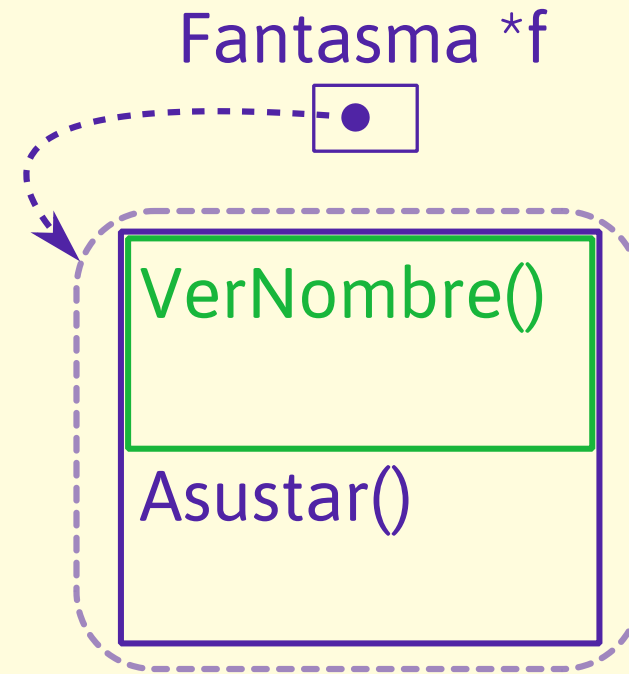
```
class Personaje {  
public:  
    string VerNombre();  
};  
  
class Fantasma : public Personaje {  
public:  
    string Asustar();  
};  
  
int main() {  
    Personaje *p = new Personaje();  
    cout << p->VerNombre(); // Ok  
    cout << p->Asustar(); // Error  
    ...  
}
```



⚠ Con un puntero a *Personaje* solo se pueden invocar métodos de *Personaje*

POLIMORFISMO EN C++

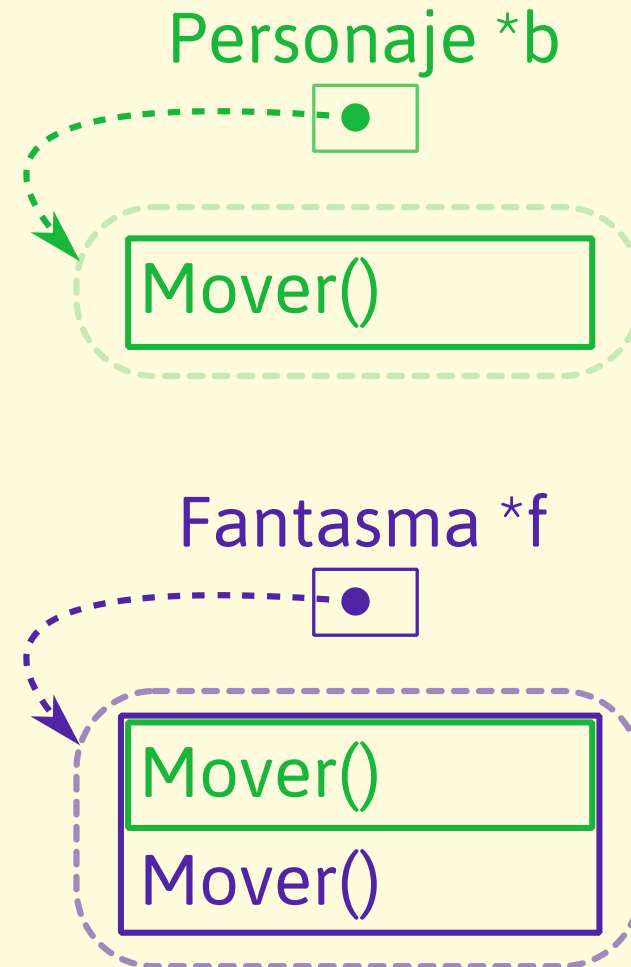
```
class Personaje {  
public:  
    string VerNombre();  
};  
  
class Fantasma : public Personaje {  
public:  
    string Asustar();  
};  
  
int main() {  
    Fantasma *f = new Fantasma();  
    cout << f->VerNombre(); // Ok  
    cout << f->Asustar(); // Ok  
    ...  
}
```



