

CONTORNOS DE DESENVOLVEMENTO

TEMA 5:
ELABORACIÓN DE
DIAGRAMAS DE
CLASES

Índice

1.	Introducción a UML e á Orientación a Obxectos.....	3
1.1	Orientación a obxectos	3
1.2	Métodos de modelaxe	13
1.3	Á Linguaxe Unificada de Modelado (UML).....	18
2.	Diagramas de clases.....	27
2.1	Introdución ós diagramas de clases.....	27
2.2	Elementos dos diagramas de clases	28
3.	Ferramentas para o traballo con diagramas de clases.....	42
3.1	Introdución.....	42
3.2	Criterios para a selección dunha ferramenta UML	42
3.3	NetBeans. Módulo UML.....	44
3.4	Visual Paradigm	49

Material complementario e vídeos:

<https://wirtzIDE.blogspot.com/>



Fernando Rodríguez Diéguez

rdf@fernandowirtz.com

Versión 2022-05-05

Traballo derivado por: Fernando Rodríguez Diéguez. rdf@fernandowirtz.com

Con Licenza Creative Commons BY-NC-SA (recoñecemento - non comercial - compartir igual) a partir dos documentos orixinais:

© Xunta de Galicia. Consellería de Cultura, Educación e Ordenación Universitaria.

Autores: María del Carmen Fernández Lameiro, Máximo Fernández López Andrés del Río Rodríguez

Licenza Creative Commons BY-NC-SA (recoñecemento - non comercial - compartir igual).

Para ver unha copia desta licenza, visitar a ligazón <http://creativecommons.org/licenses/by-nc-sa/3.0/es/>.

1. Introdución a UML e á Orientación a Obxectos

1.1 Orientación a obxectos

A programación orientada a obxectos (POO) é un paradigma. Un paradigma de programación representa un enfoque particular ou filosofía para a construcción de software.

Nas últimas décadas, a orientación a obxectos popularizouse enormemente e na actualidade existen un gran número de linguaxes de programación que soportan a orientación a obxectos.

Para o desenvolvemento de software, a POO ten varias vantaxes: fomenta unha metodoloxía baseada en compoñentes de maneira que as aplicacións xéranse como un conxunto de obxectos de tal forma que posteriormente será doado ampliar o sistema simplemente agregándolle funcionalidade aos obxectos xa existentes ou xerando obxectos novos, ademais, de ser necesario, poderán volverse a utilizar os obxectos xa creados cando desenvolva unha nova aplicación, co cal reducirá substancialmente o tempo de desenvolvemento dun sistema.

Para poder desenvolverse con soltura dentro deste paradigma, a parte de coñecer con claridade os conceptos relacionados cos propios obxectos, como *instancia*, *clase*, *atributos* e *métodos*, é necesario comprender outros aspectos tales como *abstracción*, *herdanza*, *polimorfismo* e *encapsulación*. Outros conceptos tamén importantes son: o envío de mensaxes, as asociacións, e a agregación.

Clases e obxectos

Un *obxecto* é a instancia dunha *clase*. De forma más laxa podemos dicir que unha *clase* é un modelo para fabricar *obxectos*.

Unha clase conta cunha estrutura formada por un ou máis datos (*atributos*, *propiedades*, variables de instancia ou variables membro) xunto ás operacións de manipulación de devanditos datos (*métodos* ou accións). Os atributos se implementan mediante variables que almacenarán os datos específicos do obxecto e os métodos se implementan mediante funcións ou procedementos que poderemos invocar para realizar operacións específicas cos obxectos. Normalmente as variables que implementan os atributos son privadas á propia clase (non son accesibles desde fóra) e o acceso ás mesmas realiza-se a través dunhas variables especiais que contan cuns métodos que nos permitirán controlar o acceso aos atributos e realizar comprobacións adicionais. A isto se lle chama normalmente *encapsulado de campos* áínda que algúns autores denominan *propiedades* a estas variables especiais para distinguilas dos *atributos* privados ós cales permiten acceder.

Como exemplo de todo isto imos poñer o caso dos seres humanos. Os seres humanos somos instancias da clase Persoa. Como obxectos da clase Persoa, contamos cos seguintes atributos: altura, peso e idade (podemos imaxinar moitos mais). Tamén realizamos tarefas como: comer, durmir, ler, escribir, falar, traballar, etc. Si tivésemos que crear un sistema que manexase información acerca das persoas sería moi probable que incorporásemos algúns dos seus atributos e accións no noso software.

Para poñer un exemplo¹ de clase software imos a empregar a plataforma *.Net*, nela temos a clase *TextBox* que ten como propiedades *BackColor* para indicar a cor de fondo, *Cursor* para indicar a

¹ Rodríguez Carballal, J.Luis. *Análisis y diseño Orientado a Objetos*. [Presentación PowerPoint]

forma do punteiro do rato cando nos situamos sobre un `TextBox`, etc. Como métodos ten por exemplo `Clear` para borrar todo o texto contido no `TextBox`.

Na nosa aplicación podemos crear distintos obxectos da clase `TextBox`. Cada obxecto pode ter valores diferentes para as distintas propiedades que ten a clase `TextBox`.



Fig. 1

Outro exemplo² de obxecto software podería ser un cadro de texto concreto dunha pantalla, que ten como atributos as coordenadas nas que está situado, o seu ancho e longo, a cor, texto, etc. Operacións que podería realizar serían, por exemplo, cambiar de posición, desactivarse, etc.

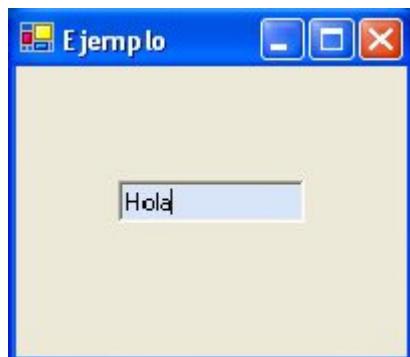


Fig. 2

Construtores e destrutores.

O *construtor* dunha clase é un método estándar para instanciar os obxectos desa clase. É unha función que se executa sempre ao crear un obxecto, o propósito deste procedemento é o de inicializar os datos do obxecto. Os construtores dunha clase teñen sempre o nome da clase e non teñen ningún valor devolto (nin sequera `void`).

Algunhas linguaxes teñen tamén métodos *destrutores*, utilizados para eliminar da memoria un obxecto que xa non se utiliza.

² Rodríguez Carballal, J.Luis. *Análisis y diseño Orientado a Objetos*. [Presentación PowerPoint]

C# e Java posúen recolección de lixos (*garbage collector*) que se encarga de destruír automaticamente os obxectos que non posúen ningunha referencia (que xa non se utilizan).

No seguinte exemplo³ utilízase unha clase chamada Tiempo. Dita clase posúe tres atributos (hora, minuto, segundo; todos variables enteras), e tres métodos (cambiarHora, horaUniversal, horaEstandar):

```
using System;
public class Tiempo
{
    private int hora;      // 0 -23
    private int minuto;    // 0-59
    private int segundo;   // 0-59
    // Constructor de la clase Tiempo que inicialize
    // las variables a cero para poner la hora en media noche
    public Tiempo()
    {
        cambiarHora(0, 0, 0);
    }
    // este metodo asigna una nueva hora en formato 24-horas.
    public void cambiarHora(
        int valorHora, int valorMinuto, int valorSegundo)
    {
        hora = (valorHora >= 0 && valorHora < 24) ?
            valorHora : 0;
        minuto = (valorMinuto >= 0 && valorMinuto < 60) ?
            valorMinuto : 0;
        segundo = (valorSegundo >= 0 && valorSegundo < 60) ?
            valorSegundo : 0;
    }
    // convertir a hora universal con <span id="IL_AD7" class="IL_AD">el
    // metodo</span> format
    public string horaUniversal()
    {
        return String.Format(
            "{0:D2}:{1:D2}:{2:D2}", hora, minuto, segundo);
    }
    // convertir a tiempo estandar (12 horas) usando el metodo format
    public string horaEstandar()
    {
        return String.Format("{0}:{1:D2}:{2:D2} {3}",
            ((hora == 12 || hora == 0) ? 12 : hora % 12),
            minuto, segundo, (hora < 12 ? "AM" : "PM"));
    }
}
```

- Na segunda liña vemos a instrución `public class Tiempo;` isto non é máis que a declaración da nosa clase. As clases polo xeral son de tipo `public`, áinda que si usamos clases internas anónimas é recomendable que sexan `private`.
- Nas liñas 4, 5 e 6 temos a declaración dos atributos da nosa clase.
- Na liña 9 temos a declaración dun construtor. Podemos ter cantos construtores necesitemos, por suposto con diferente tipo de argumentos.
- Nas liñas 14, 25 e 31 están declarados os métodos ou funcións, que permiten realizar operacións sobre o noso obxecto. Por exemplo, o método `cambiarHora` recibe tres argumentos, cos cales modifica os atributos.

Co seguinte código, utilizamos a clase Tiempo:

```
using System;
class PruebaTiempo
```

³ Harvey M. Deitel. *Como programar en C#*. Ed. Prentice Hall. ISBN 9789702610564.

```

    {
        static void Main(string[] args)
        {
            Tiempo tiempo = new Tiempo(); // llamada al constructor de Tiempo
            string salida;
            // mostrar datos iniciales
            salida = "Hora universal inicial es: " +
                tiempo.horaUniversal() +
                "\nHora estandar inicial es: " +
                tiempo.horaEstandar();
            // <span id="IL_AD11" class="IL_AD">cambiar</span> hora (valida)
            tiempo.cambiarHora(13, 27, 6);
            // aniadir nueva hora a la salida
            salida += "\n\nHora universal despues de cambiada: " +
                tiempo.horaUniversal() +
                "\nHora estandar despues de cambiada: " +
                tiempo.horaEstandar();
            // cambiar hora (invalida)
            tiempo.cambiarHora(99, 99, 99);
            salida += "\n\nDespues de poner valores invalidos: " +
                "\nHora universal: " + tiempo.horaUniversal() +
                "\nHora estandar: " + tiempo.horaEstandar();
            Console.WriteLine(salida);
        }
    }
}

```

- Na liña 6 creamos (instanciamos) un obxecto de tipo `tiempo`. Polo xeral en C# a sintaxis para crear obxectos é a seguinte: `nombre_Clase nome_objeto = new nombre_Clase()`. É dicir, o nome da clase, o nome do obxecto, e posteriormente iníciase o obxecto facendo unha invocación ao construtor da clase (antepoñendo a instrución `new`).
- Na liña 10, invocamos un dos métodos da clase (`horaUniversal`). A sintaxe para a invocación dos métodos é: `objeto.nombreMetodo(arg1, arg2, ...)`. Así na liña 14 podemos ver outro exemplo de invocación, na que pasamos algúns argumentos ao método.

Exemplo en C# do uso do Encapsulado de Campos:

```

public class Clase
{
    //atributo
    private int variable;
    //constructor
    public Clase(int variable)
    {
        this.variable = variable;
    }
    //declaracion de la variable para el encapsulado
    public int atributo
    {
        //get sirve para devolver el valor del atributo
        get
        {
            return variable;
        }
        //set sirve para cambiar el valor del atributo
        set
        {
            variable = value;
        }
    }
}

```

- Na liña 11 temos a declaración da variable para o encapsulado; en devandita declaración debemos especificar un nome (diferente ao do atributo ao que imos acceder), e un tipo (neste caso `int`, enteiro). Dentro do encapsulado temos dous bloques: `get` e `set`. Dentro de `get` debemos poñer os valores a devolver cando se acceda á variable; mentres que en `set` podemos usar a palabra clave `value` para asignar un valor ao atributo.

Na seguinte clase utilizase o encapsulado da clase anterior, para obter e modificar os valores do atributo:

```
using System;
public class Propiedades {
    public Propiedades() {
        //declaracion del objeto
        Clase objeto = new Clase(5);
        //obteniendo el valor de la variable 'variable'
        //usando la variable 'atributo' (se usa el bloque get)
        Console.WriteLine("El atributo del objeto es "+ objeto.atributo);
        Console.WriteLine("Cambiando el valor usando la propiedad
'atributo'...");
        //cambiando el valor de la variable 'variable'
        //usando la variable 'atributo' (se usa get)
        objeto.atributo = 10;
        Console.WriteLine("Ahora el atributo del objeto es "+ objeto.atributo);
    }
    static void Main(string[] args) {
        new Propiedades();
    }
}
```

- Desta forma é posible obter e modificar os datos dun atributo privado usando unha variable (propiedade segundo algúns autores) pública, e todo dun xeito totalmente seguro, posto que dentro do encapsulado podemos verificar a consistencia dos datos.

Abstracción

A *abstracción* consiste en illar un elemento do seu contexto ou do resto dos elementos que o acompañan para centrarnos nas características particulares do mesmo que nos axudarán na resolución dun problema.

Na práctica, a *abstracción* permítenos ter as características que precisamos dun obxecto. Se precisamos do obxecto Persoa nun software administrativo, poderíamos poñer o nome, idade, enderezo, estado civil, etc. Pero se o empregamos nun software para o eido da bioloxía dentro dos seus atributos pode ter ADN, RND, Gen_x1, Gen_x2, etc. e os atributos anteriormente mencionados non son necesarios. En xeral, podemos dicir que a persoa ten todos os atributos mencionados aquí pero o proceso de abstracción permítenos eliminar aqueles que non pertencen ou son irrelevantes para o noso sistema.

Máis aló da propia definición, a abstracción é un dos mecanismos máis importantes mediante os cales podemos fazer fronte a sistemas complexos que debamos modelar. A abstracción vainos a permitir simplificar o problema illando os seus distintos elementos para centrarnos tan só nos aspectos relevantes para a solución final, evitando o abrumarse con cada un dos detalles do problema.

O concepto de abstracción a nivel de código conséguese mediante a utilización de *interfaces* ou clases *abstractas*. Ambos son unha especie de contrato de como se debe comportar a clase no mundo exterior. É dicir, é un resumo do comportamento da mesma (sen importar como implemente devandito comportamento). No seguinte exemplo vemos a clase `Rectangulo` que herda de máis dun `Interfaz` (`Figura`, `Dibujable`).

```

interface Dibujable
{
    void Dibujar();
}

class Rectangulo : Figura, Dibujable
{
    public void Dibujar()
    {
        if (Alto > 0)
        {
            string s = "";
            for (int x = 0; x < Ancho; ++x)
                s += "[]";
            Console.WriteLine(s);
            if (Alto > 2)
            {
                for (int y = 0; y < Alto - 2; ++y)
                {
                    s = "[]";
                    for (int x = 0; x < Ancho - 2; ++x)
                        s += " ";
                    if (Ancho > 1)
                        s += "[]";
                    Console.WriteLine(s);
                }
            }
            if (Alto > 1)
            {
                s = "";
                for (int x = 0; x < Ancho; ++x)
                    s += "[]";
                Console.WriteLine(s);
            }
        }
    }
}

```

Encapsulamento

Estreitamente relacionado coa *abstracción* atópase o *encapsulamento*. En programación orientada a obxectos, denomínase *encapsulamento* (encapsulación ou ocultamento) á ocultación dos detalles internos dunha clase, mostrando aos posibles usuarios da clase só o que fai pero non como o fai.

Isto ten dúas vantaxes iniciais: o que fai o usuario da clase pode ser controlado moito mellor, evitando que todo colapse por unha intervención non desexada. A segunda vantage é que, ao facer que a maior parte do código estea oculto, pódense facer cambios e/ou melloras sen que iso afecte o modo como outros programadores vaian a utilizar o código. Só ten que manterse igual a forma de acceder a el. Estas “portas de acceso” para poder interactuar cunha clase son o que anteriormente nomeamos coma *interface*.

Por exemplo, a maioría da xente que ve a televisión non sabe ou non se preocupa da complexidade electrónica que hai detrás da pantalla nin de todas as operacións que teñen que ocorrer para mostrar unha imaxe na pantalla. A televisión fai o que ten que facer sen mostrarnos o proceso necesario para iso e nós interactuamos con ela mediante a súa interface (botóns do mando e/ou botóns da televisión).

Véxanse *exemplos de código anteriores de uso de propiedades e interfaces*.

Herdanza

Como xa mencionamos anteriormente, un obxecto é unha instancia dunha clase. Esta idea ten unha consecuencia importante: como instancia dunha clase, un obxecto ten todas as características da clase da que provén. A isto coñéceselle como *herdanza*. Por exemplo, non importa que atributos e accións decidan usarse dunha clase Lavadora, cada obxecto da clase herdará devanditos atributos e operacións.

Un obxecto non só herda dunha clase, *senón que unha clase tamén pode herdar doutra*.

Por exemplo, as lavadoras, refrixeradores, fornos de microondas, tostadores, lavapratos, radios, licuadoras e pranchas son clases e forman parte dunha clase mais xenérica chamada Electrodomésticos. Un electrodoméstico conta cos atributos de interruptor e cable eléctrico, e as operacións de acceso e apagado. Cada unha das clases Electrodoméstico herda os mesmos atributos; por iso, si sabe que algo é un electrodoméstico, de inmediato saberá que conta cos atributos e accións da clase Electrodoméstico.

Outra forma de explicalo é que a lavadora, refrixerador, forno de microondas e cousas polo estilo son subclases, clases fillas ou clases derivadas da clase Electrodoméstico. Podemos dicir que a clase Electrodoméstico é unha superclase, clase nai ou pai ou clase base de todas as demais.

Cando unha clase herda de varias dise que ten herdanza múltiple. Non todas as linguaxes de programación sopórtana, por exemplo Java e C# non o fan, mentres que C++ si.

A vantaxe principal que nos proporciona a herdanza na programación orientada a obxectos é que axuda aos programadores a aforrar código e tempo, xa que a clase pai pode ser implementada e verificada con anterioridade, e as súas clases fillas herdarán código e datos dela (poderán tamén engadir o seu propio código ou modificar o herdado).

- **Herdanza e visibilidade.** Á hora de crear unha clase, un deseñador pode definir que variables de instancia e métodos dos obxectos dunha clase son visibles e desde onde.

En C# e Java isto conséguese coas especificacións `private`, `protected` e `public` que anteceden á definición das variables e métodos:

`Private`: só é accesible desde dentro da clase.

`Public`: é accesible a todos os obxectos.

