Programación 3

Orientación a objetos con **C#**

3. Clases, interfaces, herencia, UML

maurogullino@gmail.com

Definiciones (ya conocidas)

- una **clase** es un molde para fabricar **objetos**
- un objeto es el conjunto datos + funcionalidad
- la **interfaz** de un objeto es el conjunto de sus métodos públicos
- el sistema se diseña como un conjunto de objetos que se intercambian **mensajes**

Espacios de nombres *(namespaces)*

```
namespace hola  {
    namespace chau  {
        class Probando {
            public Probando()   {
                Console.WriteLine("constructor");
            }
        }
    }
}

// Probando p = new hola.chau.Probando();
```

Accesibilidad de miembros

```
namespace Jueguito
{
    class Enemigo
    {
        private int energia;
        private int posx;
        private int posy;

        public Enemigo(int x, int y)
        {
            posx = x;
            posy = y;
        }
    }
```

```
static void Main()
{
    Enemigo e1 = new Enemigo(10, 20);
    e1.posx = 10;  //error, es privado
}
```

getter y setter clásicos

```
public int getX()
{
    return posx;
}

public void setX(int x)
{
    posx = x;
}
```

C# accesors

```
class Enemigo  {
        private int energia;
        private int posx;
        private int posy;

        public int X  {
            get
            {
                return posx;
            }

            set
            {
                posx = value;
            }
        }
```

fields

property

lectura

escritura

```
static void Main()
{
      Enemigo e1 = new Enemigo(10, 20);

      e1.X = 40;   //set

      Console.WriteLine( e1.X );  //get
}
```

C# propiedades auto-implementadas

```
class Enemigo  {

        public int X  { get; set; }

        public int Y  { get; set; }

        public int Energia  { get; set; }

}
```

Herencia

```
static void Main()   {
      Empleado e1 = new Empleado("Pepe", 15000);

      Console.WriteLine(e1.Nombre); //get
}
```

```
namespace Empresa  {
    class Empleado  {
        private string nombre;
        private decimal sueldo;

        public string Nombre
        {
            get  {
                return nombre;
            }
        }

        //qué hacemos con el sueldo?

        public Empleado(string n, decimal s)
        {
            nombre = n;
            sueldo = s;
        }
```

```
class Empleado  {

     // ...

     public decimal calcularSueldo(int mes)   {
          return sueldo;
     }
```

Herencia

- crear la clase EmpleadoPorHoras
- queremos **modificar** algo del comportamiento (cálculo del sueldo)
- queremos **mantener** la jerarquía, es decir, que sigan siendo de *tipo* Empleado (*polimorfismo!*)