! Con un puntero a *Fantasma* se pueden invocar métodos de ambas

POLIMORFISMO EN C++

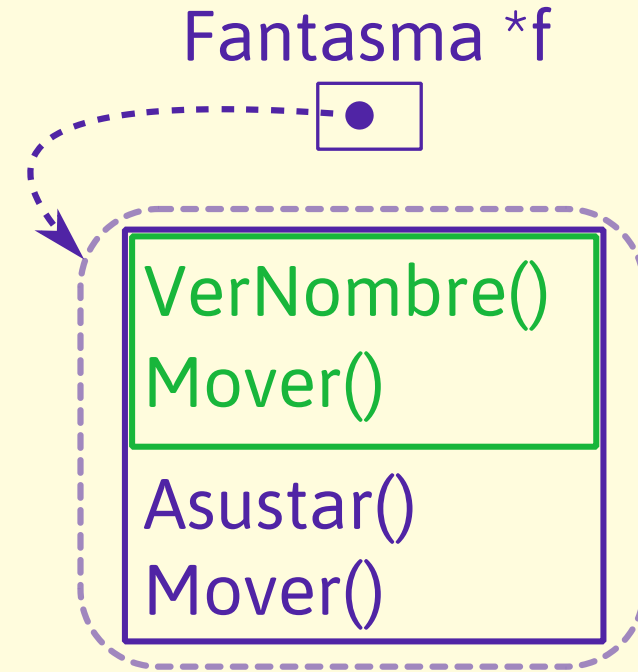
```
class Personaje {  
public:  
    void Mover();  
};  
  
class Fantasma : public Personaje {  
public:  
    void Mover();  
};  
  
int main() {  
    Personaje *p = new Personaje();  
    p->Mover();  
    Fantasma *f = new Fantasma();  
    f->Mover();  
}
```



! Cuando un **método regular** está en ambas clases, **el tipo de puntero determina** a cual se invoca

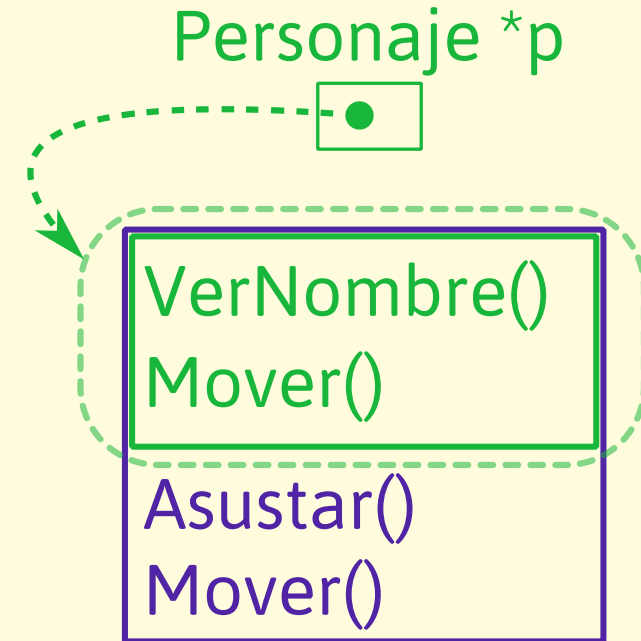
POLIMORFISMO EN C++

```
class Personaje {  
public:  
    void Mover();  
};  
  
class Fantasma : public Personaje {  
public:  
    void Mover();  
};  
  
int main() {  
    Fantasma *f = new Fantasma();  
    f->Mover();  
    ...  
}
```