`Protected`: é accesible desde as subclases, pero non é accesible nin visible para o exterior.

A continuación vemos o exemplo dunha clase `Trabajador` que herda dunha clase `Persona` e utiliza o construtor de devandita clase para implementar o seu propio.

```
using System;
class Persona
{
    // Campo de cada objeto Persona que almacena su nombre
    public string Nombre;
    // Campo de cada objeto Persona que almacena su edad
    public int Edad;
    // Campo de cada objeto Persona que almacena su NIF
    public string NIF;

    void Cumpleaños() {      // Incrementa en uno la edad del objeto Persona
        Edad++;
    }

    // Constructor de Persona
    public Persona (string nombre, int edad, string nif)
    {
        Nombre = nombre;
        Edad = edad;
        NIF = nif;
    }
}
```

```

class Trabajador: Persona
{
    // Campo de cada objeto Trabajador que almacena cuánto gana
    public int Sueldo;
    Trabajador(string nombre, int edad, string nif, int sueldo)
        : base(nombre, edad, nif)
    {
        // Inicializamos cada Trabajador en base al constructor de Persona
        Sueldo = sueldo;
    }
    public static void Main()
    {
        Trabajador p = new Trabajador("Josan", 22, "77588260-Z", 100000);
        Console.WriteLine ("Nombre=" + p.Nombre);
        Console.WriteLine ("Edad=" + p.Edad);
        Console.WriteLine ("NIF=" + p.NIF);
        Console.WriteLine ("Sueldo=" + p.Sueldo);
    }
}

```

Polimorfismo

O *polimorfismo* refírese á capacidade para que varias clases derivadas dunha antecesora implementen un mesmo método de forma diferente.

Por exemplo, podemos crear dúas clases distintas: Peixe e Ave que herdan da superclase Animal. A clase Animal ten o método abstracto mover que se implementa de forma distinta en cada unha das subclases (peces e aves móvense de forma distinta).

Un concepto que ás veces se confunde co polimorfismo é a *sobrecarga de métodos*. A sobrecarga de métodos permite definir dous ou máis métodos co mesmo nome, pero que difiren en cantidade ou tipo de parámetros. Esta característica da linguaxe facilitános a implementación de algoritmos que cumplen a mesma función pero que difiren nos parámetros. O polimorfismo e a sobrecarga diferéncianse en que:

- A sobrecarga dáse sempre dentro dunha soa clase, mentres que o polimorfismo dáse entre clases distintas.
- No polimorfismo o nome do método e os seus parámetros coinciden, na sobrecarga coincide o nome do método pero nunca os parámetros (poden diferir en cantidade ou no tipo de datos).

A continuación móstrase un exemplo dunha clase `Trabajador` que herda dunha clase `Persona`. `Trabajador` implementa o método `Cumpleaños()` que tamén existe na clase pai. Para identificar que un método está sobrescribindo outro se utiliza a palabra clave `override`.

```

using System;
class Persona {
    public string Nombre;           // Campo de cada objeto Persona que almacena su nombre
    public int Edad;                // Campo de cada objeto Persona que almacena su edad
    public string NIF;              // Campo de cada objeto Persona que almacena su NIF

    public virtual void Cumpleaños() // Incrementa en uno la edad del objeto Persona {
        Edad++;
        Console.WriteLine("Incrementada edad de persona"
    }

    public Persona (string nombre, int edad, string nif) // Constructor de Persona
    {
        Nombre = nombre;
        Edad = edad;
        NIF = nif;
    }
}

```

```

class Trabajador: Persona {
    int Sueldo; // Campo de cada objeto Trabajador que almacena cuánto gana
    Trabajador(string nombre, int edad, string nif, int sueldo): base(nombre, edad, nif)
    {
        // Inicializamos cada Trabajador en base al constructor de Persona
        Sueldo = sueldo;
    }
    public override Cumpleaños()
    {
        Edad++;
        Console.WriteLine("Incrementada edad de trabajador");
    }
    public static void Main()
    {
        Trabajador p = new Trabajador("Josan", 22, "77588260-Z", 100000);
        p.Cumpleaños();
    }
}

```

A mensaxe mostrada por pantalla ao executar este método confirma o antes dito respecto de cal é a versión de `Cumpleaños()` á que se chama, xa que o resultado obtido é:

Incrementada idade de trabajador

Envío de mensaxes

Xa adiantamos que nun sistema os obxectos traballan en conxunto. Isto lógrase mediante o envío de mensaxes entre eles. Un obxecto envía a outro unha mensaxe para realizar unha operación, e o obxecto receptor executará a operación.

O xeito de enviar unha mensaxe a un obxecto desde outro é mediante a invocación dun dos seus métodos. Polo tanto, unha nova forma de considerar aos métodos dun obxecto é como a vía para enviar unha mensaxe ao obxecto e que este reaccione acorde a devandita mensaxe.

Desta forma unha mensaxe estará composto por:

- O obxecto destino, cara ao cal a mensaxe é enviado
- O nome do método a chamar
- Os parámetros solicitados polo método

Relacións entre obxectos

Existen varios tipos de relacións que poden unir aos diferentes obxectos, pero entre elles destacan as relacións de: asociación, agregación/composición, e xeneralización/especialización.

- **Relacións de Agregación/Composición.** Neste tipo de relacións un obxecto compoñente intégrase nun obxecto composto.

A diferenza entre agregación e composición é que mentres que a composición enténdese que dura durante toda a vida do obxecto contenedor, na agregación non ten por que ser así.

- Exemplo de agregación: un ordenador e os seus periféricos. Os periféricos dun ordenador poden estar ou non, pódense compartir entre ordenadores e non son propiedade de ningún ordenador.
- Exemplo de composición: unha árbore e as súas follas. Unha árbore está intimamente ligado ás súas follas. As follas son propiedade exactamente dunha árbore, non se pueden compartir entre árbores e cando a árbore morre, as follas fano con el.

No deseño dun sistema este tipo de relacións adóitanse representar como é-parte-de (part-of) ou ten-un(has-a).

- **Relacións de Xeneralización/Especialización.** É un tipo de relación que xa vimos anteriormente ao falar de Herdanza: ás veces sucede que dúas clases teñen moitas das súas partes en común, o que normalmente se abstrae na creación dunha terceira clase (pai das dúas) que reúne todas as súas características comúns.

Este tipo de relacións é característico da programación orientada a obxectos. En realidade, a xeneralización e a especialización son diferentes perspectivas do mesmo concepto, a xeneralización é unha perspectiva ascendente, mentres que a especialización é unha perspectiva descendente: unha superclase representa unha xeneralización das subclases e unha subclase representa unha especialización da clase superior.

No deseño dun sistema este tipo de relacións adóitanse representar como é-un (is-a), A clase derivada é-un tipo de clase da clase base ou superclase.

- **Relacións de Asociación.** Serían relacións xerais, nas que un obxecto realiza chamadas aos métodos doutro, interactuando con el.

O establecemento dunha asociación define os roles (papeis) ou dependencias entre obxectos de dúas clases e a súa cardinalidade (multiplicidade); é dicir, cantas instancias de cada clase poden estar implicadas nunha asociación.

No deseño dun sistema as asociacións represéntanse por unha liña que une ás dúas clases e o nome da asociación escríbese na liña. Exemplos (traballa para, emprega a, é colaborador de,...).

Vexamos a continuación uns exemplos⁴ en C# de como implementar Asociacións e Composicións.

Como implementar Asociación

Representaremos a relación: O cliente usa tarxeta de crédito.

```
public class Customer
{
    private int id;
    private String firstName;
    private String lastName;
    private CreditCard creditCard;
    public Customer()
    {
        //Lo que sea que el construtor haga
    }
    public void setCreditCard(CreditCard creditCard)
    {
        this.creditCard = creditCard;
    }
    // Más código aquí
}
```

⁴ Padilla Donato, Humberto. *Desarrolla Sw utilizando POO*.

<https://sites.google.com/a/cecyteg.edu.mx/desarrolla-sw-utilizando-poo/atributos-compositivo-y-asociativo>

Como implementar Composición

Representaremos a relación: o portátil ten un teclado.

```
public class Laptop
{
    private String manufacturer;
    private String model;
    private String serviceTag;
    private KeyBoard keyBoard = new KeyBoard();

    public Laptop()
    {
        //Lo que sea que el constructor haga
    }
}
```

Moi similar, pero hai unha gran diferenza: Podemos crear un obxecto de tipo `Customer` e asignarlle un `CreditCard` máis tarde mediante o método `setCreditCard`. `Customer` é polo tanto independente de `CreditCard`.

Pero si creamos un obxecto `Laptop`, de entrada saberemos que terá un teclado xa creado, posto que a variable de referencia `keyBoard` é declarada e inicializada ao mesmo tempo. Non hai momento (non debería) en que a clase contenedora poida existir sen algún dos seus obxectos compoñentes.

Tarefa 5.1. Identificación de elementos da programación orientada a obxectos en código fonte.

Busca en internet anacos de código en distintas linguaxes e identifica os distintos elementos da programación orientada a obxectos: constructores, heranza, polimorfismo, sobrecarga de métodos, asociacions... Preferiblemente non en Java, xa que nesa linguaaxe xa coñecemos a súa sintaxe.

1.2 Métodos de modelaxe

A día de hoxe os diagramas más utilizados nas fases de análise e deseño de calquera proxecto software son os propostos por UML, pero antes de comezar a falar desta linguaaxe de modelado convén realizar un repaso por outras técnicas de representación que foron utilizadas e que en nalgúns casos, como os diagramas entidade-relación, aínda se utilizan no modelado de proxectos software.

Diagrama de fluxo de datos

Un diagrama de fluxo de datos (DFD) é unha representación gráfica do fluxo de datos a través dun sistema de información. Os DFD foron inventados por Larry Constantine, o desenvolvedor orixinal do deseño estruturado.

Os DFD non só pódense utilizar para modelar sistemas de sistemas de proceso de información, senón tamén como xeito de modelar organizacións enteiras, como unha ferramenta para o planeamento estratégico e de negocios.

- Compoñentes

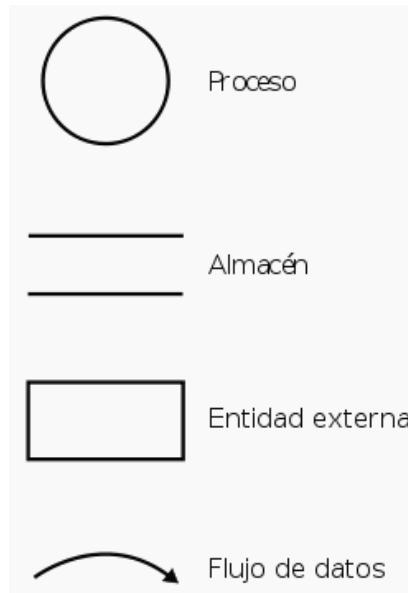


Fig. 3

- *Proceso.* Parte do sistema que transforma entradas en saídas.

Algúns analistas prefieren usar un óvalo ou un rectángulo con esquinas redondeadas, pero estas formas son puramente cosméticas, aínda que obviamente é importante usar a mesma forma de xeito consistente para representar todas as funcións dun sistema.

O nome que se lle asigne xeralmente consiste nunha frase verbo-obxecto tal como Validar Entrada.

- *Fluxo.* - Un fluxo represéntase graficamente por medio dunha frecha que entra ou sae dun proceso. O fluxo úsase para describir o movemento de bloques ou paquetes de información dunha parte do sistema a outra.

O nome representa o significado do paquete que se move ao longo do fluxo.

- *Almacén.* - O almacén utilizase para modelar unha colección de paquetes de datos en repouso. Denótase por dúas liñas paralelas. Os fluxos representan datos en movementos, mentres que os almacéns representan datos en repouso.
- *Entidade externa ou Terminador.* - Graficamente represéntase como un rectángulo. Representan entidades externas coas cales o sistema se comunica. Comunmente encóntrase fóra do control do sistema que está a modelar. Nalgúns casos pode ser outro sistema computacional co cal este se comunica.

- Exemplo⁵

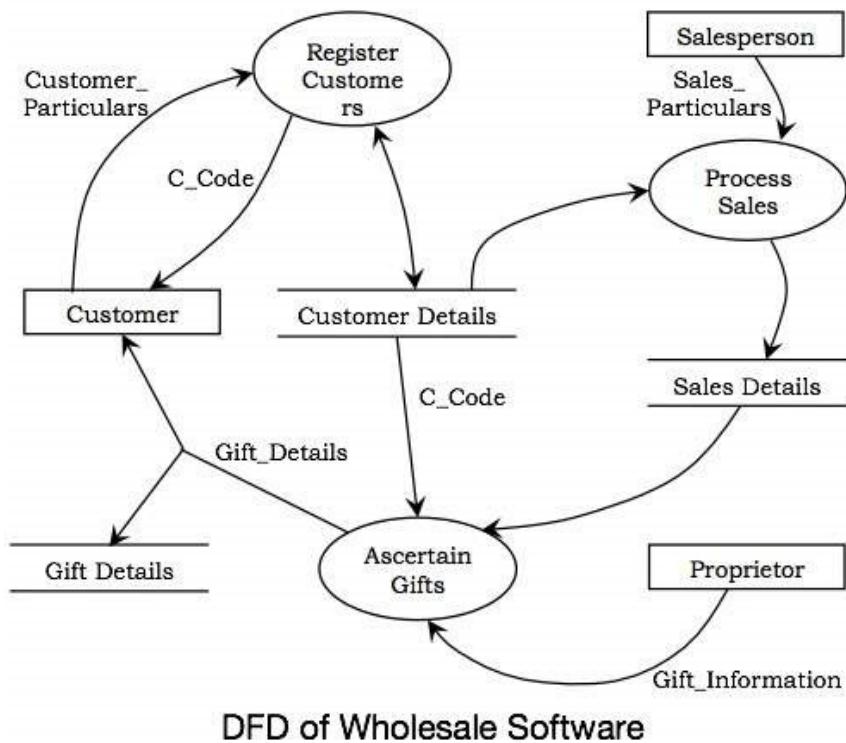
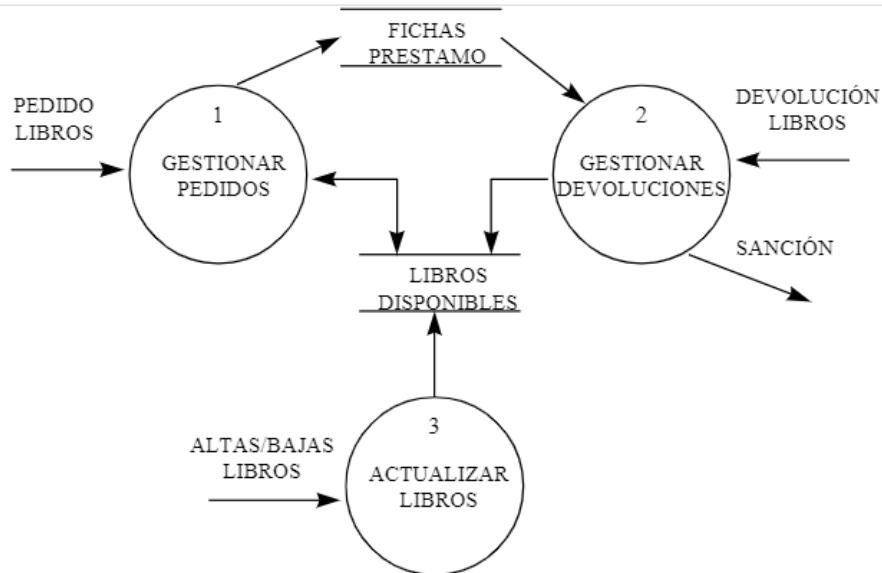


Fig. 4

Tarefa 5.2. Escribe un enunciado que se corresponda co DFD seguinte:



⁵ https://www.tutorialspoint.com/object_oriented_analysis_design/ooad_functional_modeling.htm

Diagramas entidade-relación

Un diagrama ou modelo entidade-relación (ás veces denominado polas súas siglas en inglés, E-R "Entity relationship", ou do español DER "Diagrama de Entidade Relación") é unha ferramenta para o modelado de datos que permite representar as entidades relevantes dun sistema de información así como as súas interrelacións e propiedades.

- **Compoñentes principais:**

- *Entidade*. Representa unha "cousa" ou "obxecto" do mundo real con existencia independente, é dicir, diferénciase univocamente doutro obxecto ou cousa, mesmo sendo do mesmo tipo, ou unha mesma entidade. Represéntase cun rectángulo.
- *Atributos*. Os atributos son as características que definen ou identifican a unha entidade. Estas poden ser moitas, e o deseñador só utiliza ou implementa as que considere máis relevantes. Represéntanse con círculos ou óvalos.
- *Relación*. Describe certa dependencia entre entidades ou permite a asociación destas. Represéntase cun rombo.