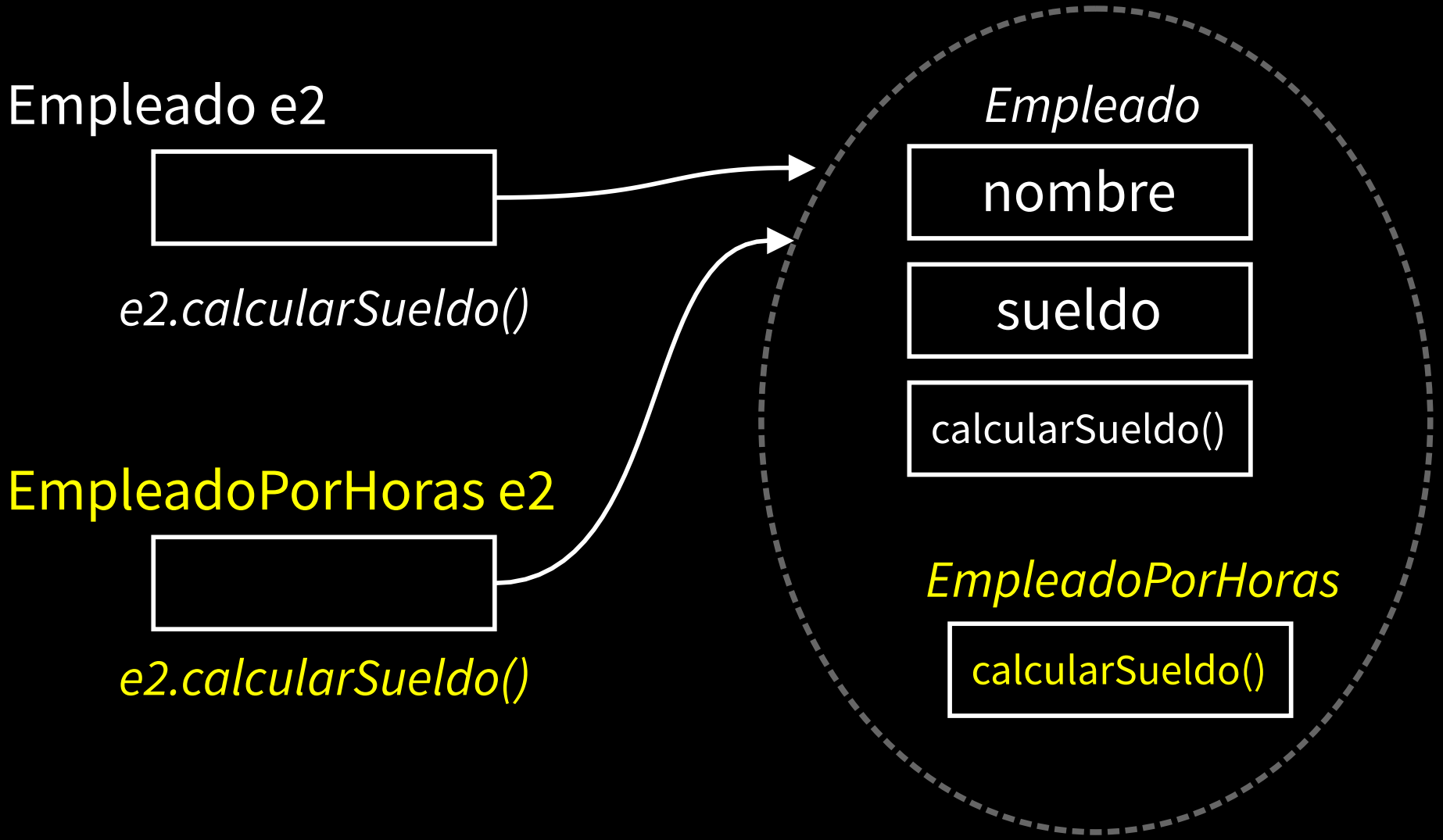
```
namespace Empresa  {

    class EmpleadoPorHoras : Empleado  {
        private decimal valorHora;
        private int horasTrabajadas;

        //constructor y llamada a clase base
        public EmpleadoPorHoras(string n, decimal vh,
                                int ht) : base(n,0) {
            valorHora = vh;
            horasTrabajadas = ht;
        }
```

```
static void Main()  {
  Empleado e1 = new Empleado("Pepe", 15000);
  Console.WriteLine(e1.Nombre);
  Console.WriteLine(e1.calcularSueldo(8) );

  EmpleadoPorHoras e2 = new EmpleadoPorHoras("Juan",
                                             220, 100);
  Console.WriteLine(e2.Nombre);
  Console.WriteLine(e2.calcularSueldo(8)); //porq da 0?

  Console.WriteLine(e2 is Empleado);
```

```
namespace Empresa  {

    class EmpleadoPorHoras : Empleado  {

        // ...

        public decimal CalcularSueldo(int mes)
        {
            return valorHora * horasTrabajadas;
        }


        // anda ok, pero...
```

```
static void Main()  {
  // ...

  Empleado e2 = new EmpleadoPorHoras("Juan",
                                     220, 100);
  Console.WriteLine(e2.Nombre);
  Console.WriteLine(e2.calcularSueldo(8)); //anda?

  Console.WriteLine(e2 is Empleado);
```

¿Por qué?


Empleado e2
e2.calcularSueldo()
EmpleadoPorHoras e2
e2.calcularSueldo()
Empleado
nombre
sueldo
calcularSueldo()
EmpleadoPorHoras
calcularSueldo()

Polimorfismo

- para elegir el método según el tipo del **objeto** es necesario declarar al método **virtual**
- si no, se elige con el tipo de la **referencia**
- se determinará en tiempo de ejecución
- el llamante no se entera de la clase derivada
- importante en colecciones

```
//en Empleado
public virtual decimal CalcularSueldo(int mes)
{
     return sueldo;
}


//en EmpleadoPorHoras
public override decimal CalcularSueldo(int mes)
{
     return valorHora * horasTrabajadas;
}
```

Ejemplo

```
static void Main()   {
     Empleado[] emps = new Empleado[5];

     emps[0] = new Empleado("Pepe", 15000);
     emps[1] = new Empleado("Juan", 18000);
     emps[2] = new Empleado("Maria", 20000);
     emps[3] = new EmpleadoPorHoras("Roberto", 150, 10);
     emps[4] = new EmpleadoPorHoras("Ana", 300, 20);

     foreach(Empleado e in emps)  {
          Console.WriteLine(e.Nombre + " " +
                            e.CalcularSueldo(1)  );
     }
```

Interfaces y clases abstractas

Interfaces

- nos permite garantizar el polimorfismo
- el lenguaje fuerza los métodos públicos que deben existir, o no compila

```
namespace Empresa
{
    interface IPagableMensual
    {
        decimal Monto(int mes);
        string Descripcion();
    }
}

//obviamente los métodos son public
```

```
class Empleado : IPagableMensual  {


class EmpleadoPorHoras :
            Empleado, IPagableMensual  {


//compila?
```

```
//en ambas: Empleado y EmpleadoPorHoras


public decimal Monto(int mes) {
    return CalcularSueldo(mes);
}

public string Descripcion() {
    return Nombre;
}

//será buena idea tener código repetido?
// (tiene solución)
```

```
namespace Empresa  {
    class Factura : IPagableMensual  {
        int mes;
        decimal monto;
        string proveedor;

        public Factura(string p, int me, decimal mo)  {
            proveedor = p;
            monto = mo;
            mes = me;
        }

        public decimal Monto(int m)  {
            if (m == mes) return monto;
            else return 0;
        }

        public string Descripcion() {
            return proveedor;
```