POLIMORFISMO EN C++

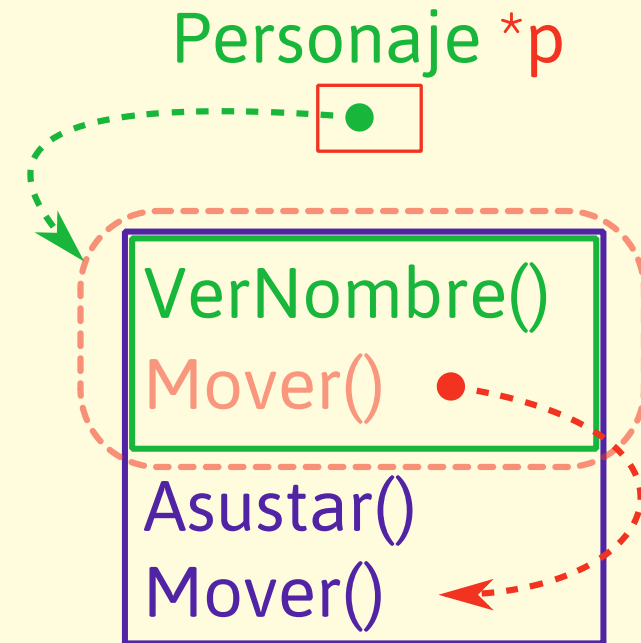
```
class Personaje {  
public:  
    void Mover();  
};  
  
class Fantasma : public Personaje {  
public:  
    void Mover();  
};  
  
int main() {  
    Personaje *p = new Fantasma();  
    p->Mover();  
    ...  
}
```



⚠ Cuando un método **regular** está en ambas clases, **el tipo de puntero** determina a cual se invoca

POLIMORFISMO EN C++

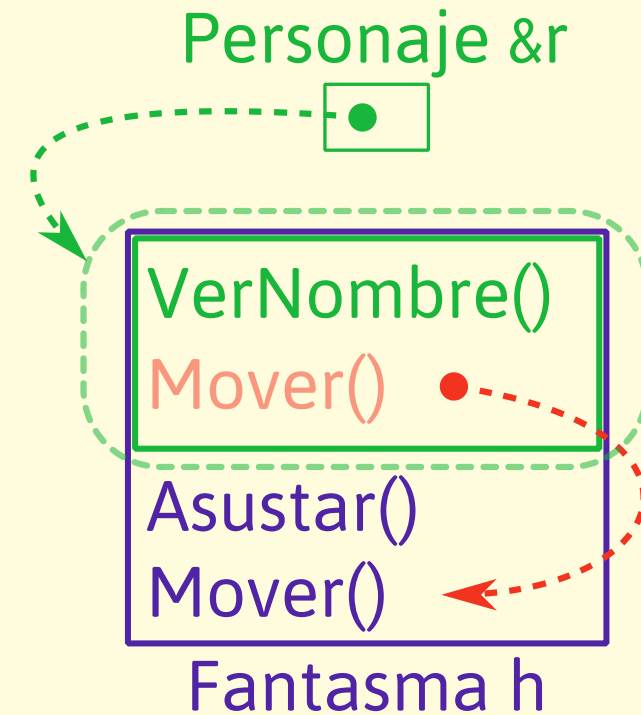
```
class Personaje {  
public:  
    virtual void Mover() = 0;  
};  
  
class Fantasma : public Personaje {  
public:  
    void Mover() override;  
};  
  
int main() {  
    Personaje *p = new Fantasma();  
    p->Mover();  
    ...  
}
```



! Cuando el método es **virtual**, el **tipo del dato apuntado** determina a cual se invoca

POLIMORFISMO EN C++

```
class Personaje {  
public:  
    virtual void Mover() = 0;  
};  
  
class Fantasma : public Personaje {  
public:  
    void Mover() override;  
};  
  
int main() {  
    Fantasma h;  
    Personaje &r = h;  
    r.Mover();  
}
```



⚠️ *Aplica tanto a punteros como a referencias*

POLIMORFISMO EN C++

```
class Personaje {  
public:  
    virtual void Mover();  
};  
  
class Fantasma : public Personaje {  
public:  
    void Mover() override;  
};  
  
int main() {  
    Fantasma h;  
    Personaje b = h; // solo copia la parte "Personaje"  
    b.Mover();  
}
```

Fantasma h

VerNombre()
Mover()