- **Exemplo⁶:**

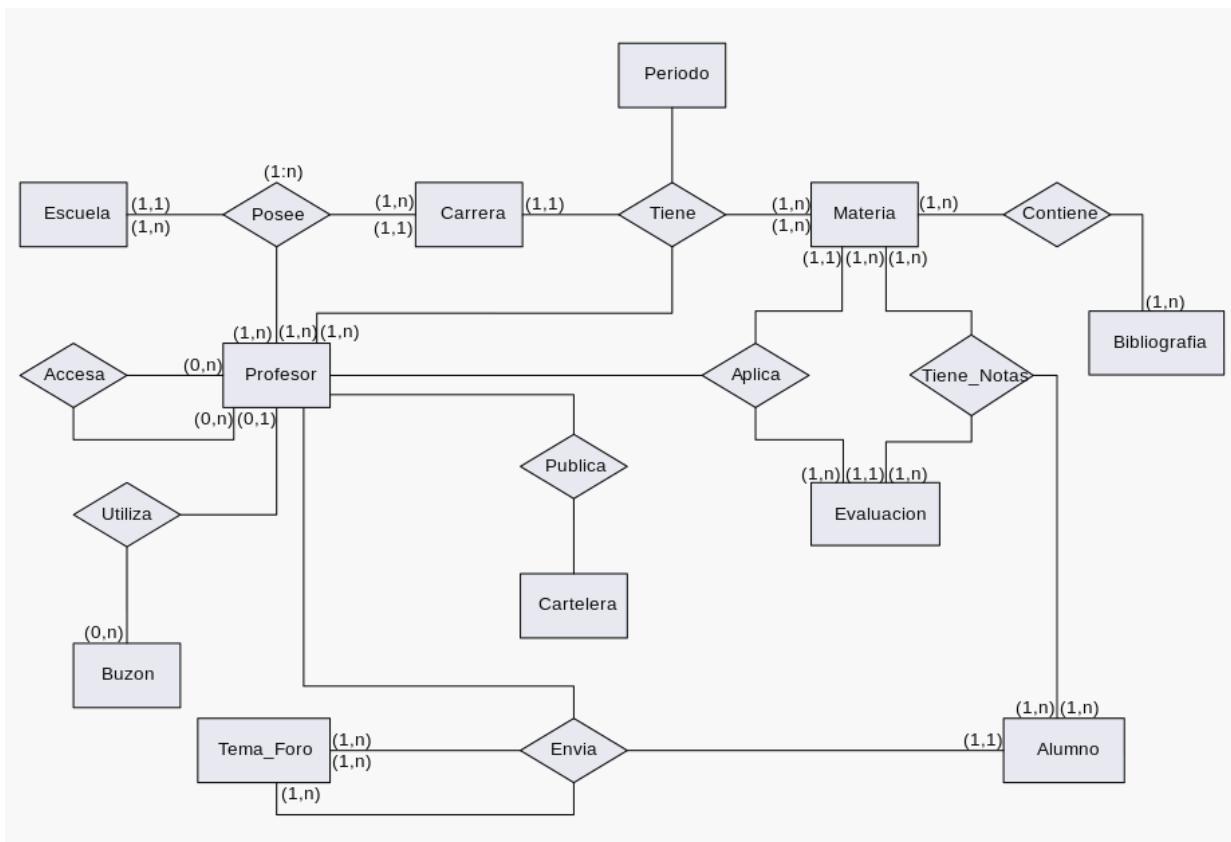


Fig. 5

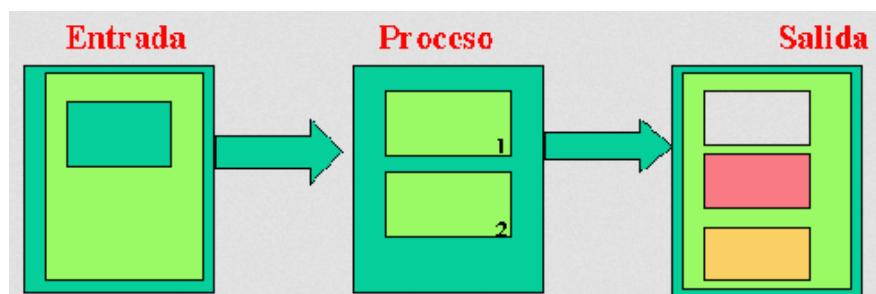
Tarefa 5.3. Escribe un enunciado que se corresponda co diagrama E-R anterior

⁶ From Wikimedia Commons, the free media repository. CC BY-SA 3.0. User:Wilfredor - selft work.

Diagramas HIPO

HIPO (Hierarchy-Input-Process-Output) é un acrónimo de xerarquía de entrada, proceso e saída. En primeiro lugar é unha técnica xerárquica porque o sistema completo de programación se completa con pequenos subsistemas. Estes soportan un enfoque de deseño descendente e tamén reduce a complexidade, xa que cada un dos compoñentes pode consultarse de xeito separado. Como segundo punto o acrónimo recórdanos as tres partes principais dun sistema que son as entradas, os procesos e as saídas.

HIPO é unha técnica visual, o principal beneficio desta é a facilidade de lectura de símbolos estandarizados, utilizados para ilustrar os diferentes tipos de entrada, almacenamento de datos e dispositivos de saídas.



Diagramas VTOC (Visual Tables of Contents)

Son diagramas xerárquicos e soen acompañar e completar ós diagramas HIPO. Proporcionan un mapa que permite facilmente localizar un módulo dentro do sistema. Os números de cada proceso ou módulo seguen un patrón definido de tal forma que é doadoo recoñecer as relacións existentes entre os módulos.

Un diagrama VTOC é similar ós típicos diagramas de estrutura dunha organización tomando a forma dunha pirámide e na parte de debaixo da folla en que se debuxa o diagrama déixase un espazo para unha descripción mais detallada dos cadros.

- Exemplo:

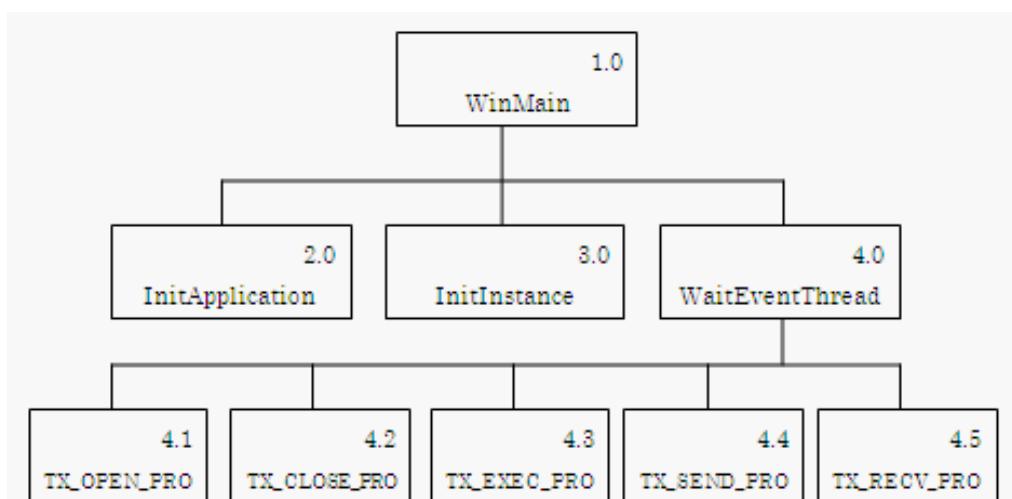


Fig. 6

Diagramas Warnier-Orr

Os diagramas de Warnier/Orr (tamén coñecidos como construcción lóxica de programas/construcción lóxica de sistemas) foron desenvolvidos inicialmente en Francia por Jean Dominique Warnier e nos Estados Unidos por Kenneth Orr. Este método axuda ao deseño de estruturas de programas identificando nun primeiro paso a saída e resultado do procedemento, e a partir del traballa cara atrás para determinar os pasos e combinacións de entrada necesarios para producilos. Os símbolos gráficos usados nos diagramas de Warnier/Orr fan evidentes os niveis nun sistema e más claros os movementos dos datos nos devanditos niveis.

- Exemplo⁷:

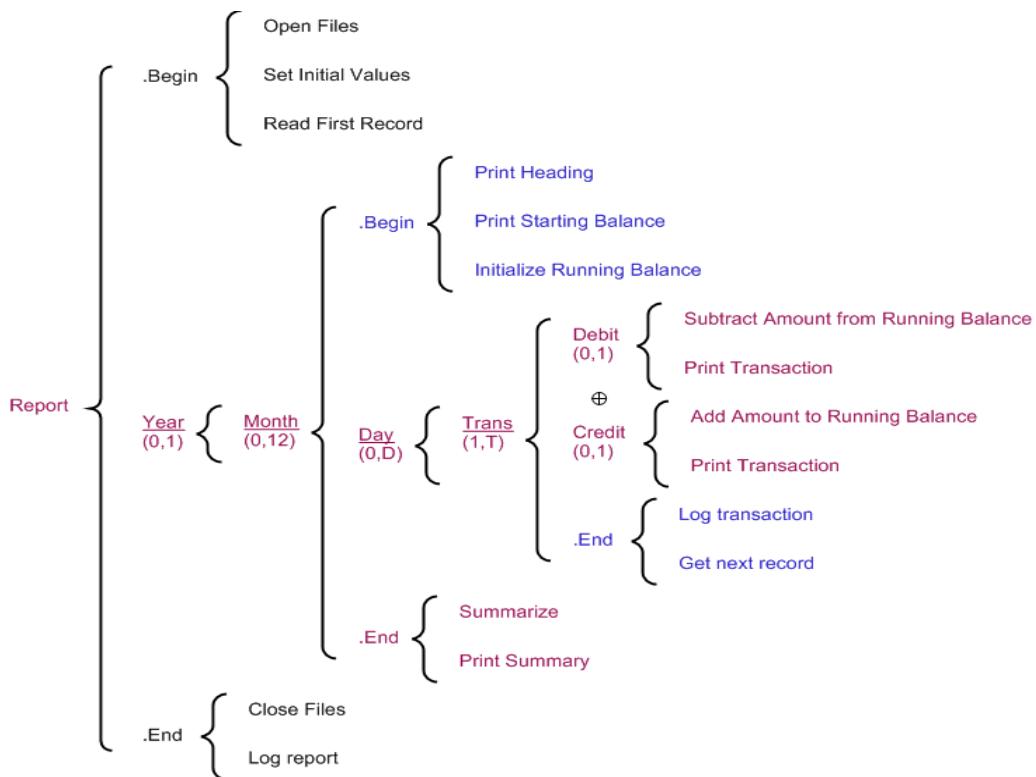


Fig. 7

1.3 Á Linguaxe Unificada de Modelado (UML)

Por que é necesario o UML?

Nos principios da computación, os programadores non realizaban análises moi profundas sobre o problema por resolver. Con frecuencia comezaban a escribir o programa desde o principio, e o código necesario escribíase conforme se requiría.

Conforme aumentou a complexidade do mundo, os sistemas informáticos tamén deberon crecer en complexidade. Nos sistemas actuais atopámonos diversas pezas de hardware e software que se comunican a grandes distancias mediante unha rede que está vinculada a bases de datos que, á súa

⁷ <http://csassignment.weebly.com/software-development-part-2.html>

vez, conteñen enormes cantidades de información. Si desexamos crear sistemas nestes escenarios, como podemos manexar tanta complexidade?

A clave está en organizar o proceso de análise e deseño de tal forma que todas as persoas involucradas no desenvolvemento do sistema compréndano e conveñan con el. Así mesmo será necesario empregar algunha linguaxe ou notación que permita describir os distintos elementos que formarán o sistema, as súas características e interrelación entre eles, e ademais debe de permitir amosar o sistema dende diferentes puntos de vista xa que non e o mesmo a visión que terán do proxecto un responsable de hardware, un programador, un xefe de proxecto, un administrador de bases de datos ou incluso o cliente final. O UML e a linguaxe de modelado que nos dará os elementos necesarios para acadar estes obxectivos.

A concepción do UML

O UML é a creación de *Grady Booch, James Rumbaugh e Ivar Jacobson*. Estes cabaleiros, alcumados "os tres amigos", traballaban en empresas distintas durante a década dos anos oitenta e principios dos noventa e cada un deseñou a súa propia metodoloxía para a análise e deseño orientado a obxectos. As súas metodoloxías predominaron sobre as dos seus competidores. A mediados dos anos noventa empezaron a intercambiar ideas entre si e decidiron desenvolver o seu traballo en conxunto. De este traballo conxunto xurdiría o UML.

Actualmente o estándar UML está promovido polo *Object Management Group ou OMG (Grupo de xestión de obxectos)* que é unha organización sen ánimo de lucro que promove o uso de tecnoloxías orientadas a obxectos mediante guías e especificacións das mesmas.

As versións de UML existentes poden verse con todo detalle en <http://www.omg.org/spec/UML/>:

- *UML 1.4.2*, que é o estándar ISO/IEC 19501 edición 2005
- *UML 2.4.1*, que é o estándar ISO/IEC 19501-1 e ISO/IEC 19501-1 edición 2012 OMG
- *UML 2.1* non foi lanzada por OMG como unha especificación formal pero apareceron as versións 2.1.1, 2.1.2 en 2007.
- *UML 2.4.1*, que é a versión lanzada por OMG en agosto de 2011.
- *UML 2.5* é a versión actual que OMG comezou en xuño de 2015.

Diagramas do UML

UML non é unha metodoloxía, senón que é unha notación para modelar un sistema, polo que non permite describir a documentación de usuario nin a interface gráfica, por exemplo. Así pois, ao empezar un proxecto software, primeiro deberase escoller a metodoloxía baixo a que se vai a traballar e despois utilizar UML ao longo do ciclo de vida que marque a tecnoloxía elixida.

UML esta composto de varios símbolos gráficos combinados seguindo unhas regras para formar *diagramas*. O conxunto dos diagramas formará o *modelo* e cada un deles aportará unha perspectiva diferente do sistema.

Un modelo UML pode definirse como unha abstracción dun sistema ou dun problema que hai que resolver, considerando un certo propósito ou un punto de vista determinado. Indica que é o que vai facer o sistema pero non indica como o vai facer.

O código fonte é a expresión más detallada do modelo pero non é unha ferramenta cómoda de comunicación, sería más cómoda unha gráfica. Un diagrama permitirá representar graficamente un conxunto de elementos do modelo, a veces como un grafo con vértices conectados, e outras veces como secuencias de figuras conectadas que representen un fluxo de traballo.

Un resumo dos diagramas propostos por UML 2.x pode verse na seguinte táboa⁸:

Diagrama	Descripción	Prioridade
Diagrama de Clases	Mostra unha colección de elementos de modelado declarativo (estáticos), tales como clases, tipos e os seus contidos e relacóns.	Alta
Diagrama de Compoñentes	Representa os compoñentes dunha aplicación, sistema ou empresa. Os compoñentes, as súas relacóns, interaccións e as súas interfaces públicas.	Media
Diagrama de Estrutura de Composición	Representa a estrutura interna dun clasificador (tal como unha clase, un compoñente ou un caso de uso), incluíndo os puntos de interacción de clasificador con outras partes do sistema.	Baixa
Diagrama de Despregamento Físico	Un diagrama de despregamento físico mostra como e onde se despregará o sistema. As máquinas físicas e os procesadores represéntanse como nodos e a construcción interna pode ser representada por nodos ou artefactos embebidos.	Media
Diagrama de Obxectos	Un diagrama que presenta os obxectos e as súas relacóns nun punto do tempo. Un diagrama de obxectos pódese considerar como un caso especial dun diagrama de clases ou un diagrama de comunicacóns.	Baixa
Diagrama de Paquetes	Un diagrama que presenta como se organizan os elementos de modelado en paquetes e as dependencias entre eles, incluíndo importacións e extensións de paquetes.	Baixa
Diagrama de Actividades	Representa os procesos de negocios de alto nivel, incluídos o fluxo de datos. Tamén pode utilizarse para modelar lóxica complexa e/ou paralela dentro dun sistema.	Alta
Diagrama de Comunicacións (anteriormente: Diagrama de colaboracións)	É un diagrama que enfoca a interacción entre liñas de vida, onde é central a arquitectura da estrutura interna e como ela se corresponde coa pasaxe de mensaxes. A secuencia das mensaxes dáse a través dun esquema numerado.	Baixa
Diagrama de Revisión da Interacción	Os Diagramas de Revisión da Interacción enfocan a revisión do fluxo de control, onde os nodos son Interaccións ou Ocorrencias de Interaccións.	Baixa
Diagrama de Secuencias	Un diagrama que representa unha interacción, poñendo o foco na secuencia das mensaxes que se intercambian, xunto coas súas correspondentes ocorrencias de eventos nas Liñas de Vida.	Alta
Diagrama de Máquinas de Estado (ou diagrama de estados)	Un diagrama de Máquina de Estados ilustra como un elemento, moitas veces unha clase, pódese mover entre estados que clasifican o seu comportamento, de acordo con disparadores de transicións, gardas de restricións e outros aspectos dos diagramas de Máquinas de Estados, que representan e explican o movemento e o comportamento.	Media
Diagrama de Tempos	O propósito primario do diagrama de tempos é mostrar os cambios no estado ou a condición dunha liña de vida (representando unha Instancia dun Clasificador ou un Rol dun clasificador) ao longo do tempo lineal. O uso más común é mostrar o cambio de estado dun obxecto ao longo do tempo, en resposta aos eventos ou estímulos aceptados.	Baixa
Diagrama de Casos de Uso	Un diagrama que mostra as relacóns entre os actores e o suxeito (sistema), e os casos de uso.	Alta

⁸ https://es.wikipedia.org/wiki/Lenguaje_unificado_de_modelado

Estes diagramas permítenos abrancar as perspectivas más relevantes dun sistema:

- Definición do problema con diagramas de casos de uso.
- Modelo estrutural. Estrutura estática con diagramas de clases, paquetes e obxectos.
- Modelado de comportamento. Vista de procesos con diagramas de comportamento:
 - Diagrama de estados
 - Diagrama de actividade
- Diagramas de interacción ou intercambio de mensaxes entre obxectos dentro dun contexto para conseguir un obxectivo:
 - Diagrama de secuencia no que destaca a ordenación temporal das mensaxes.
 - Diagrama de colaboración ou comunicación no que destaca a secuencia de mensaxes dentro da organización de obxectos.
- Vista de implementación con diagramas de implementación:
 - Diagrama de compoñentes
 - Diagrama de despregue

Non é posible establecer unha secuencia perfecta entre os diagramas, nin unha correspondencia total coas diferentes fases do ciclo de vida, xa que dependendo das persoas involucradas en cada fase e da súa experiencia e coñecemento, algúns diagramas poden quedar completados nas primeiras fases mentres que outros van a estar sometidos a continuas revisións. Tampouco é obligatorio que un modelo inclúa todos os diagramas.

Na seguinte figura⁹ indícanse as correspondencias más evidentes entre diagramas mediante frechas, aínda que case todos eles teñen relación cos demás, de forma que un cambio nalgún pode afectar aos outros. Por exemplo, un cambio nun caso de uso leva consigo cambios noutros diagramas ou un cambio no diagrama de compoñentes pode afectar aos diagramas de clases e de despregue.