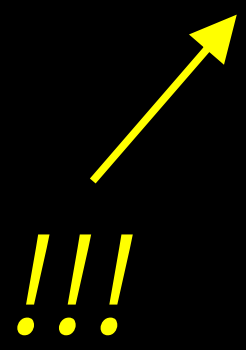
```
static void Main()  {
 IPagableMensual[] deudas = new IPagableMensual[5];

 deudas[0] = new Empleado("Pepe", 15000);
 deudas[1] = new Empleado("Maria", 20000);
 deudas[2] = new EmpleadoPorHoras("Roberto", 150, 10);
 deudas[3] = new EmpleadoPorHoras("Ana", 300, 20);
 deudas[4] = new Factura("Ferreteria Cacho", 1, 890.50m);

 foreach(IPagableMensual d in deudas)
 {
     Console.WriteLine(d.Descripcion() + " " + d.Monto(1));
 }
```

```
namespace Empresa  {

   class Tesoreria  {
     public void ImprimirPagos(IPagableMensual[] pagar,
                                                 int mes)
     {
        foreach (IPagableMensual p in pagar)
        {
           Console.WriteLine(p.Descripcion() + " " +
                                  p.Monto(mes));
        }

     }

   }
```

```
static void Main()  {
  IPagableMensual[] deudas = new IPagableMensual[5];

  deudas[0] = new Empleado("Pepe", 15000);
  deudas[1] = new Empleado("Maria", 20000);
  deudas[2] = new EmpleadoPorHoras("Roberto", 150, 10);
  deudas[3] = new EmpleadoPorHoras("Ana", 300, 20);
  deudas[4] = new Factura("Ferreteria Cacho", 1, 890.50m);

  Tesoreria t = new Tesoreria();
  t.ImprimirPagos(deudas, 1);
```

Solución al código repetido

- clases abstractas ~ interfaces
- pueden contener implementación
- lo contrario a *abstracto* es *concreto*




```
abstract class EmpleadoBase : IPagableMensual
    {
        protected string nombre;
        protected decimal sueldo;
        public string Nombre
        {
            get  {  return nombre; }
        }

        public EmpleadoBase(string n, decimal s)
        {
            nombre = n;
            sueldo = s;
        }

        public abstract decimal CalcularSueldo(int mes);

        // sigue...
```

```
// sigue abstract class EmpleadoBase : IPagableMensual


        public decimal Monto(int mes)
        {
            return CalcularSueldo(mes);
        }

        public string Descripcion()
        {
            return Nombre;
        }
```

```
class Empleado : EmpleadoBase {

    public override decimal CalcularSueldo(int mes) {
        return sueldo;
    }

    public Empleado(string n, decimal s)
                                    : base(n, s) { }


}
```

```
EmpleadoBase eb = new EmpleadoBase();

//no se puede crear una instancia de
//una clase abstracta!

//sólo se puede heredar de ella
```

protected ó private ?

private

· no accesible desde fuera de la clase

· no accesible desde clases derivadas

protected

· no accesible desde fuera de la clase

· accesible desde las clases derivadas

internal

- es como *public* pero dentro del mismo *assembly*

Clases y métodos *sealed*

- no se puede heredar / override
- parece lo “contrario” a abstracto
- en Java se llama *“final”*

```
sealed class EmpleadoPorComision {
        //...


class EmpleadoTrucho : EmpleadoPorComision {
        //no compila
```

Sobrecarga

- **de métodos**
  cuando hay varios métodos con el mismo nombre y distintos parámetros

- **de operadores**
  permite redefinir el comportamiento de algunos operadores del lenguaje

de métodos

```
class Tesoreria  {
    public void ImprimirPagos(IPagableMensual[] pagar, int mes)  {
       foreach (IPagableMensual p in pagar)
       {
          Console.WriteLine(p.Descripcion() + " " + p.Monto(mes));
       }
    }

    public void ImprimirPagos(IPagableMensual[] pagar)  {
       for (int i = 1; i <= 12; i++)
       {
          Console.WriteLine("--- Mes " + i);
          foreach (IPagableMensual p in pagar)
          {
              if(p.Monto(i)>0)
                Console.WriteLine(p.Descripcion() + " " + p.Monto(i));
          }
       }
    }
```

de operadores

```
class Factura : IPagableMensual {
    //....

    public static bool operator < (Factura a, Factura b)  {
        if (a.monto < b.monto) return true;
        else return false;
    }

    public static bool operator > (Factura a, Factura b) {
        if (a.monto > b.monto) return true;
        else return false;
    }

    public static Factura operator + (Factura a, Factura b) {
        //if a.mes != b.mes  tenemos problemas!

         return new Factura(a.proveedor + "+" + b.proveedor,
                                        a.mes, a.monto + b.monto);
    }
```