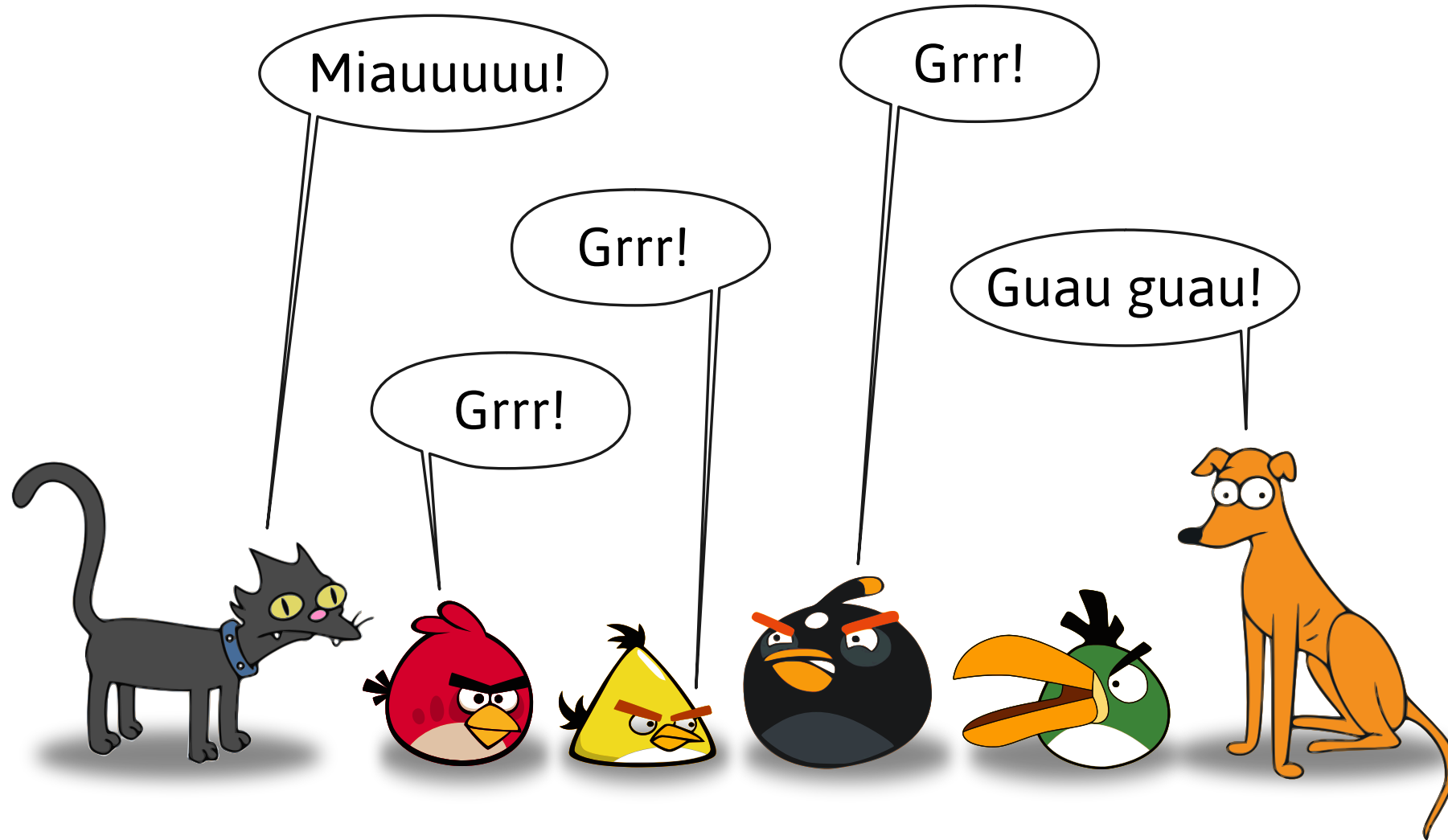
Asustar()
Mover()

Personaje b

VerNombre()
Mover()

! ...pero no a instancias "regulares"

POLIMORFISMO - EJEMPLO ANIMALES



MÉTODOS VIRTUALES

- ▶ En la clase base:
 - ▶ Se declara anteponiendo la palabra clave **virtual**.
 - ▶ Si no se provee implementación, se iguala a cero.
 - ▶ En este caso se dice que es un método **virtual puro**.