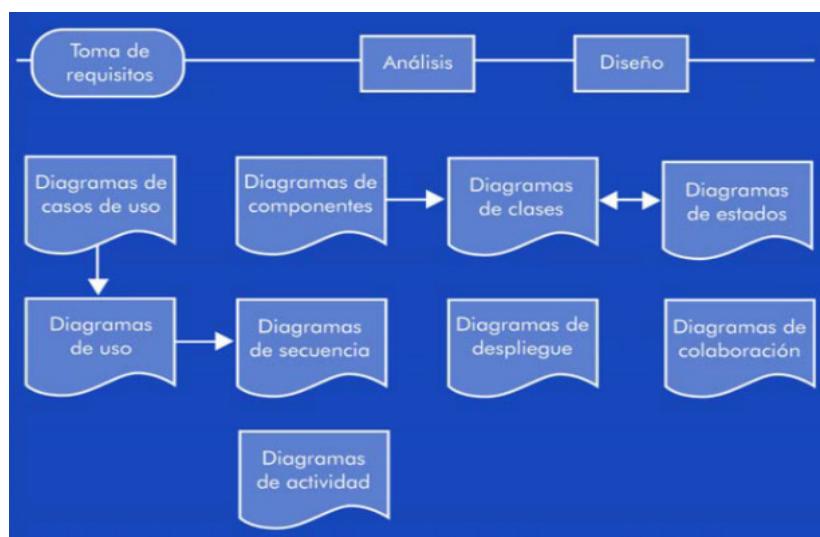


Fig. 8

⁹ AYCART PÉREZ, David. GIBERT GINESTA, Marc HERNÁNDEZ MATÍAS, Martín, MAS HERNÁNDEZ, Jordi. *Ingeniería de software en entornos de SL*. Universitat Ouberta de Catalunya.

A continuación veremos algunos exemplos¹⁰ dos principais diagramas UML:

- **Diagrama de casos de uso.** O diagrama de casos de uso permite representar as interaccións entre o sistema e os seus actores en resposta a un evento que inicia un actor.

Un caso de uso proporciona un ou más escenarios de actuación do sistema dende o punto de vista do cliente.

Nos diagramas de casos de uso evítase a utilización de xerga técnica para que o cliente poida interpretalo facilmente. Por exemplo:



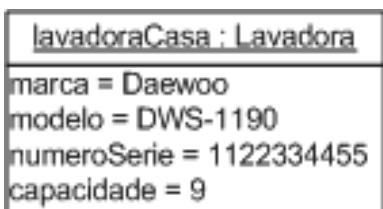
- **Diagrama de clases.** O diagrama de clases representa a estrutura lóxica do sistema, é dicir, as clases que o forman e as súas relacóns. Por cada clase indícase o seu nome, atributos e métodos.

Exemplos de posibles diagramas da clase Lavadora:

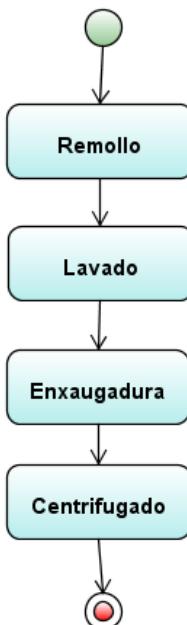
Lavadora	Lavadora	Lavadora
<p>-marca -modelo -numeroSerie -capacidade</p> <p>+agregarRoupa() +agregarDeterxente() +activarLavado() +sacarRoupa()</p>	<p><i>Attributes</i></p> <p>private String marca private String modelo private int numeroserie private int capacidade</p> <p><i>Operations</i></p> <p>public void agregarRoupa() public void agregarDeterxente() public void activarLavado() public void sacarRoupa()</p>	<p><i>Attributes</i></p> <p>private String marca private String modelo private int numeroserie private int capacidade</p> <p><i>Operations</i></p> <p>public Lavadora() public String getMarca() public void setMarca(String val) public String getModelo() public void setModelo(String val) public int getNumeroserie() public void setNumeroserie(int val) public int getCapacidade() public void setCapacidade(int val) public void agregarRoupa() public void agregarDeterxente() public void activarLavado() public void sacarRoupa()</p>

¹⁰ Os exemplos da Lavadora foron extraídos do libro "*Aprendiendo UML en 24 Horas*" de Joseph Schmuller.

Un caso especial de diagrama de clases é o diagrama de obxectos que permite representar unha instancia dunha clase. Por exemplo o diagrama do obxecto lavadoraCasa da clase Lavadora:



- **Diagrama de estados.** É un diagrama que mostra as transicións entre diferentes estados dun obxecto. A transición entre estados é instantánea e correspón dese coa ocorrencia dun evento.



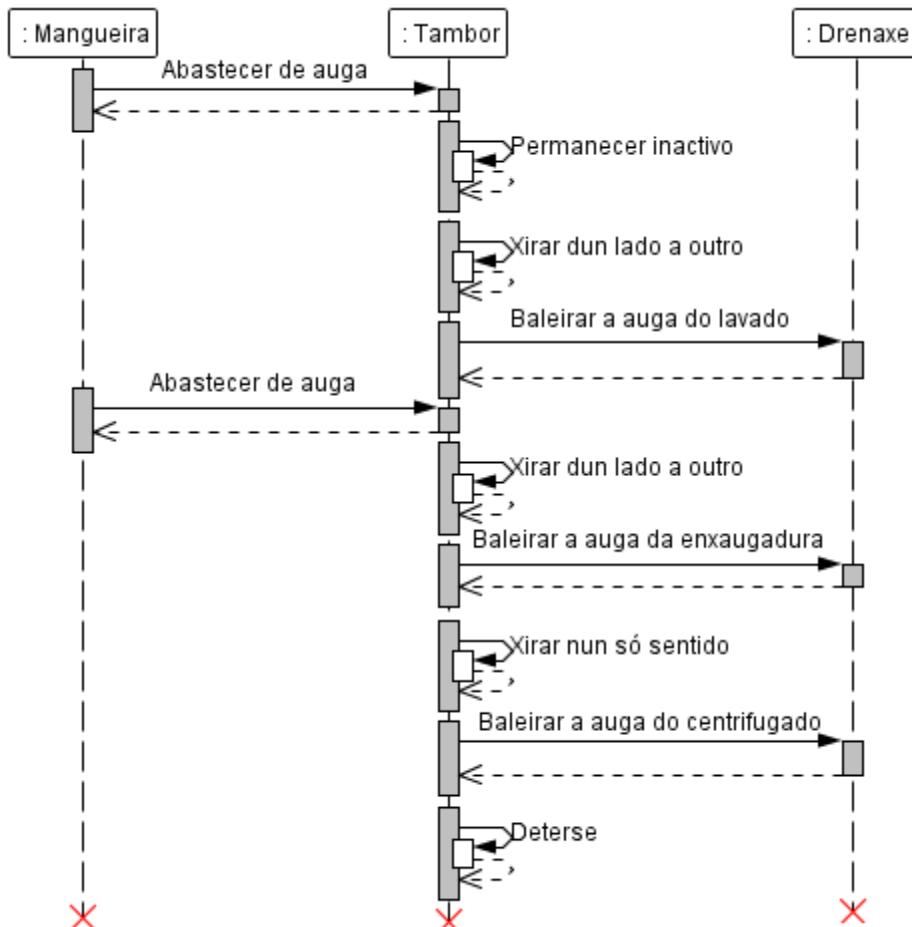
Por exemplo o diagrama de estados dun obxecto da clase Lavadora no proceso de activar lavado podería ser:

- **Diagrama de secuencias.** Representa a interacción entre obxectos ao longo do tempo. Por exemplo, o proceso de lavado de roupa despois de haber completado as operacións "agregar roupa", "agregar deterxente" e "activar" podería estar formado polas actividades representadas textualmente como:

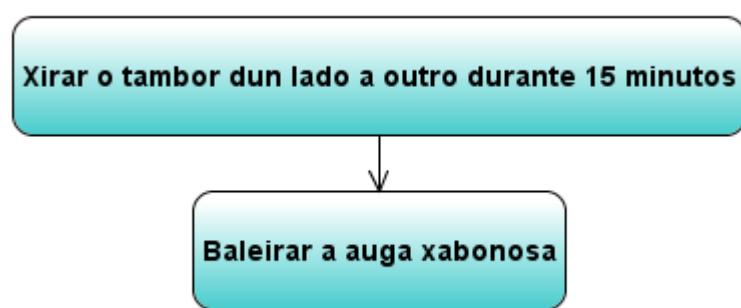
- A auga empeza a encher o Tambor mediante unha mangueira.
- A mangueira deixa de abastecer auga.
- O tambor permanece inactivo.
- O tambor xira dun lado a outro.
- A auga con xabón sae polo drenaxe.
- Comeza un novo abastecemento de auga.
- O abastecemento de auga detense.
- O tambor continúa xirando dun lado a outro.
- A auga da enxaugadura sae polo drenaxe.
- O tambor xira nunha soa dirección e incrementase a velocidade.
- A auga sobrante sae polo drenaxe.
- O tambor deixa de xirar e o proceso de lavado finaliza.

Enlazando o anterior co diagrama de estados: os pasos 1-3 corresponderían ao estado de Remollo; os pasos 4-5 ao estado de Lavado, os pasos 6-9 ao estado de enxaugadura e os pasos 10-12 ao estado de centrifugado.

E en forma de diagrama de secuencias suporemos a existencia de 3 obxectos das clases Mangueira, Tambor e Drenaxe:



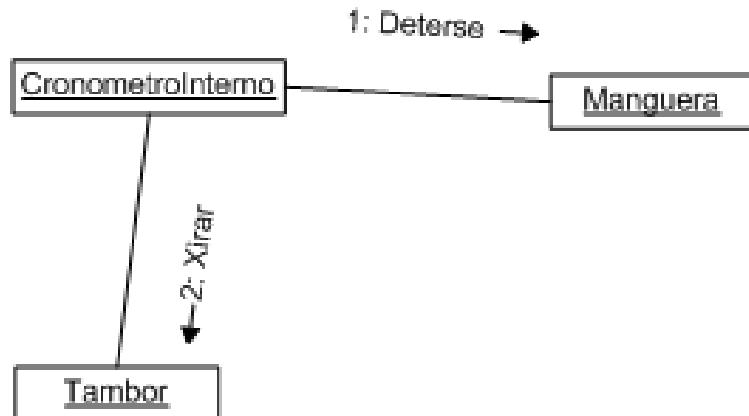
- **Diagrama de actividades.** O diagrama de actividades representa o fluxo de traballo paso a paso dentro dun caso de uso ou dentro do comportamento dun obxecto. Normalmente este fluxo describese cun diagrama de secuencia.



A representación das actividades para o proceso de lavado anterior podería ser:

- **Diagrama de colaboracións.** Representa o traballo en conxunto dos elementos dun sistema para cumplir cos obxectivos do mesmo.

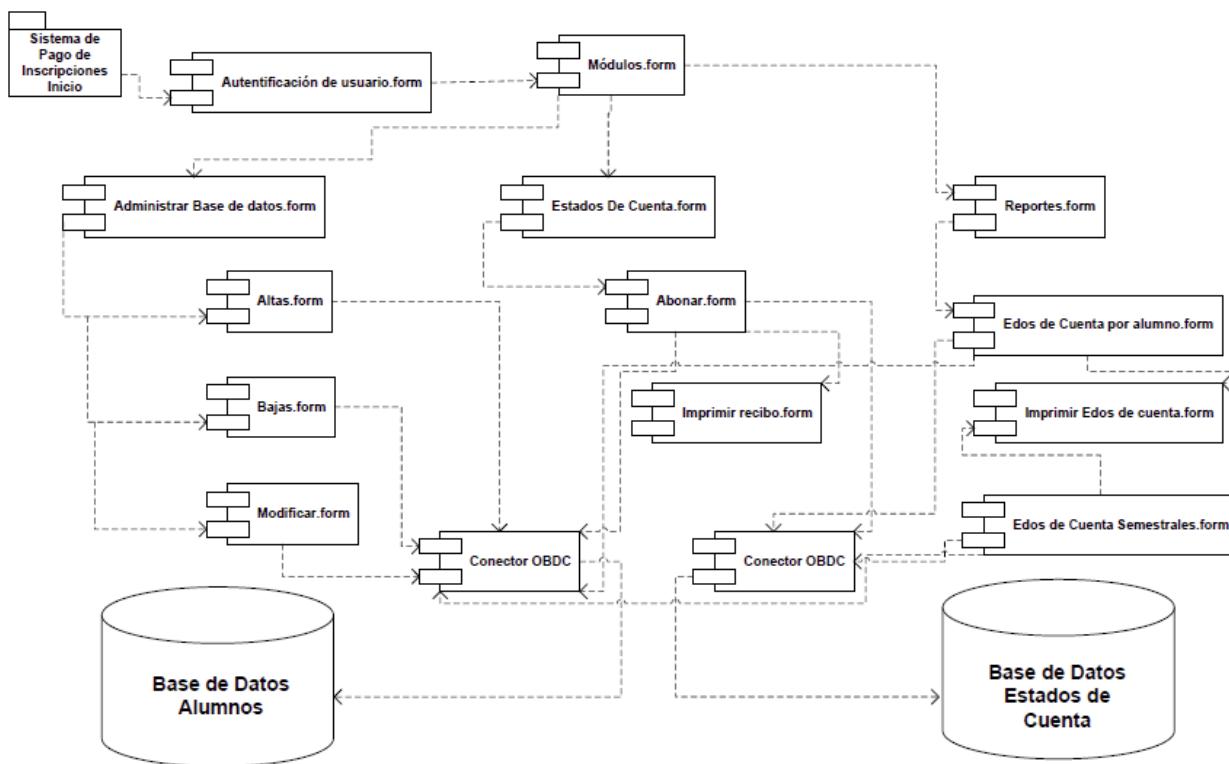
Por exemplo a existencia dunha nova clase para o cronómetro interno que actúa entre a mangueira de auga e o tambor para controlar o tempo. O diagrama de colaboracións podería ser:



- **Diagrama de componentes.** O diagrama de componentes representa a subdivisión dun sistema en componentes e mostra as dependencias entre eles. Estes componentes poden ser arquivos, bases de datos programas, bibliotecas, módulos executables ou paquetes.

A relación entre componentes represéntase cunha liña descontinua acabada en punta de frecha no destino indicando que o componente orixe depende do destino.

Exemplo¹¹:



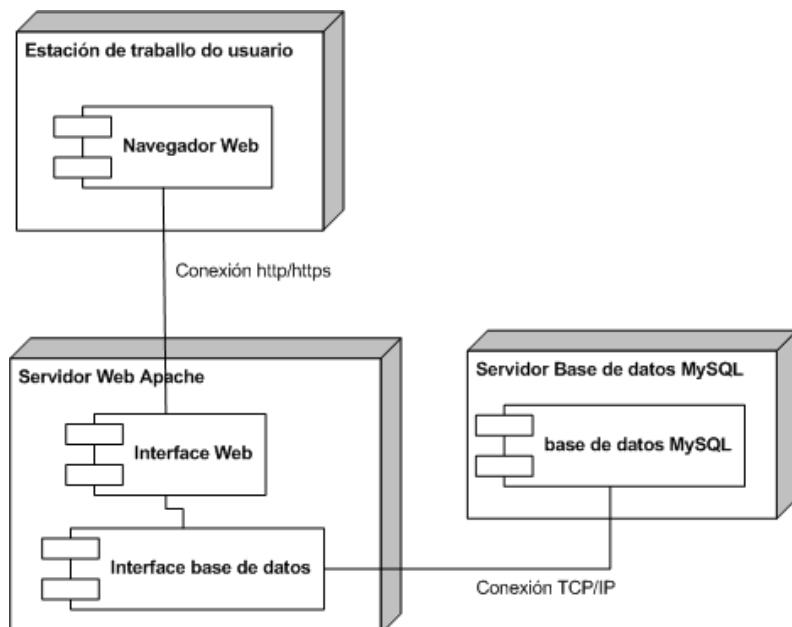
¹¹ <http://gzloluna8sm.blogspot.com.es/2010/06/diagrama-de-componentes.html>

- **Diagrama de despregue.** O diagrama de despregue mostra a disposición física dos distintos nodos que componen o sistema e o reparto de componentes sobre esos nodos.

Enlazando o diagrama de despregue co de componentes:

- Os componentes son os elementos que participan na execución do sistema.
- Os nodos son os elementos onde se executan os componentes e pode ser unha linguaxe de programación, un sistema operativo ou un ordenador.

O nodo representase mediante un cubo co nome do nodo dentro e os nodos ou os componentes únense mediante liñas para indicar conexión entre eles. Pódense utilizar estereotipos para precisar o tipo de conexión.

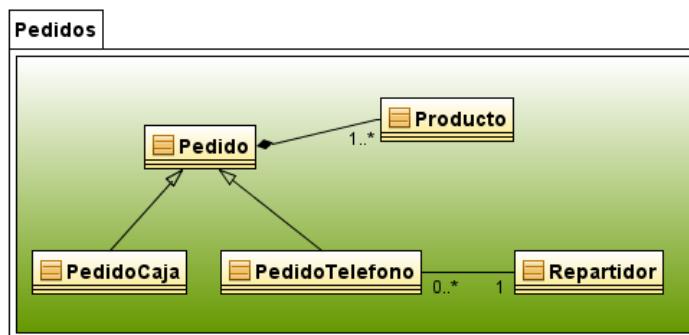


- **Paquetes.** Un paquete mostra agrupacións lóxicas de elementos de modelado. Determina un espacio de nomes e por tanto os nomes dos elementos dun paquete non poden repetirse.

Graficamente representase como un rectángulo con pestana na que se coloca o nome do paquete. Dentro do rectángulo colócanse os elementos que forman o paquete.

Un paquete pode ter como contido a outro paquete.