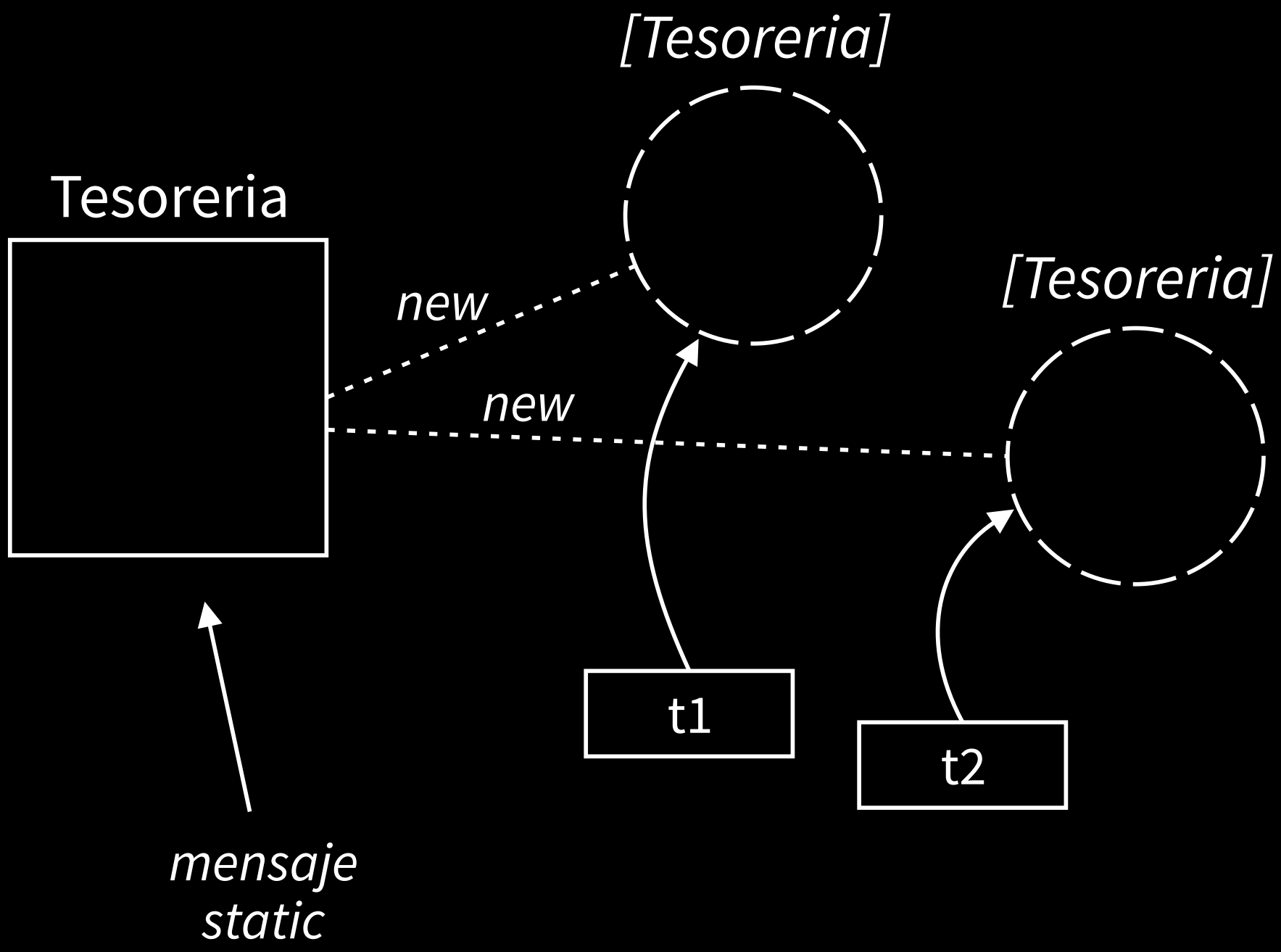
```
Factura f1 = new Factura("Ferreteria Cacho", 1, 890.50m);
Factura f2 = new Factura("Spa Tecnológico", 1, 200m);
Factura f3 = f1 + f2;

Console.WriteLine(f1 < f2);
Console.WriteLine(f3);


//elevamos el nivel de abstracción!
```

Miembros static

```
Tesoreria t = new Tesoreria();
t.ImprimirPagos(deudas, 1);
```

- ¿tiene sentido instanciar para hacer la llamada?
- Tesoreria conserva algún estado?