Si una clase tiene al menos un método virtual puro se denomina **clase abstracta** y no puede ser instanciada.

MÉTODOS VIRTUALES

- ▶ En la clase hija:
 - ▶ Se implementa respetando el prototipo:
 - ▶ tipos de argumentos y de retorno deben coincidir.
 - ▶ Opcionalmente se agrega la palabra clave **override** al final del prototipo:
 - ▶ indica al compilador la intención de definir/redefinir un método virtual.

POLIMORFISMO "DINÁMICO"

```
Base *b;  
if (...) b = new Hija1();  
else     b = new Hija2();  
...  
b->AlgunMetodoVirtual();
```

A este tipo de polimorfismo se lo denomina **polimorfismo dinámico** porque el método a invocar se define en **tiempo de ejecución**

POLIMORFISMO Y DESTRUCTORES

```
Base *b;  
if (...) b = new Hija1();  
else     b = new Hija2();  
...  
delete b; // ¿cual destructor se ejecuta?
```

Toda clase base con uno o más métodos virtuales debería declarar a su destructor también como virtual (aún cuando no haga nada):

```
class Base {  
    ...  
public:  
    ...  
    virtual ~Base() {}  
};
```

POLIMORFISMO - RESUMIENDO...

- ▶ Debe existir una relación de herencia.
- ▶ El método debe:
 - ▶ declararse como **virtual** en la clase base.
 - ▶ debe implementarse en la clase hija.
 - ▶ invocarse a través de un puntero o referencia.

```
class Base {  
    public: virtual void Metodo() = 0;  
};  
class Hija : public Base {  
    public: void Metodo() override { ... }  
};  
void miFuncion(Base &ref) {  
    ref.Metodo();  
}
```

POLIMORFISMO - EJEMPLO 1

```
class Personaje {  
    ...  
public:  
    virtual void Mover() = 0;  
    ...  
};  
  
class Juego {  
    ...  
    vector<Personaje*> vp;  
public:  
    ...  
    void MoverTodos() {  
        for (size_t i=0; i<vp.size(); ++i)  
            vp[i]->Mover();  
    }  
    ...  
}
```