Por exemplo un paquete formado por varias clases podería ser:

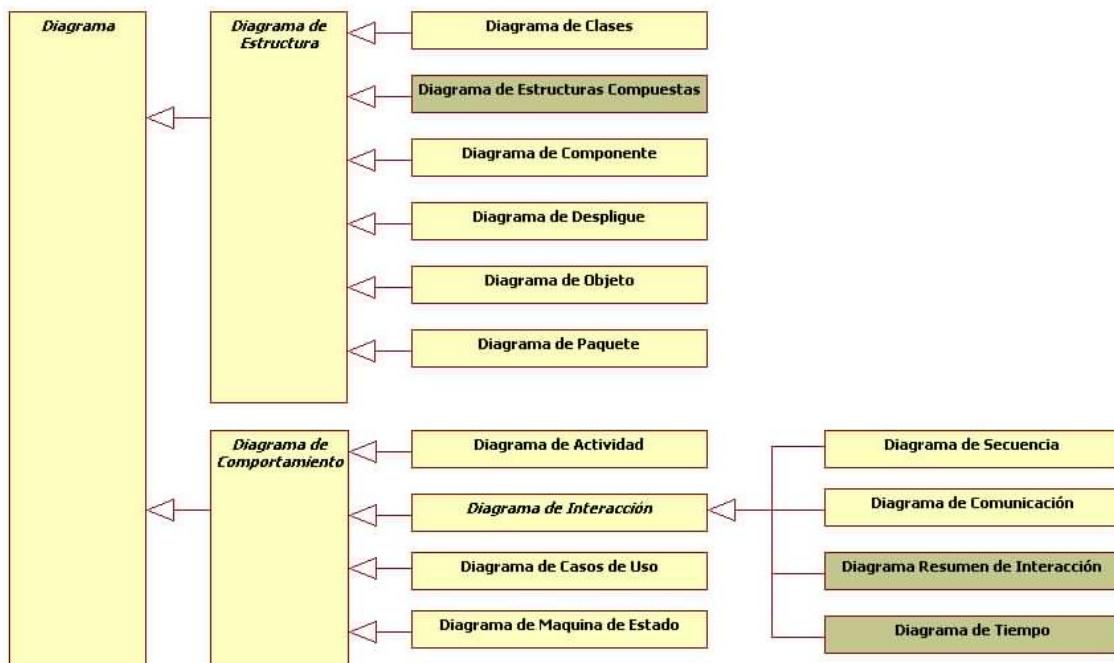


2. Diagramas de clases.

2.1 Introdución ós diagramas de clases

Os diagramas estruturais de UML

Os diagramas de clases son sen lugar a dúbidas os diagramas máis utilizados no modelado de sistemas orientados a obxectos e pertencen o grupo dos chamados *diagramas estruturais* de UML. O conxunto de diagramas de UML subdivídese habitualmente, tal e como pode verse na seguinte figura¹², en dous grandes subconxuntos: *diagramas estruturais (ou de estrutura)* e *diagramas de comportamento*.



Os diagramas estruturais *presentan elementos estáticos do modelo*, tales como clases, paquetes ou compoñentes; en tanto que os diagramas de comportamento *mostran a conduta en tempo de execución do sistema*, tanto visto como un todo como das instancias ou obxectos que o integran.

Os diagramas sinalados nunha cor distinta foron incorporados na versión 2 de UML polo que resultan ser dos menos coñecidos e empregados.

Propósito e función dos diagramas de clases

Os diagramas de clases permítennos representar unha vista dos compoñentes estáticos dun sistema, xa sexan estes clases ou módulos, indicando as relacións entre estas e os atributos (datos) das clases, así como os seus métodos (código).

¹² <http://synergix.wordpress.com/2008/07/20/tipos-de-diagramas-en-uml/>

Os diagramas de clases poden ser utilizados para presentar a vista estática do *modelo de dominio* - modelo da fase de análise que serve para a compresión do entorno ó cal o sistema ten que servir ou emular -, do *modelo de deseño* - encárgase de refinar a arquitectura definida na fase de análise adaptándoa ao ambiente de implementación-, ou ben, do *detailed da implementación* dun sistema nunha linguaxe de programación orientada a obxecto, como Eiffel, Java ou C++. Para todos estes usos, o que se desea é expresar as unidades en que o código se organiza -as clases- así como algunas características destas, como son as súas relacións, atributos e métodos.

Aínda que a especificación de UML fala de diagramas diferentes para os paquetes e as clases, é valido combinalos dando lugar a un diagrama que mostra simultaneamente clases e paquetes. Isto axuda a documentar elementos que estando en distintos paquetes gardan algunha relación entre eles.

É importante mencionar tamén que os autores de UML consideran os diagramas de clases como un superconxunto dos coñecidos diagramas de entidade/relación empregados para deseñar bases de datos relacionais.

2.2 Elementos dos diagramas de clases

Ver vídeo: <https://www.youtube.com/watch?v=Z0yLerU0g-Q>



Clase, atributo, operación

Unha *clase* é a descripción dos atributos e operacións que comparten un conxunto de obxectos. En UML represéntanse graficamente cun rectángulo de tres filas.

O *nome* da clase escríbese normalmente coa primeira letra de cada palabra en maiúsculas e está formado por un substantivo singular seguido dun ou varios adjetivos que o cualifican sen espazos no medio. Graficamente colócase na primeira fila do rectángulo. Se se quere facer referencia ao paquete ao que pertence a clase, farase nomeando a clase como nomepaquete::nomoclase.

O *atributo* é unha propiedade da clase. Unha clase pode ter ou non atributos. O atributo ten un nome corto que normalmente empeza por minúsculas e ten o resto das primeiras letras de cada palabra en maiúsculas sen espazos no medio. Por cada atributo pódese indicar o modificador de visibilidade (+=public, #=protected, -=private), tipo de dato (int, String, ...), nome e valor por defecto. Graficamente colócanse na segunda fila do rectángulo. Os atributos ou métodos de clase (*static*) aparecerán subliñados.

Unha *operación* ou *método* é a implementación dun servizo que pode ser requerido por calquera obxecto da clase para que se execute. Unha clase normalmente ten operacións. O nome do método normalmente é un verbo ou unha expresión verbal que empeza por minúsculas e ten o resto das primeiras letras de cada palabra en maiúsculas sen espazos no medio. O método pode ter ou non parámetros e pode ter ou non valor de retorno. Graficamente colócase a firma completa (modificador da visibilidade, tipo do retorno, nome do método, e para cada parámetro: tipo, nome e valor por defecto) na terceira fila do rectángulo.

Opcionalmente pode engadirse unha fila máis ao rectángulo para poñer a responsabilidade da clase ou texto libre e curto no que se expresan as obligacións da clase.

Un diagrama de clases normalmente está formado por más dunha clase relacionadas; neste caso pode ser máis claro que non aparezan tan detallados os atributos e os métodos.

Cliente	PorciónCarneCaballar
<p>Attributes</p> <pre>private String nome private String direccion private float tasa = 5.3</pre> <p>Operations</p> <pre>public Cliente() public String getNome() public void setNome(String val) public String getDireccion() public void setDireccion(String val) public float getTasa() public void setTasa(float val)</pre>	<p>Attributes</p> <pre>private int peso private int calidad protected float precioSinIVA protected int tasaIVA = 10</pre> <p>Operations</p> <pre>public PorciónCarneCaballar() public int getPeso() public void setPeso(int val) public int getCalidad() public void setCalidad(int val) public float getPrecioSinIVA() public void setPrecioSinIVA(float val) public int getTasaIVA() public void setTasaIVA(int val) public float precioConIVA()</pre>

Recomendacións para a elaboración dos diagramas

Nas conversacións cos clientes débese prestar atención aos substantivos que utilizan para describir o seu negocio porque normalmente deles sairán as clases; dos substantivos relacionados coas clases sairán os atributos; dos verbos sairán os métodos.

Por exemplo supoñamos a seguinte conversación entre un adestrador de baloncesto e un analista que necesita saber como funciona o xogo:

- *Analista:* "Adestrador, de que trata o xogo? "
- *Adestrador:* "Consiste en botar o balón a través dun aro, coñecido como cesto, e facer unha maior puntuación que o opoñente. Cada equipo consta de cinco xogadores: dous defensas, dous dianteiros e un central. Cada equipo leva o balón ao cesto do equipo opoñente co obxectivo de facer que o balón sexa encestado. "
- *Analista:* "Como se fai para levar o balón ao outro cesto? "
- *Adestrador:* "Mediante pases e dribles. Pero o equipo terá que encestar antes de que remate o lapso para tirar. "
- *Analista:* "O lapso para tirar? "
- *Adestrador:* "Son 24 segundos no baloncesto profesional, 30 nun xogo internacional, e 35 no colecial para tirar o balón logo do cal un equipo toma posesión del, "
- *Analista:* "Como se puntúa? "
- *Adestrador:* "Cada canasta vale dous puntos, a menos que o tiro fose feito detrás da liña dos tres puntos. En tal caso, serán tres puntos. Un tiro libre contará como un punto. A propósito, un tiro libre é a penalización que paga un equipo por cometer unha infracción. Se un xogador realiza unha infracción sobre un opoñente, se detén o xogo e o opoñente pode realizar diversos tiros ao cesto dende a liña de tiro libre"
- *Analista:* "Fáleme máis acerca do que fai cada xogador. "
- *Adestrador:* "Os que xogan de defensa son, en xeral, os que realizan a maior parte de dribles e pases. Polo xeral teñen menor estatura que os dianteiros, e estes, á súa vez, son menos altos que o central (que tamén se coñece como 'poste'). Supонse que todos os xogadores poden burlar, pasar, tirar e rebotar. Os dianteiros realizan a maioría dos rebotes e os disparos de mediano alcance, mentres que o central se mantén preto do cesto e dispara dende un alcance curto. "

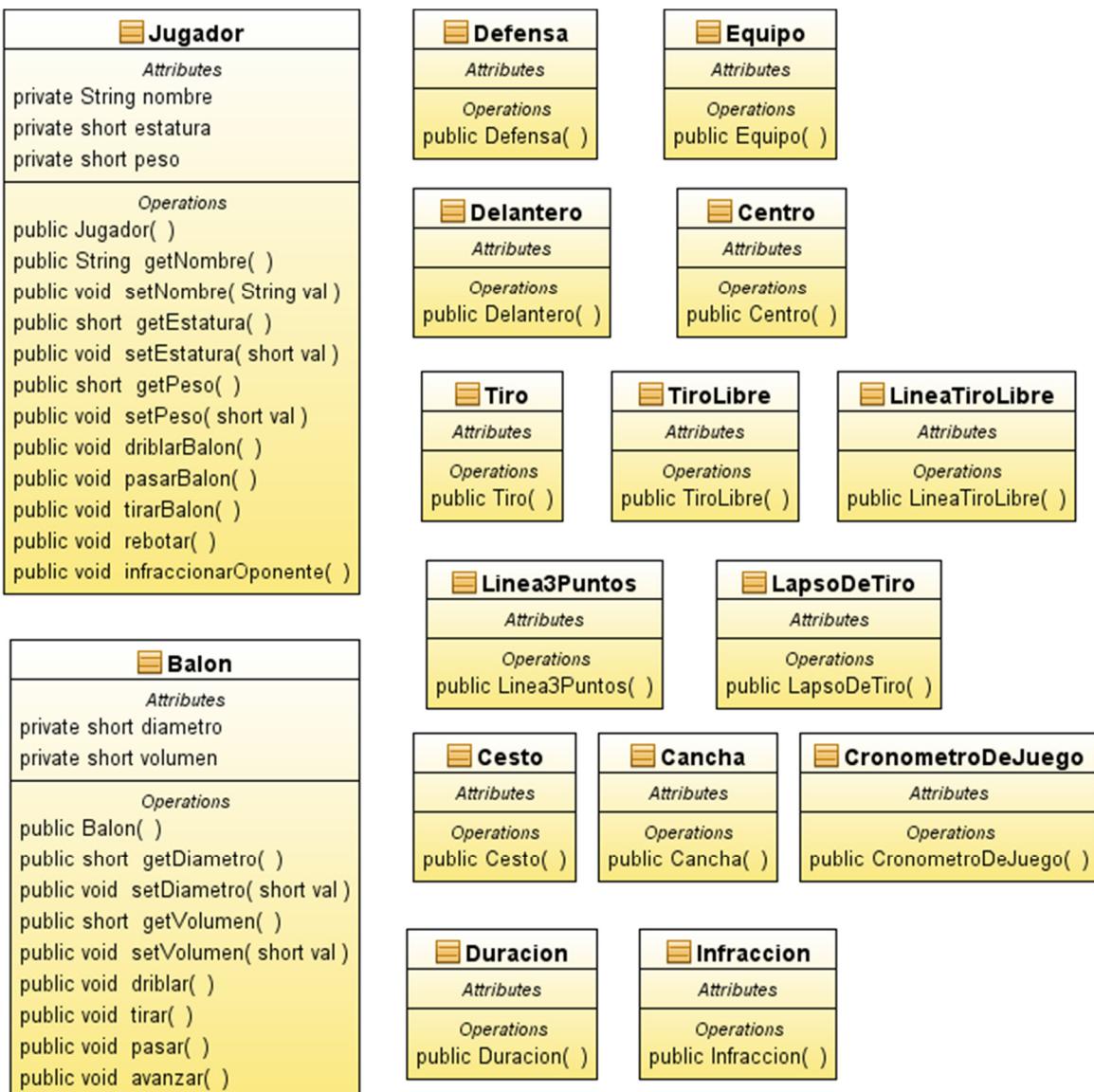
- *Analista:* " Cales son as dimensíons da cancha? E xa que estamos niso canto dura o xogo? "
- *Adestrador:* 'Nun xogo internacional, a cancha mide 28 metros de lonxitude e 15 de ancho; o cesto encóntrase a 3.05 m do piso. Nun xogo profesional, o xogo dura 48 minutos, divididos en catro cuartos de 12 minutos cada un. Nun xogo internacional, a duración é de 40 minutos, divididos en dúas metades de 20 minutos. Un cronómetro do xogo leva o control do tempo restante. ".

Substantivos que apareceron: balón, cesto, equipo, xogadores, defensas, dianteiros, centro, tiro, lapso para tirar, liña de tres puntos, tiro libre, infracción, liña de tiro libre, cancha, cronómetro do xogo.

Verbos que apareceron: tirar, avanzar, driblar (o burlar), pasar, facer unha infracción, rebotar.

Información adicional respecto a algúns substantivos: estaturas relativas dos xogadores de cada posición, dimensíons da cancha, cantidade total de tempo nun lapso de tiro e duración dun xogo.

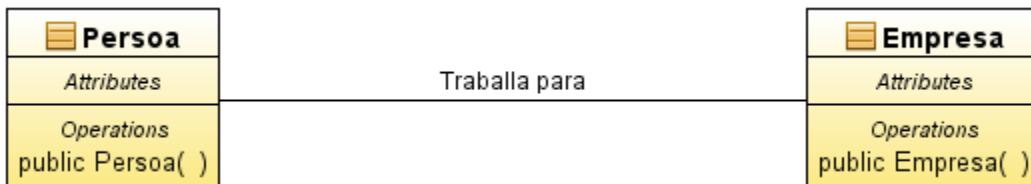
Unhas clases posibles poderían ser:



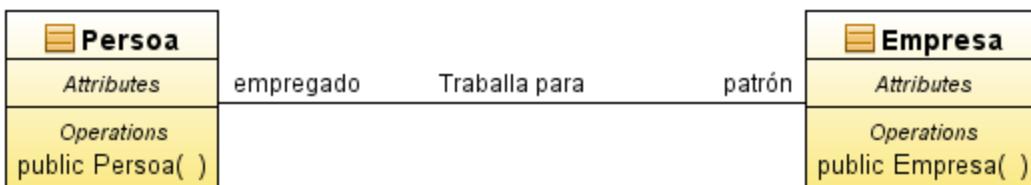
Asociación

Unha *asociación* é unha relación entre clases que especifica que os obxectos dunha clase están conectados cos doutra. Graficamente represéntase como una liña que une as clases e soe ter nome que se colocariba da liña que une as dúas clases.

As asociación son normalmente binarias (entre dúas clases). Cando se establece unha asociación, pódese navegar dende un obxecto dunha clase ata un obxecto da outra e viceversa. Si a liña remata cunha punta de frecha nun dos extremos dise que a asociación ten *navegabilidade*, isto quere dicir que é posible ir dun obxecto a outro no sentido que indica a frecha pero non ó contrario.



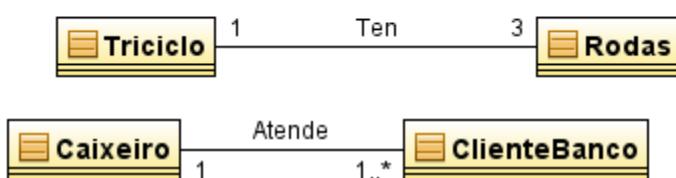
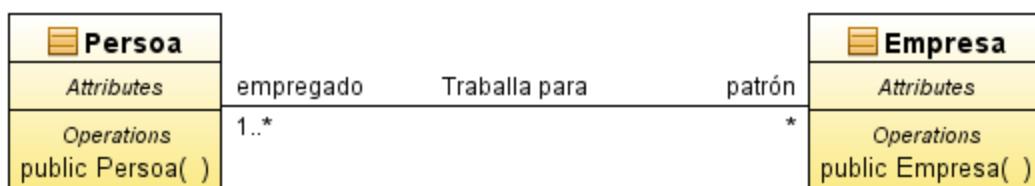
Un *rol* é a cara que unha clase presenta á outra clase da asociación. Pode nomearse explicitamenteriba da liña da asociación e a carón da clase correspondente.



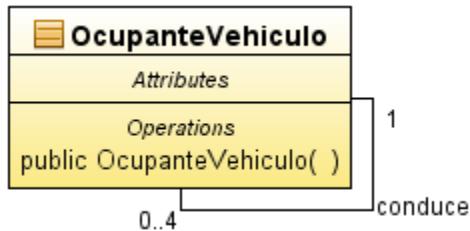
A *multiplicidade* ou *cardinalidade* da asociación é o número de obxectos dunha clase que poden relacionarse cun obxecto da outra clase e colócase na liña de asociación e a carón da clase correspondente. A multiplicidade pode ser: 1, cero ou un: 0..1, cero ou máis: = 0..* (tamén se pode poñer só *), un ou máis: 1..*, un número exacto, ou expresións más complexas como: 0..1,3..4,6..* (calquera número agás o 2 e o 5).

Cando se pon o símbolo da multiplicidade nun extremo da asociación indica que para cada obxecto da clase do extremo oposto pode haber os obxectos desa clase que se indican na multiplicidade.

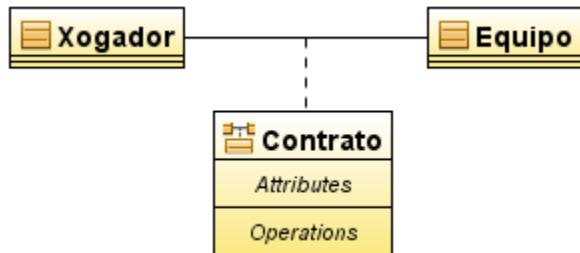
Exemplos:



A asociación reflexiva dáse cando os extremos dunha asociación están conectados á mesma clase.