- en la OOP también podemos mandarle mensajes a las clases (ref. SmallTalk)


[Tesoreria]
Tesoreria
new
[Tesoreria]
new
t1
t2
mensaje
static

```
// ...
Tesoreria.ImprimirPagos(deudas, 1); //mensaje directo a la clase



static class Tesoreria
    {
        public static void ImprimirPagos(IPagableMensual[] pagar,
                                                int mes)  {

            foreach (IPagableMensual p in pagar)
            {
                // .....
```

- si marcamos *static class* no se podrá instanciar
- pueden mezclarse miembros estáticos con miembros de instancia
- pueden haber variables estáticas
- las instancias pueden acceder a miembros static pero no al revés

```
class Monto  {
   const decimal cambio = 20.50m;
   private decimal pesos;
   private decimal dolares;

   public Monto(decimal pesos)  {
      this.pesos = pesos;
   }

   public decimal Pesos {
      get { return pesos; }
      set { pesos = value; }
   }

   public decimal Dolares {
      get { return PesoADolar(pesos); }
   }
```

```
    public static decimal PesoADolar(decimal p) {
        return p / cambio;
    }

    public static decimal DolarAPeso(decimal d) {
        return d * cambio;
    }

}


// main
Monto m = new Monto(300);
Console.WriteLine(m.Pesos);
Console.WriteLine(m.Dolares);
Console.WriteLine( Monto.DolarAPeso(100) );
```

```
class Contador {
    private static int sucesos = 0;

    public int VerContador() {
        return sucesos;
    }

    public static Contador operator ++ (Contador c) {
        sucesos++;
        return c;
    }
}
```

```
static void Main()
{
     Contador c1 = new Contador();
     Contador c2 = new Contador();

     c1++;
     c2++;

     Console.WriteLine(c1.VerContador());
     Console.WriteLine(c2.VerContador());
}
```

Diagramas UML

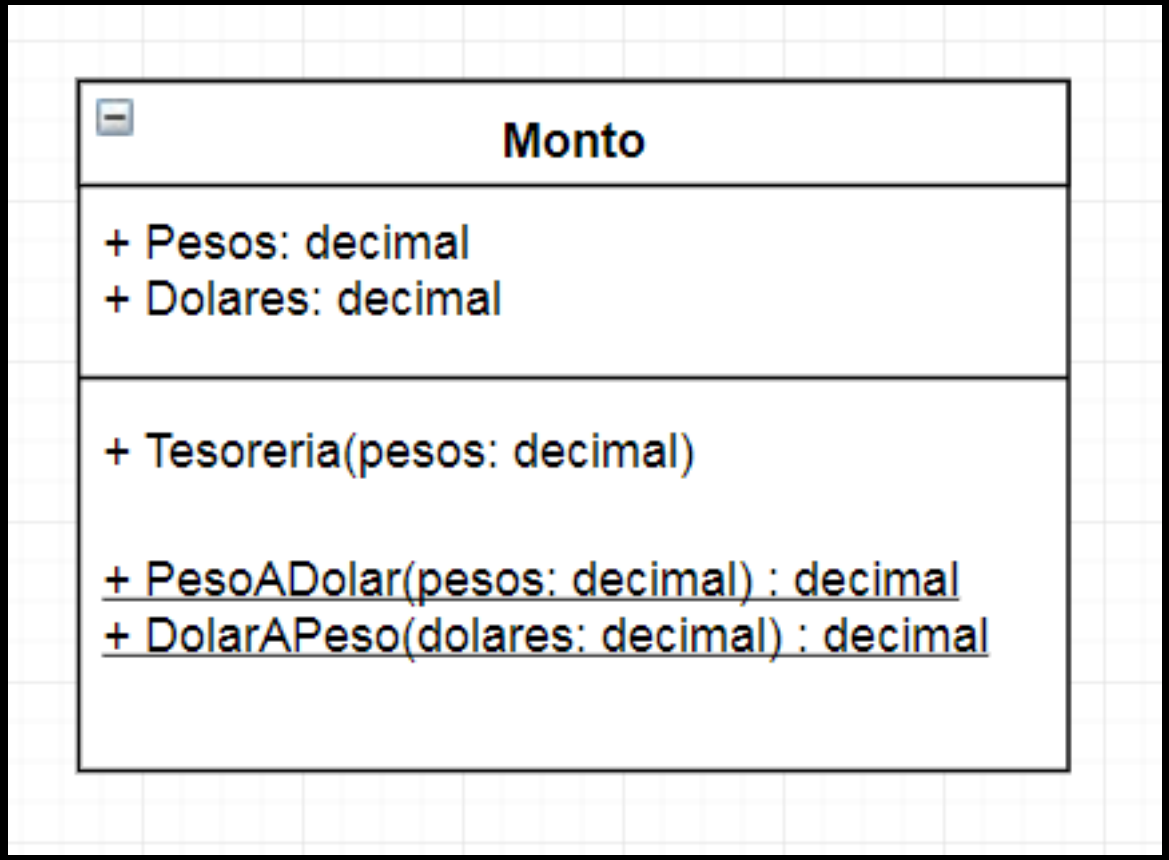
Unified Modeling Language

- es un lenguaje visual (semántica + sintaxis)
- se construye un modelo: representación de otra cosa (software, personas, procesos)
- surge de los primeras metodologías OOP

- se usa para:
  - diseño de software
  - documentación
  - comunicación de procesos