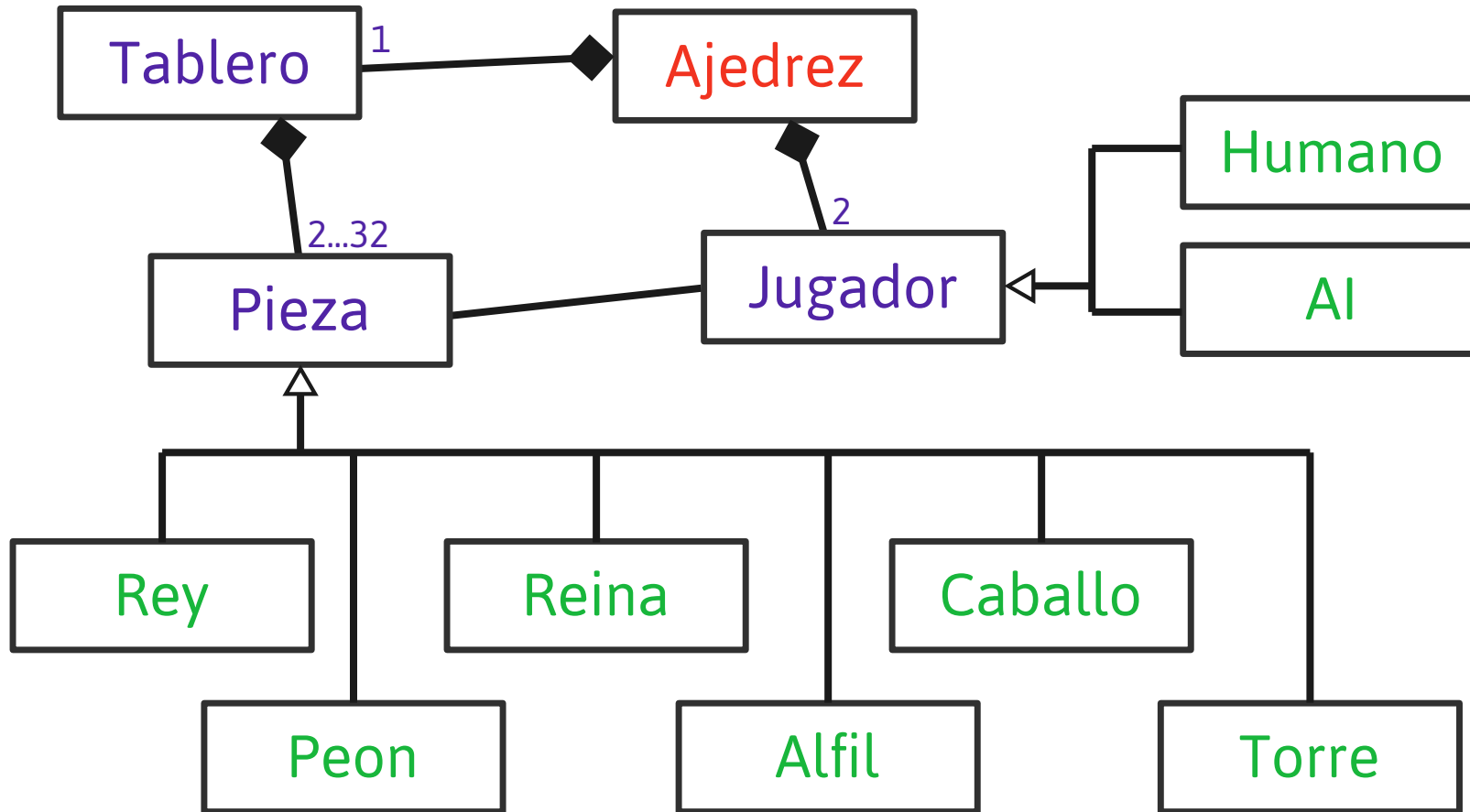
POLIMORFISMO - EJEMPLO 2

```
class Control {
public:
    virtual void Dibujar();
    virtual void ProcesarClick(int x, int y);
    virtual void ProcesarTecla(char k);
    ...
};

class Boton : public Control {...};
class Texto : public Control {...};
class Combo : public Control {...};
class Lista : public Control {...};
class Imagen : public Control {...};

class Ventana {
    vector<Control*> controles;
    ...
};
```

POLIMORFISMO - EJEMPLO 3



POLIMORFISMO - EJEMPLO 3

```
struct Movimiento {
    Pieza *p;
    int f,c;
};

class Jugador {
    ...
public:
    virtual Movimiento Mover(Tablero &t) = 0;
    ...
};

class Ajedrez {
    Tablero tab;
    Jugador *jug_blancas;
    Jugador *jug_negras;
public:
    ...
}
```

POLIMORFISMO - EJEMPLO 3

```
void Ajedrez::Jugar() {  
  
    bool turno_blancas = true;  
  
    while(!Termino()) {  
        // un Jugador decide su movimiento  
        Movimiento mov;  
        if (turno_blancas)  
            mov = jug_blancas->Mover(tab);  
        else  
            mov = jug_negras->Mover(tab);  
  
        // Ajedrez valida si el mov es legal  
        // y si lo es aplica los cambios  
        if (AplicarMovimiento(mov))  
            turno_blancas = !turno_blancas;  
  
    }
```

POLIMORFISMO - EJEMPLO 3

```
class Pieza {  
    ...  
public:  
    virtual bool MovimValido(int f, int c) = 0;  
    ...  
};  
  
class Ajedrez {  
    ...  
public:  
    bool AplicarMovimiento(Movimiento m){  
        if (m.p->MovimValido(m.f,m.c)) {  
            // ... mover, comer, etc ... //  
            return true;  
        } else  
            return false;  
    }  
    ...  
}
```