Unha asociación pode ser tamén unha clase e por tanto conter atributos e operacións. Neste caso utilízase unha liña descontinua para enlazar a clase de asociación coa propia asociación.



Tamén existe a posibilidade de incluír *restriccións*, é dicir, que a asociación teña que seguir algúna regra. Esta restrición pode expressarse mediante una expresión entre chaves, unha decisión {Or} ou mediante unha linguaxe denominada OCL (Linguaxe de restrición de obxectos). Ver máis sobre OCL en <https://www.omg.org/spec/OCL/2.0/About-OCL/>

A asociación pode especializarse e entón transformarse en dependencia, xeneralización, composición forte (composición) ou composición débil (agregación).

Xeneralización

A *xeneralización* é unha especialización da asociación na que intervén a herdanza.

Terminoloxía relacionada coa herdanza entre clases:

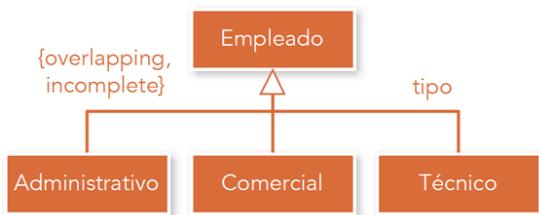
- Unha clase sen pai e con clases filla denomínase clase raíz ou clase base.
- Unha clase sen fillas chámase clase folla.
- Unha clase cunha única clase pai dise que ten heranza simple.
- Unha clase con máis dun pai dise que ten heranza múltiple.

Unha *xeneralización* é unha relación entre unha clase xeneral chamada súper clase ou pai e unha clase más específica chamada subclase ou filla. A clase filla herda os atributos e operacións da clase pai, pode engadir atributos e operacións aos que herda e pode redefinir operacións herdadas (polimorfismo).

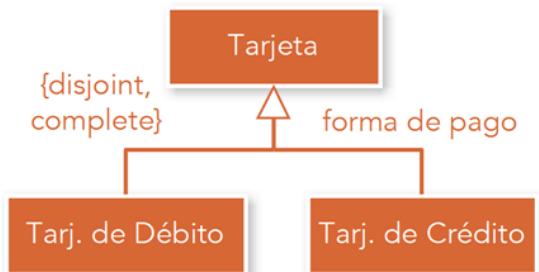
Unha xeneralización represéntase en UML como unha liña continua coa punta da frecha sen recheo apuntando ao pai.

Na xeneralización pode especificarse máis a relación entre a súper clase e as subclases indicando mediante estereotipos estándar:

- Cobertura: Indica se é obligatorio que tódalas instancias dunha clase pai se clasifiquen nalgúnha das clases fillas:
- Complete: Cada instancia da superclase é obligatoriamente tamén instancia dalgúnha ou varias subclases.
- Incomplete: Pode haber instancias da superclase que non sexan instancias de ningunha subclase.
- Solapamento: Se unha instancia pode pertencer á vez a diferentes subclases:
- Disjoint: Unha instancia da clase pai só pode ser dunha única clase filla
- Overlapping. Permítese que unha instancia pertenza a varias subclases.

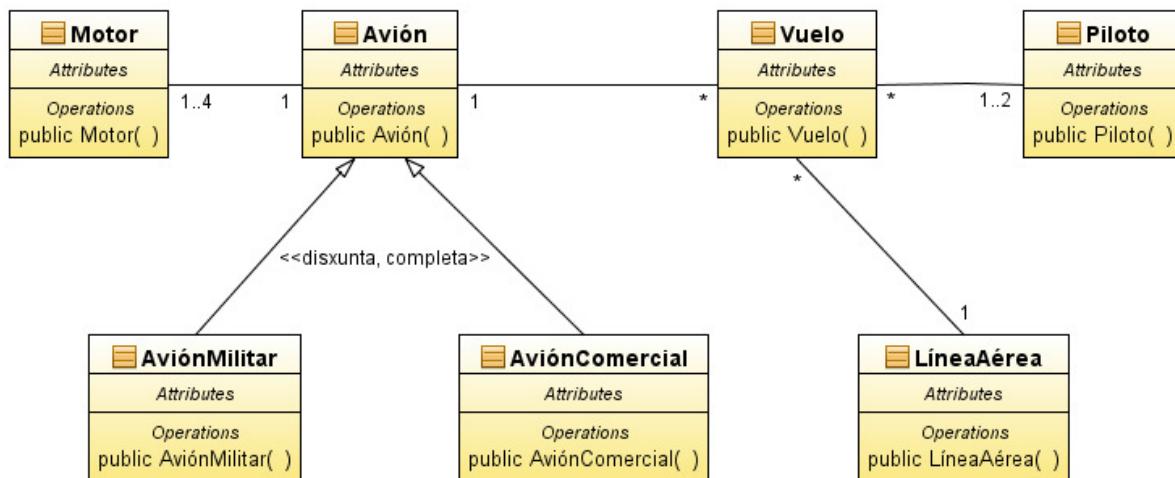


Neste caso un empregado podería ser Técnico e Administrativo á vez. Tamén podería non ser de ningunha categoría e pertencer directamente á clase Empregado

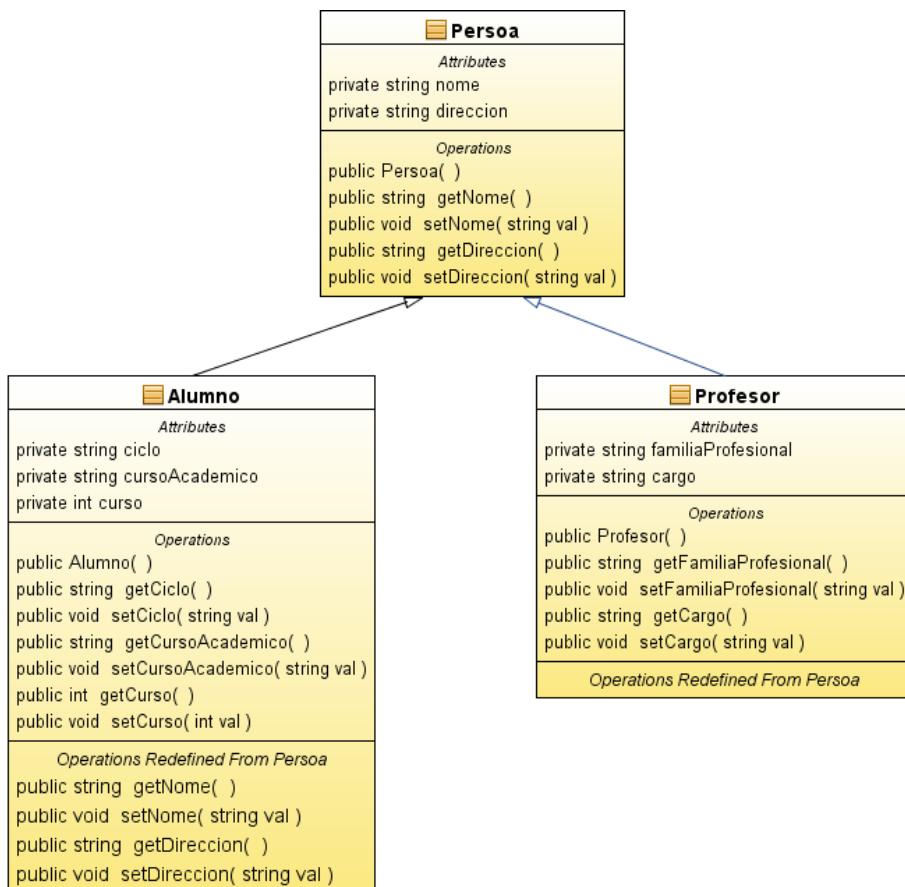


Neste outro caso as tarxetas só poden ser de dous tipos: crédito ou débito. Unha tarxeta de débito non pode ser nunca de crédito e viceversa

Se algunha subclase redefine métodos da clase pai, pode indicarse na subclase cales son os métodos que se redefinen.

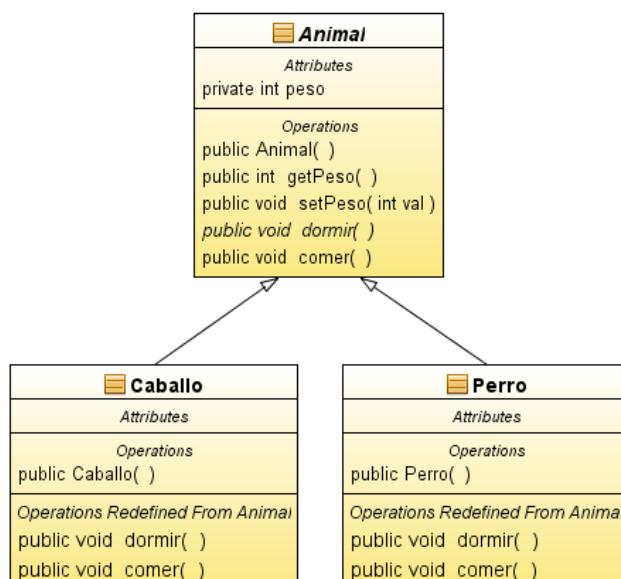


Clases abstractas



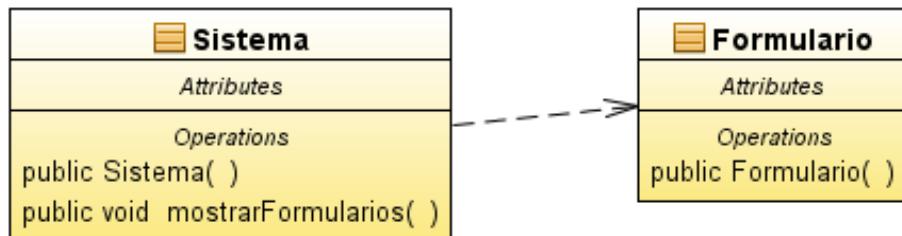
Unha clase *abstracta* é aquela que non ten implementados todos os métodos e por tanto non pode ser instanciada. Pode ter métodos abstractos (só a firma e non o contido) ou non. A súa finalidade é posuír subclases concretas que poden ser instanciadas e nas que se definirán os métodos abstractos da súper clase. Utilízanse cando nunha xerarquía de clases algún comportamento está presente en todas elas pero se materializa de forma distinta para cada unha.

As clases e métodos abstractos represéntanse co nome en cursiva e está dentro dunha clase pai dunha xeneralización. Podemos poñerlle `<>` diante, para que quede más claro.



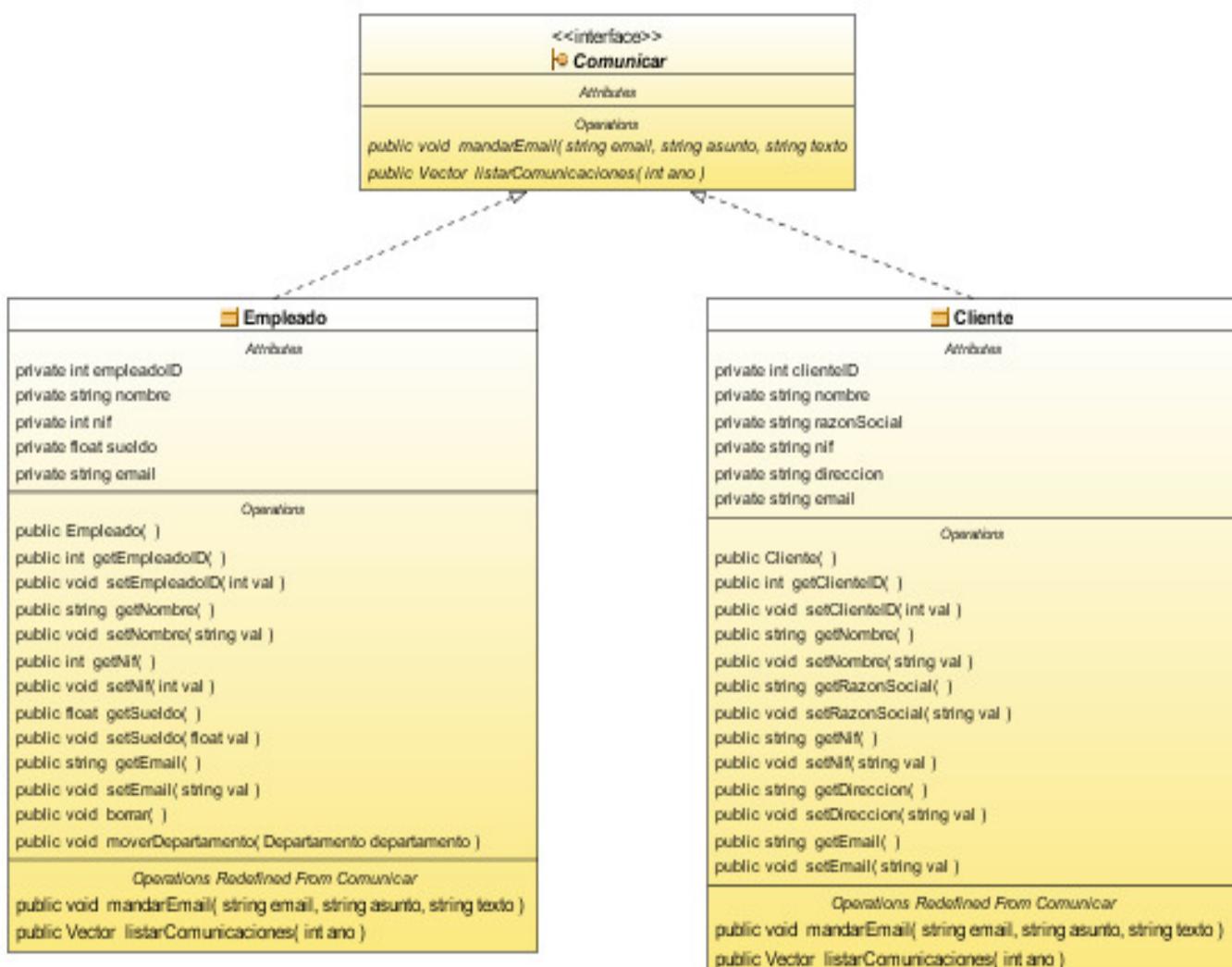
Dependencia

Unha *dependencia* é unha asociación na que se indica que unha clase necesita doutra para o seu cometido. Represéntase cunha frecha descontinua que vai dende a clase dependente á clase utilizada. As dependencias indican que un cambio na clase utilizada pode afectar ao funcionamento da clase dependente pero non ao contrario.



Interface

Unha **interface** pode ser definida como unha clase abstracta pura, é dicir, declara a forma dunha clase e por tanto só define métodos abstractos e atributos constantes que logo serán implementados en clases. A interface represéntase cunha icona específica e o estereotipo `<<interface>>` riba do nome e as clases que implementan esa interface serán dependentes da interface.



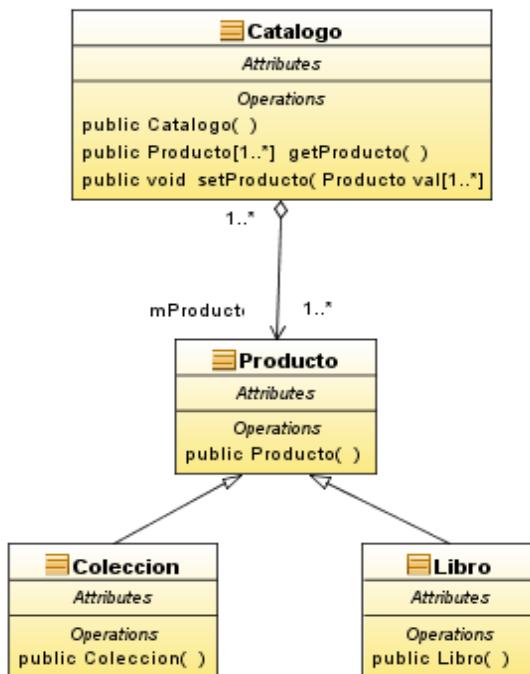
Composición débil ou Agregación

É unha asociación entre clases que indica que unha clase A está composta pola clase B de tal forma que os componentes poden ser compartidos por outros elementos compostos. Como consecuencia, a supresión do obxecto da clase A non implica a supresión do obxecto agregado da clase B.

Represéntase graficamente cunha liña continua cun rombo sen recheo na clase A.

A agregación pode levar navegabilidade para indicar que a propiedade agregada se inclúe automaticamente na clase A; en caso contrario, habería que definir explicitamente esa propiedade.

Exemplo: unha librería na que os produtos se ofrecen agrupados en catálogos. Cada catálogo está formado por un ou máis produtos. Os produtos poden ser libros ou coleccións. Agrégase a clase Produto á clase Catalogo e se a agregación é con navegabilidade agregaranse os métodos get e set na clase Catalogo.



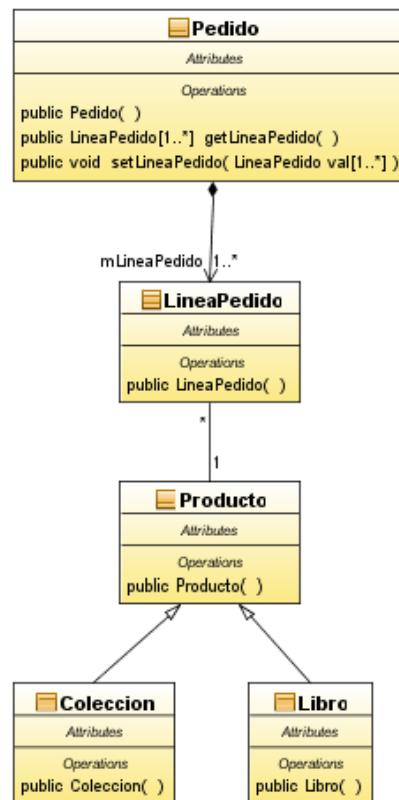
Composición

É unha asociación entre clases que especifica que unha clase A está composta pola clase B de tal forma que os obxectos de B teñan a mesma vida que os obxectos de A e non poden ser compartidos por outros obxectos compostos. Como consecuencia, a supresión do obxecto da clase A implica a supresión do obxecto da clase B.

Represéntase graficamente cunha liña continua cun rombo con recheo na clase A.