Diagrama de clases

+ publico
- private
protected



nombre

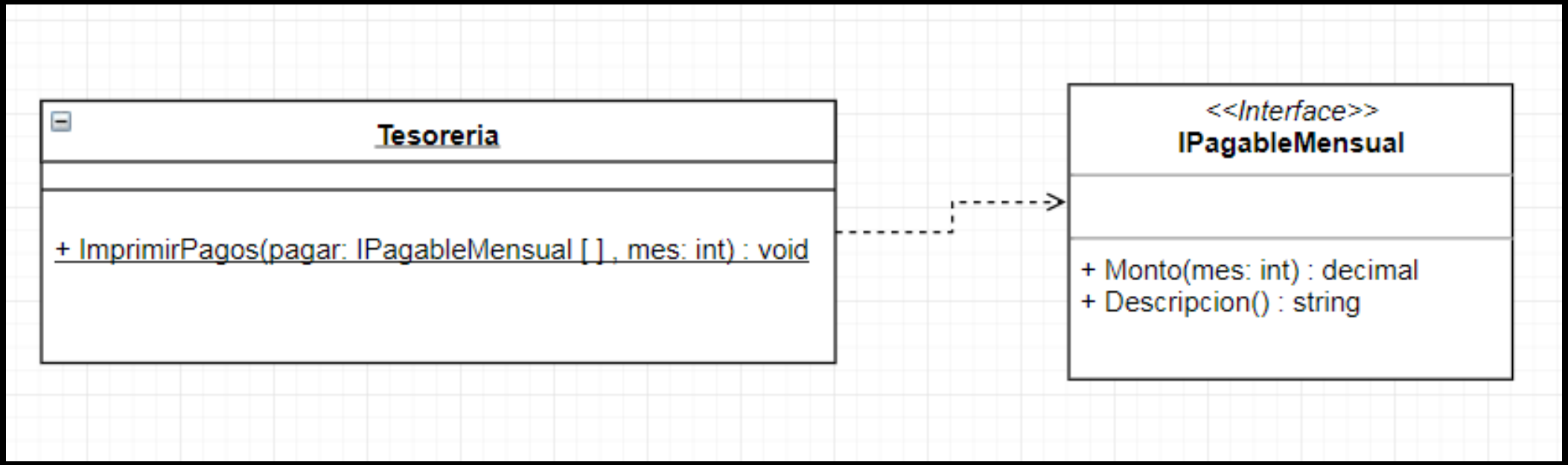
propiedades

métodos de instancia y estáticos

www.draw.io

clase estática

interfaz



relación de dependencia
"usa un"
(interacción por corto tiempo)