A composición pode levar navegabilidade para indicar que a propiedade componente se inclúe automaticamente na clase A; en caso contrario, habería que definir explicitamente esa propiedade.

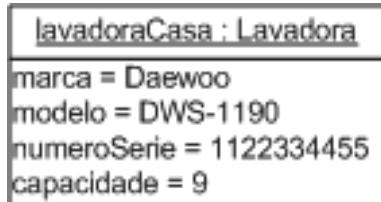
Exemplo: pedidos compostos cada un por unha ou máis liñas de pedido (unha para cada produto).



Diagramas de obxectos

Permite representar unha instancia dunha clase. O obxecto está identificado polo nome da instancia do obxecto seguido de dous puntos e o nome da clase e todo iso subliñado. O nome do obxecto iníciase con letra minúscula. Os atributos aparecen normalmente con valores.

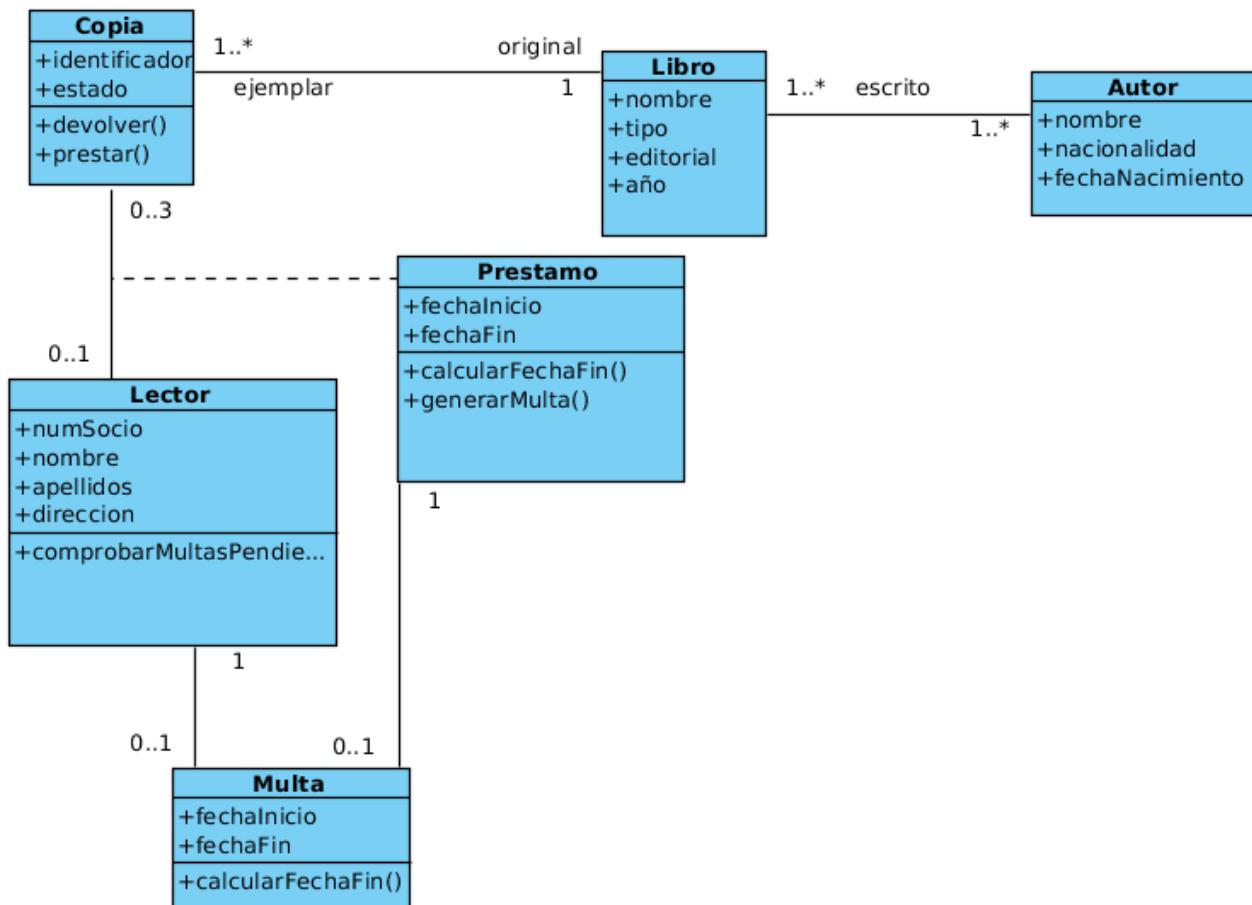
Por exemplo o diagrama do obxecto lavadoraCasa da clase Lavadora:



Os obxectos de clases relacionadas mediante unha asociación tamén están asociadas e neste caso o nome da asociación tamén aparece subliñado.

Tarefa 5.4. Interpretación dun diagrama de clases

Dado o seguinte diagrama de clases, redacta un posible escenario que se adapte á información recollida no diagrama.



En todos os exercicios seguintes non é preciso que inclúas getters nin setters nin construtores pero si debes indicar a cardinalidade nas relacóns.

Tarefa 5.5. Elaboración de diagramas de clases sinxelos.

Representa mediante distintos diagramas de clase independentes os seguintes escenarios:

a) Os periféricos que poden ser extraíbles e non extraíbles

b) Hai varios tipo de periféricos, por exemplo: os disquetes, memorias USB e os discos duros.

Hai algunha diferencia no tipo de relación entre o apartado a) e o apartado b) ?

c) Unha tarxeta de crédito ten unha serie de movementos

d) Un grupo de traballo está formado por un ou máis traballadores.

e) Hai varios tipos de produtos bancarios, cada un cos seus atributos e métodos pero todos eles deben ter un método para cobrar intereses e cobrar comisións aínda que cada un poderá empregar unha fórmula distinta, e ter atributos distintos.

f) Os teléfonos de prepago teñen un número único, un saldo e unha operación para recargar que é igual para todos os tipos de teléfono. Existen actualmente dous tipos de teléfono prepago: NavegaPlus (ten un atributo co límite de megas mensual) e outro GastaPouco (con límite de chamadas mensuais). Tamén deben ter todos unha operación para efectuar chamadas pero é distinta para cada tipo de teléfono xa que reduce o saldo con distintas fórmulas. Non se poden crear instancias de teléfono, haberá que facelo dalgún tipo concreto: GastaPouco, NavegaPlus, etc.

g) Os alumnos poden matricularse en distintos cursos dunha academia. De cada alumno queremos saber o seu nome e idade. De cada curso o seu nome e duración. Tamén queremos saber a data na que se matricula un alumno nun curso e a cualificación obtida, habendo unha operación para calcular dita cualificación.

Nota: Podes facelos coa mesma ferramenta online que fixemos os diagramas de McCabe en temas anteriores (app.diagrams.net) ou instalar xa Visual Paradigm como se mostra máis adiante, e xa facelos nesa ferramenta.

Tarefa 5.6. Elaboración dun diagrama de clases

Representa mediante diagramas de clase a seguinte:

- Unha superclase ObxectoGráfico que ten como atributos protexidos: grosorTrazo, ordenada, abscisa, color e como operacións públicas: mover(), visualizar() e xirar().

- Tres subclases Punto, Círculo (ten atributo privado diámetro) e Cadrado (ten atributo privado lado). Cada unha destas clases ten unha maneira diferente de moverse, visualizarse e xirar.

- Contesta as seguintes preguntas:

- É accesible diámetro dende Cadrado?
- Un obxecto Círculo posúe un atributo color?
- Pode aplicarse o método mover a un obxecto Punto?
- Que interese pode ter que a clase ObxectoGráfico sexa abstracta?

Estende o modelo anterior para que un novo obxecto gráfico chamado GráficoComposto estea composto de varios obxectos gráficos, de tal forma que a supresión do obxecto GráficoComposto implique a supresión dos ObxectoGráfico que o compoñen.

Tarefa 5.7. Elaboración dun diagrama de clases

Representa a través dun diagrama de clases o seguinte escenario:

- Necesitamos desenvolver unha aplicación para xestionar a información de diferentes empresas, dos seus clientes e dos seus empregados.
- Tanto de clientes coma de empregados almacenaremos o seu nome e a súa idade.
- Dos empregados almacenaremos o seu salario bruto e a aplicación terá que calcular o salario neto. Os empregados que son directivos teñen unha categoría así como un conxunto de empregados subordinados.
- Dos clientes ademais necesítase coñecer o seu teléfono de contacto.
- Para un directivo, pódese obter o listado dos seus empregados subordinados.
- Das empresas non temos ningún atributo xa que haberá varios tipos (por exemplo: autónomo, PEME, holding, etc.) e os atributos e operacións serán diferentes para cada un deles. Pero cada un deses tipos de empresa deberá desenvolver un método chamado *calcularFacturación*, que dependerá das súas características.

Tarefa 5.8. Elaboración dun diagrama de clases

Representa a través dun diagrama de clases o seguinte escenario, especificando os tipos de datos que consideres apropiados para os atributos

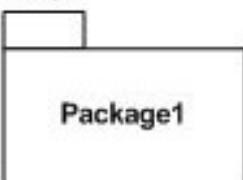
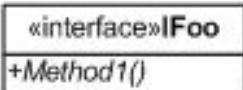
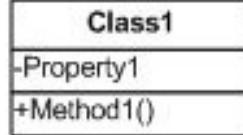
- Deséxase almacenar a información de distintos tipos de obras culturais. As devanditas obras poden ser libros, discos ou películas.
- De toda obra nos interesa almacenar o seu título e ano de edición.
- Dos libros interésanos almacenar o seu editorial e o seu número de páginas.
- Dos discos interésanos almacenar a súa discográfica e o número de cancións.
- Das películas interésanos almacenar a súa produtora.
- Ademais interésanos almacenar os distintos artistas relacionados coas obras, en concreto os autores que crearon a obra e, no caso das películas, os actores que a interpretan.
- Dos artistas interésanos o seu nome e ano de nacemento e todos deben ter unha operación para calcular a súa remuneración pero depende do que sexan: autores, actores, etc.

Tarefa 5.9. Elaboración dun diagrama de clases

Representa mediante un diagrama de clase a xestión dunha conferencia científica coas consideracións seguintes:

- A conferencia pode ter varias sesións.
- Unha sesión posúe fecha e hora de inicio, pertence só a unha conferencia e non ten razón de ser sen unha conferencia.
- Os participantes nunha sesión poden ser oradores ou público. Todos eles teñen que inscribirse na conferencia. Pode cancelarse ou confirmarse unha inscrición. Tamén queremos gardar a data na que se inscribiu na sesión.
- Un ou máis artigos científicos preséntanse nunha sesión. Cada artigo pode ser curto ou longo e trata dun tema determinado.
- Un autor pode ter un ou varios artigos presentados na conferencia.
- Coloca os atributos que che parezan convenientes.

UML Cheatsheet

Shape	Description
	Package A collection of interfaces and classes.
	Interface Microsoft guidelines specify that interfaces should start with I. This graphic can also sometimes be used as an abstract class.
	Class Properties or attributes sit at the top, methods or operations at the bottom. + indicates public and # indicates protected.
B —————> A	These are both typically drawn vertically: Inheritance - B inherits from A. "is-a" relationship.
B - - - - -> A	Generalization - B implements A,
A ————— B	Association - A and B call each other
A ◊———— B	Aggregation A "has-a" instance of B. B can survive if A is disposed.
A ♦———— B	Composition A has an instance of B, B cannot exist without A.
	A note Some descriptive text attached to any item.

Associations and aggregation/composition can have *, 1 or n attached to either end of the relationship.

3. Ferramentas para o traballo con diagramas de clases

3.1 Introducción

Aínda que o traballo con diagramas UML é perfectamente posible sen o emprego de ferramentas software, o mais habitual e empregar aplicacións que nos faciliten a representación de ditos diagramas e que incorporen a maiores unha serie de funcionalidades para o traballo con eles como poden ser a xeración automática de código a partir dos propios diagramas ou incluso a posibilidade de facer “enxeñería inversa” a partir do noso código, e dicir, xerar diagramas UML a partir de código fonte do noso proxecto.

3.2 Criterios para a selección dunha ferramenta UML

Aspectos a ter en conta para elixir unha ferramenta que permita elaborar e traballar con modelos UML:

- Licenza e prezo.
- Plataforma sobre a que traballa.
- Tipo de diagramas que permite.
- Linguaxes de programación que permita asociar ao modelo e por tanto que poida xerar código a partir de diagramas.
- Posibilidade de xerar automaticamente documentación.
- Posibilidade de enxeñaría inversa, é dicir, xerar diagramas a partir de código.
- Facilidade de navegación no modelo.
- Posibilidades de exportación do modelo.
- Interface de comunicación có usuario.

Existen centos de ferramentas UML independentes con licenza propietaria ou libre e moitos contornos de desenvolvemento que poden utilizar ferramentas UML.

Comparativas

Relación de sitios web con comparativas entre ferramentas UML:

http://en.wikipedia.org/wiki/Comparison_of_Unified_Modeling_Language_tools

Algunhas ferramentas Open Source

Dia (<http://dia-installer.de/index.html.en>)



Umbrello (<http://uml.sourceforge.net/>)

PlantUML: (<https://plantuml.com/es/>)

ArgoUML (<http://argouml.tigris.org/>)



Papyrus: (<https://www.eclipse.org/papyrus/>) É a ferramenta empregada co IDE Eclipse.

Modelio (<http://www.modelio.org/>)

StarUML: (<http://staruml.io/>)

NetBeans (<https://apache.netbeans.org/>) é un contorno de desenvolvemento que dispón dun plugin ou complemento para realizar diagramas UML chamado EasyUML

Ferramentas gratuítas

[JPlantUML](#) é unha das ferramentas más completas e empregadas na actualidade na que os diagramas constrúense a partires dunha linguaxe propia.

[JDeveloper](#) é un contorno de desenvolvemento integrado desenvolvido por Oracle que simplifica o desenvolvemento de aplicacións baseadas en Java de xeito que contempla cada paso do ciclo de vida dunha aplicación.



Algunhas ferramentas propietario

Visual Paradigm (<http://www.visual-paradigm.com/>) é una potentísima ferramenta que nos permite traballar con proxectos UML. Ver documetación: <https://www.visual-paradigm.com/guide/> Dispón dunha versión gratuita para uso non comercial.



UModel (<http://www.altova.com/es/umodel.html>) defínese como unha ferramenta UML para deseñar software de forma visual. Deseña modelos de aplicacións con UML de forma visual e xera código Java, C++, C# o Visual Basic .NET, así como documentación de proxecto. Converte programas en diagramas UML mediante enxeñería inversa. Dispón dunha versión de proba de 30 días.



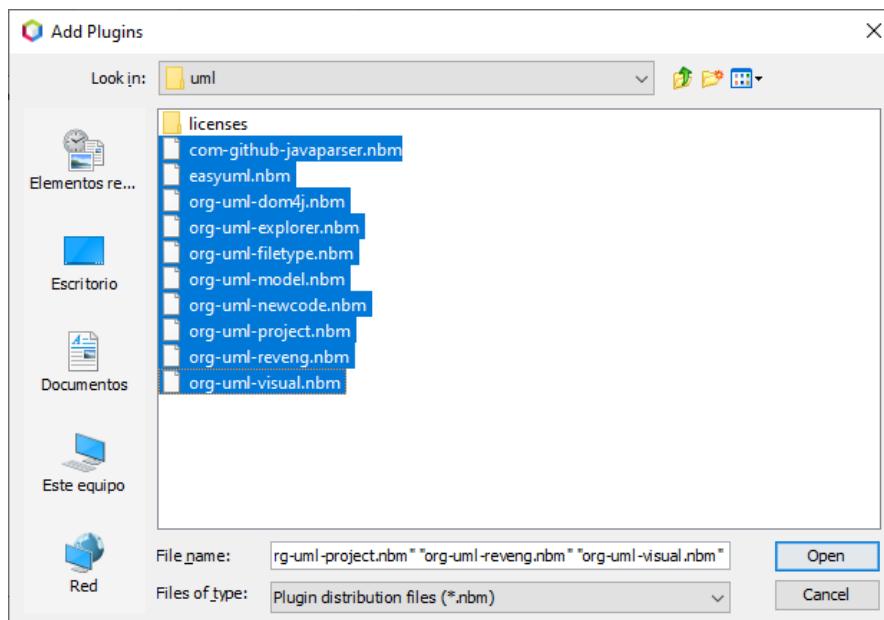
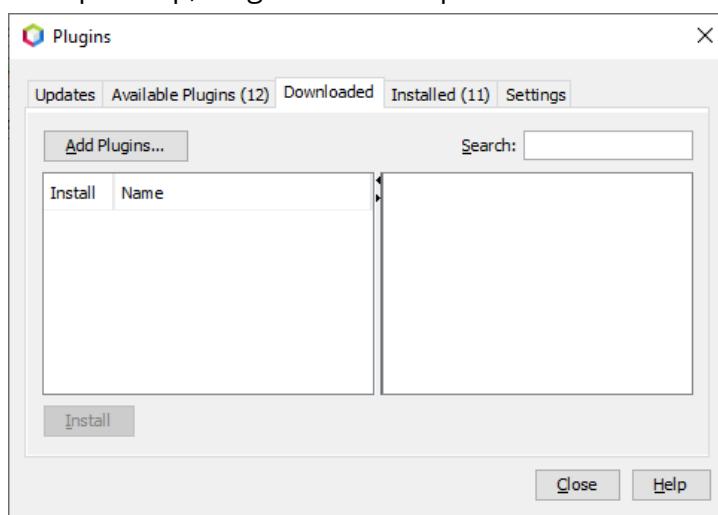
3.3 NetBeans. Módulo UML

Instalación

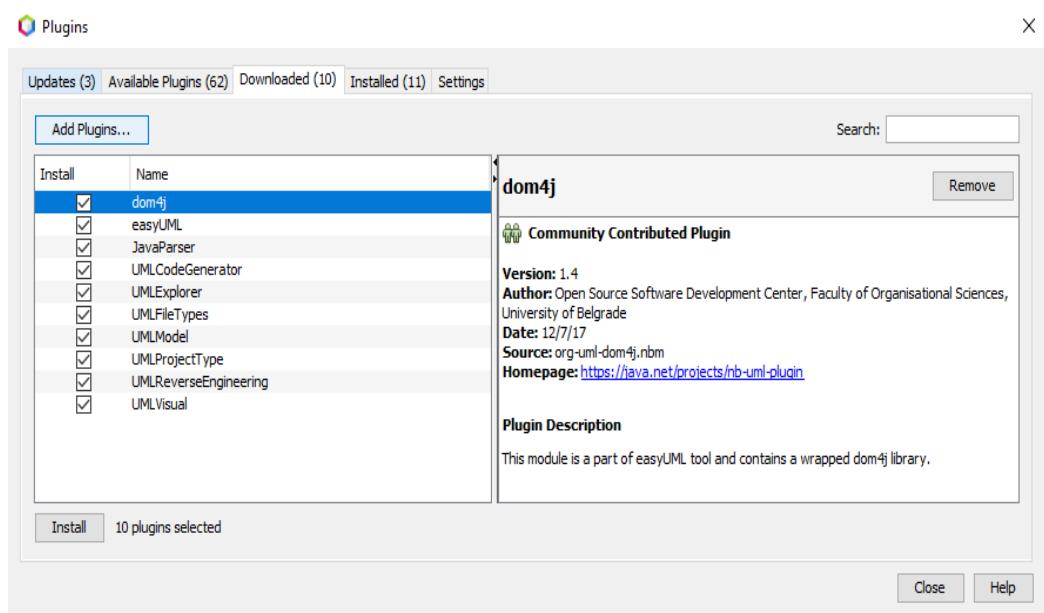
O contorno de desenvolvemento NetBeans permite instalar un módulo para o traballo con diagramas UML que ademais de aportarnos ferramentas gráficas para a representación dos distintos diagramas vai a darnos a posibilidade de xerar código a partir dos diagramas que teñamos feito e incluso poderemos facer "enxeñería inversa" a partir do noso código, e dicir, xerar diagramas UML a partir de código fonte do noso proxecto.