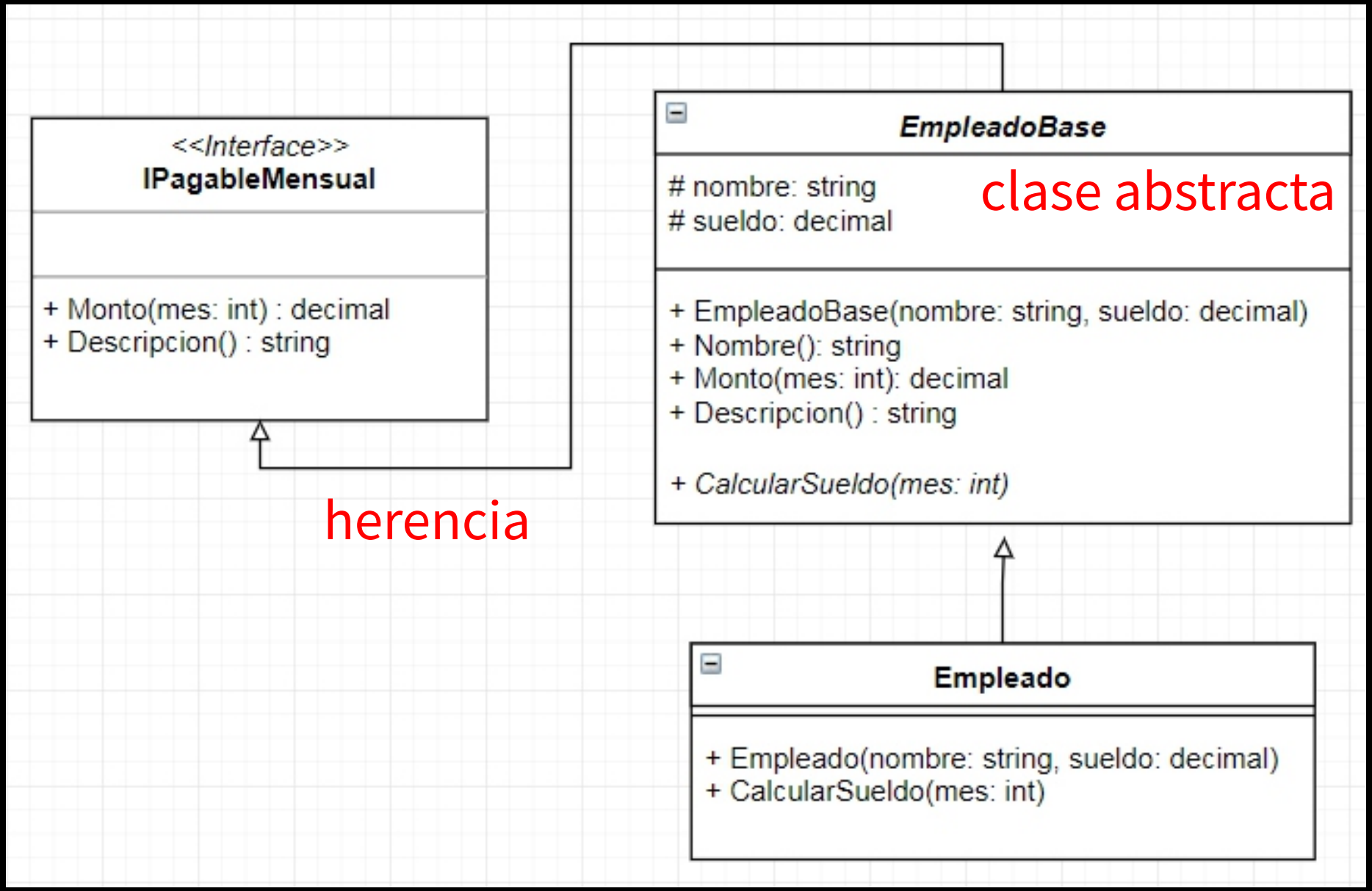
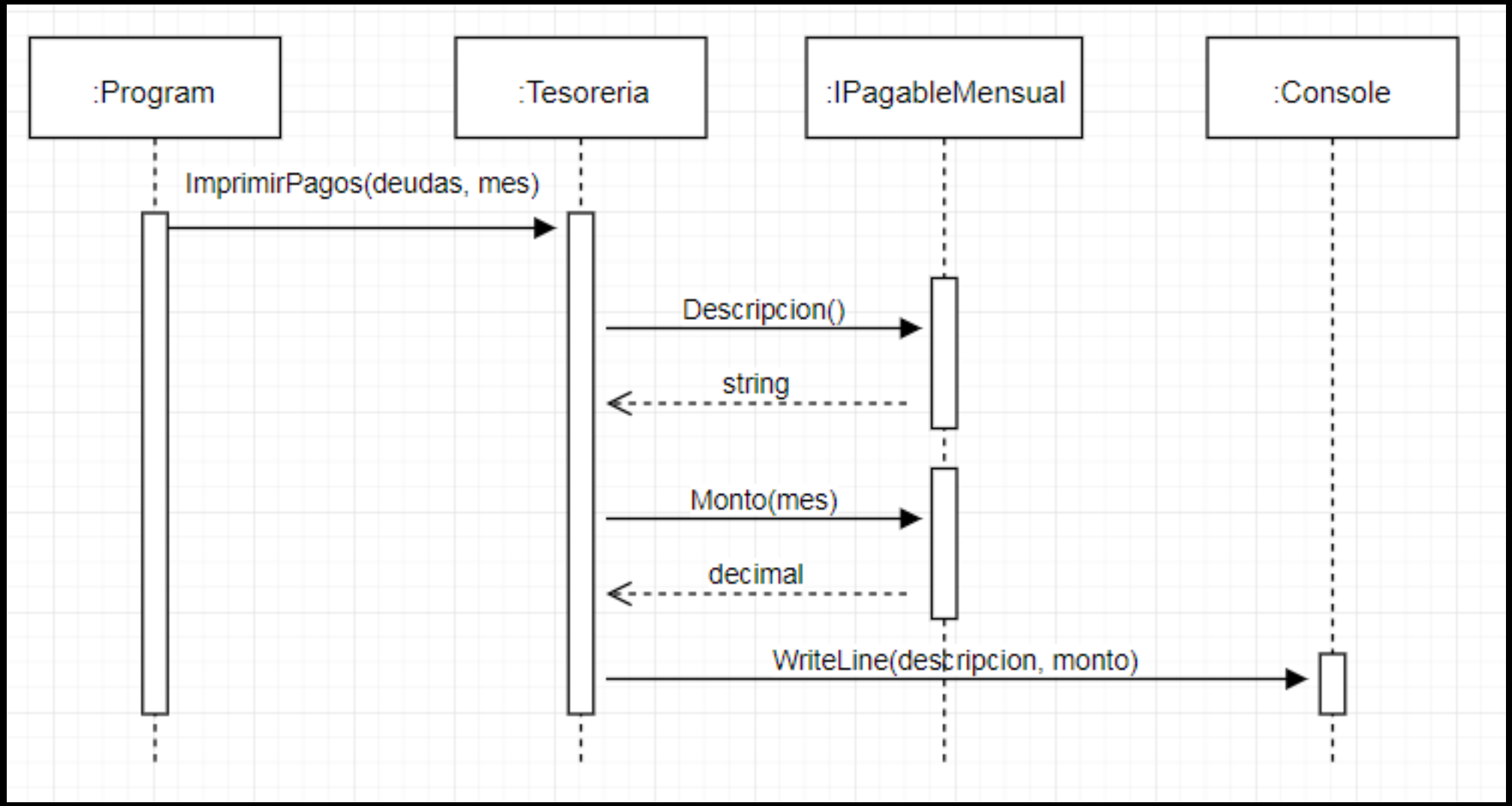
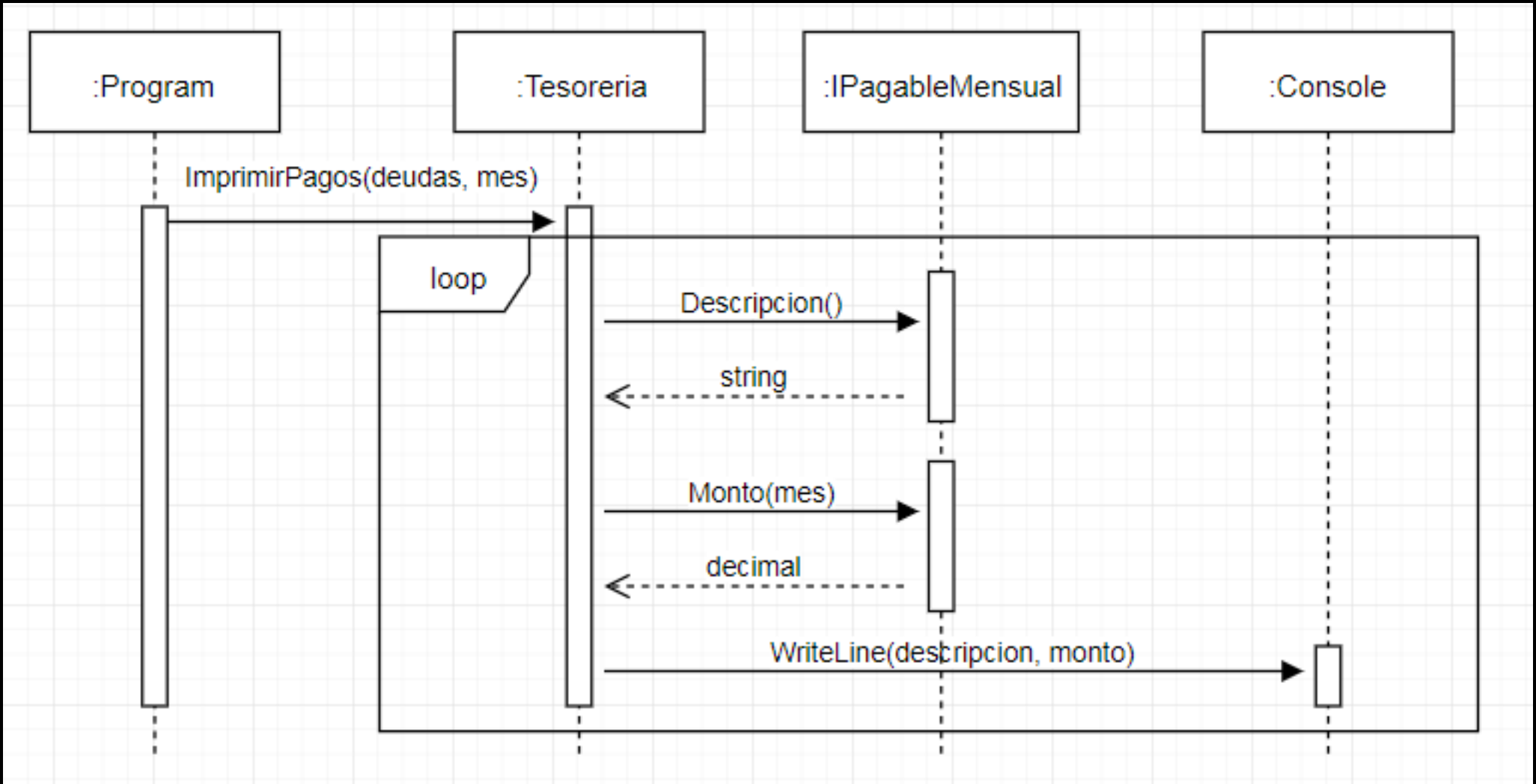

<<Interface>>
IPagableMensual
+ Monto(mes: int) : decimal
+ Descripcion() : string
herencia
EmpleadoBase
nombre: string
sueldo: decimal
clase abstracta
+ EmpleadoBase(nombre: string, sueldo: decimal)
+ Nombre(): string
+ Monto(mes: int): decimal
+ Descripcion() : string
+ CalcularSueldo(mes: int)
Empleado
+ Empleado(nombre: string, sueldo: decimal)
+ CalcularSueldo(mes: int)

Diagrama de interacción




:Program
:Tesoreria
:IPagableMensual
:Console
ImprimirPagos(deudas, mes)
loop
Descripcion()
string
Monto(mes)
decimal
WriteLine(descripcion, monto)

Unified Modeling Language

- casi nada es obligatorio
- si agregamos detalles debe ser por claridad
- es una herramienta de comunicación!