Imos ver como instalar o módulo easyUML:

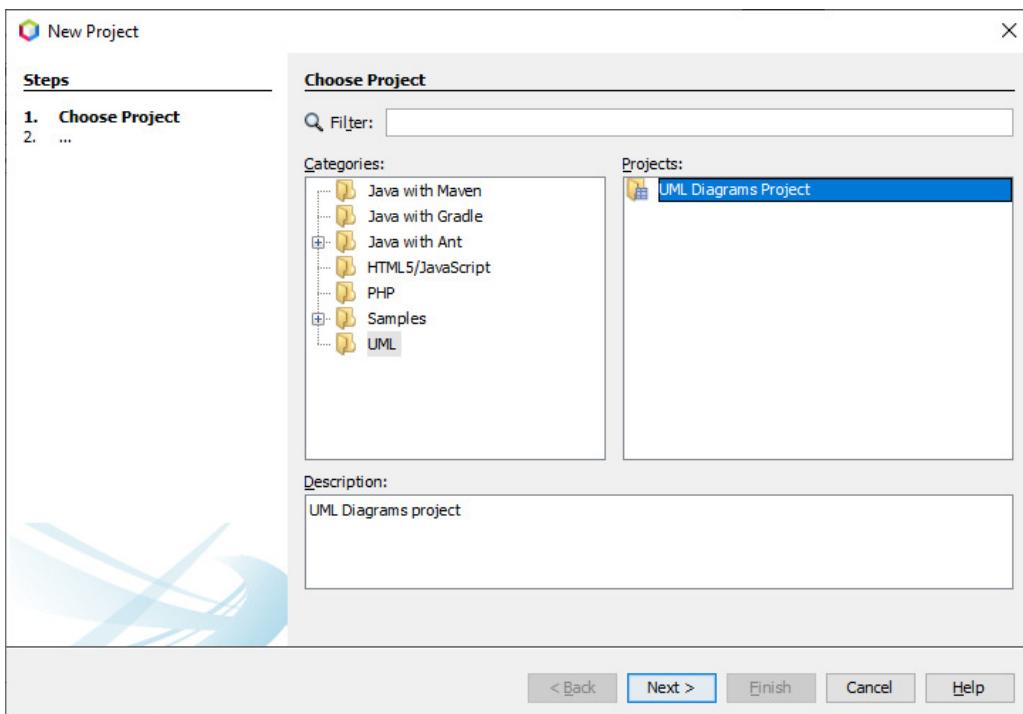
- 1) Iremos á opción *Plugins* do menú *Tools*:
- 2) Non aparece o módulo easyUML polo que hai que descárgalo na súa última versión dende <http://plugins.netbeans.org/plugin/55435/easyuml>
- 3) Descomprimimos o arquivo zip, engadindoos na pestana de Downloaded premendo Add



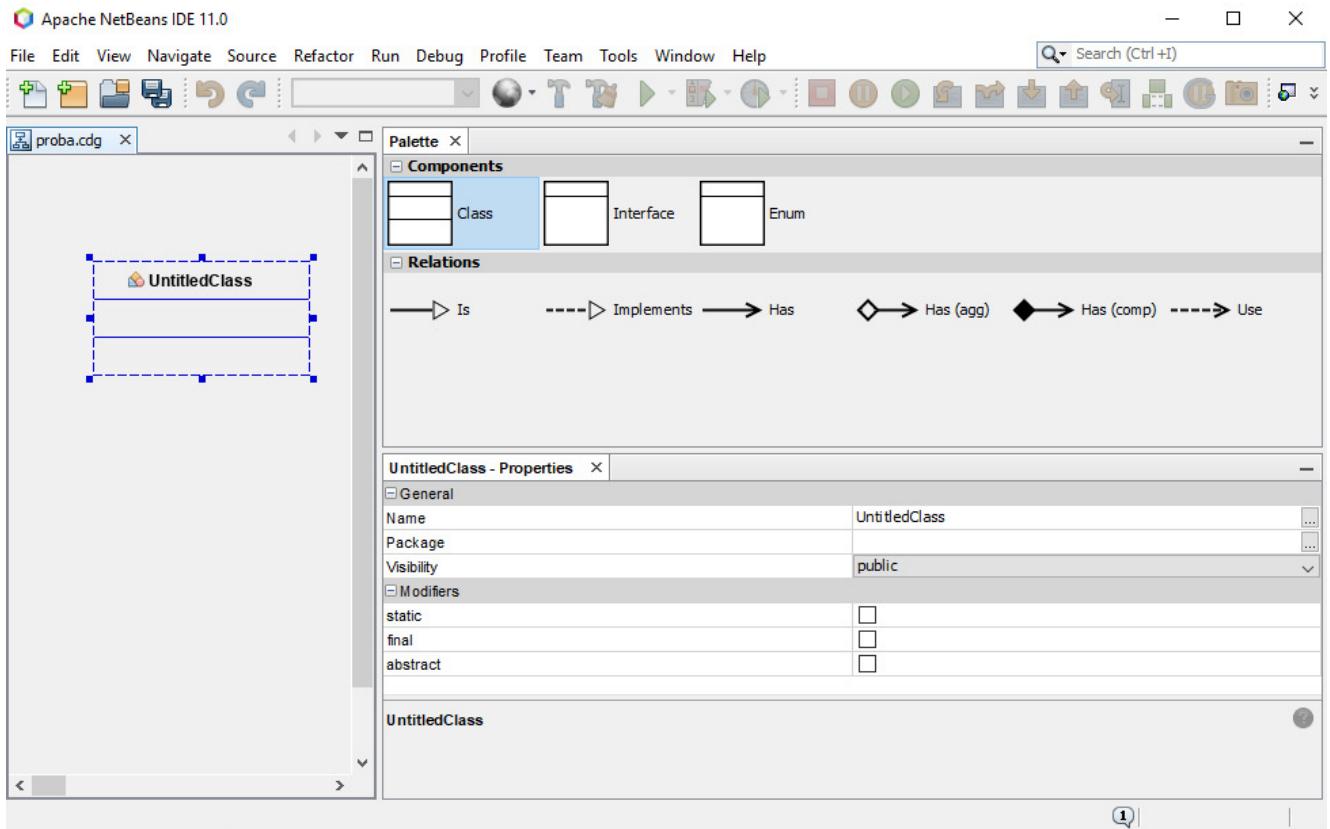
4) E prememos Instalar.



Finalmente, despois da instalación comprobaremos que na opción de crear un proxecto novo nos parece a posibilidade de crear un proxecto UML:



E xa podemos empezar a traballar:

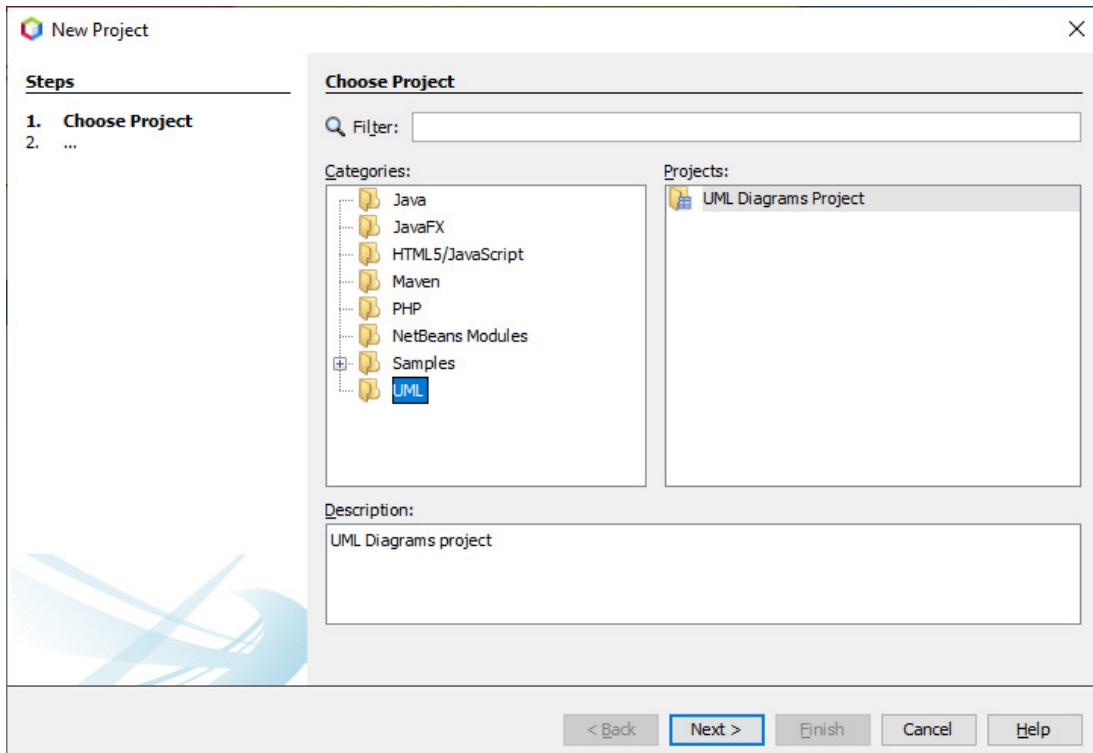


Xeración de código a partir de diagramas de clases

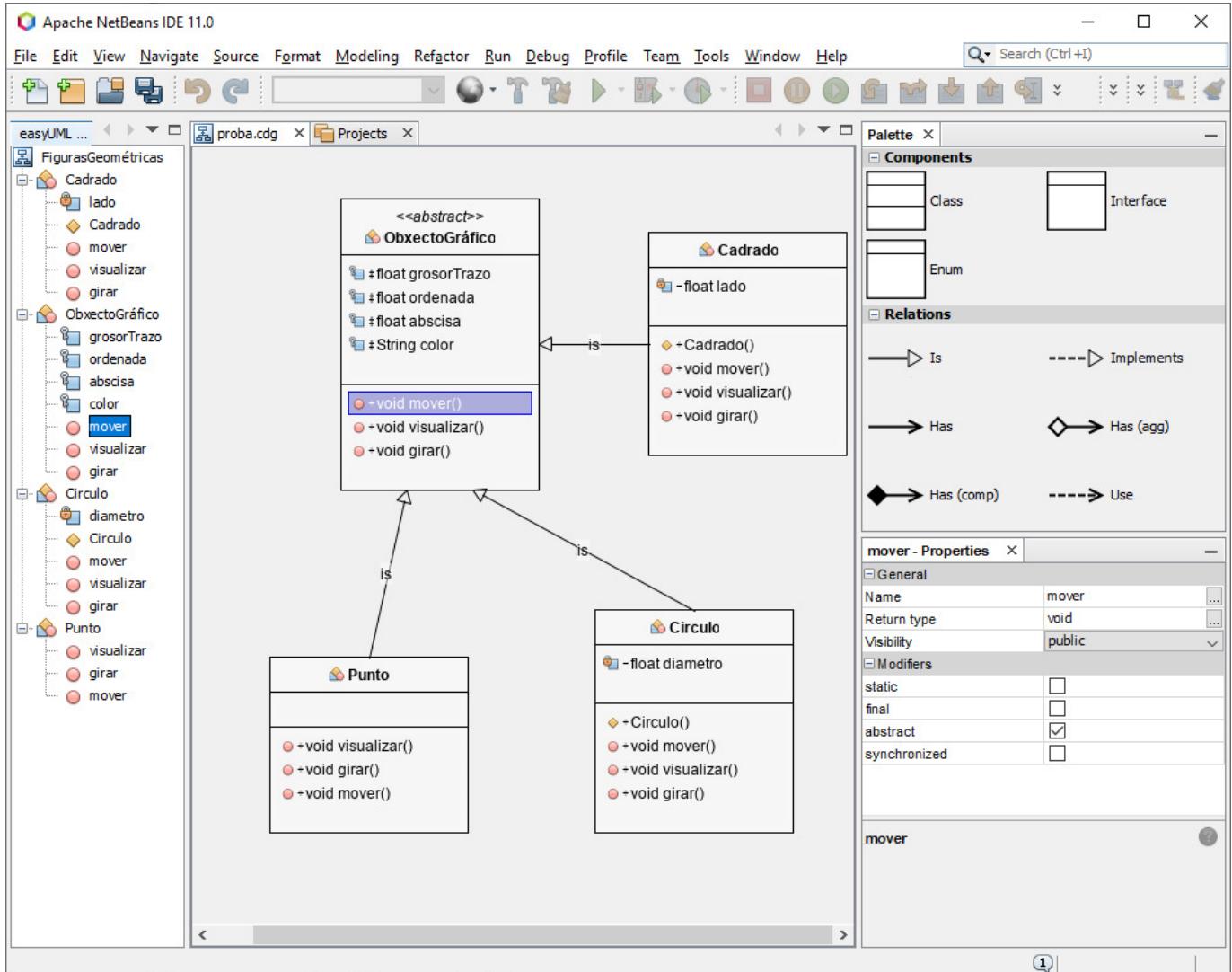
Imos ver como se pode xerar código co módulo UML de Netbeans a partir dun diagrama de clases existente.

En primeiro lugar temos que xerar un diagrama UML onde se van situar tódolos nosos diagramas UML que xeremos dos diferentes proxectos.

Netbeans-> New project-> UML Diagrams Project



Despois imos xerando os diferentes elementos que componen o noso diagrama de clase. No noso caso estamos traballando cun obxecto xenérico chamado ObxectoGráfico que ten tres clases fillas: Cadrado, circulo e punto.



É máis limitado que Visual Paradigm e ten algúns erros como non poñer en cursiva os métodos abstractos, nin marcar os atributos/métodos protexidos con #.

Tampouco ofrece relacións de asociación simple (hai que empregar 'Use')

Agora temos que crear o proxecto Java onde imos situar o código que se xenera co diagrama de clases anterior. Para iso imos a File->New Project e lle poñemos de nome ObxectoGrafico

Situando o rato enriba do nome do diagrama de clases e premendo no botón derecho, seleccionaremos a opción de *EasyUML generate code..*. Seleccionamos o nome do proxecto onde imos a xerar o código e pulsamos *Aceptar*.

```
public class Círculo extends ObxectoGrafico {
    private float diametro;
    public Círculo() {
    }
    public void mover() {
    }
    public void visualizar() {
    }
    public void girar() {
    }
}
```

Agora aparécenos as clases co código xeradas.

Debemos facer algunha corrección como eliminar o arquivo ObxectoGrafico.java do paquete Obxectografico, xa que é o arquivo baleiro que nos xera Java ó crear o proxecto. Eliminar o modificador de protexido na clase ObxectoGrafico e podemos mover tódalas clases a un paquete que tamén chamamos obxectografico

```
package obxectografico;
abstract class ObxectoGrafico {
    protected float grosorTrazo;
    protected float ordenada;
    protected float abscisa;
    public abstract void mover();
    public abstract void visualizar();
    public abstract void girar();
}
```

Tarefa 5.10. Instalación e manexo do módulo UML de NetBeans

Empregando o teu contorno de desenvolvemento NetBeans, realiza os seguintes apartados:

- Segundo as indicacións dadas no texto da actividade instala o módulo de UML no contorno de desenvolvemento.
- Usando o módulo UML instalado inventa un diagrama de clases que inclúa herdanza e agregación.
- Crea en NetBeans un novo proxecto Java baleiro e xera automaticamente código a partir do diagrama feito previamente.

3.4 Visual Paradigm

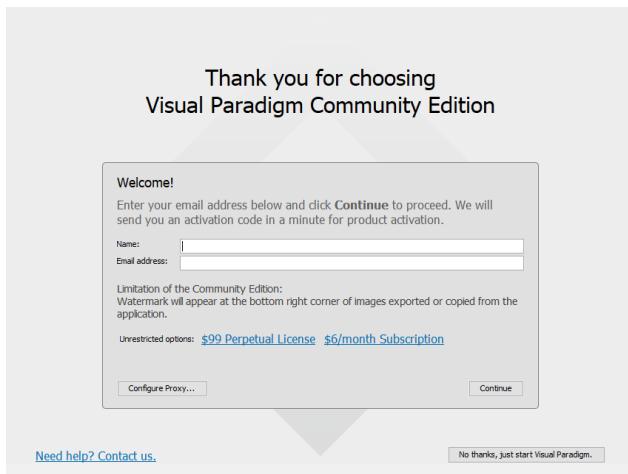
Instalación de Visual Paradigm Community Edition

Visual Paradigm é unha das ferramentas mais potentes para o traballo con diagramas UML e outro tipo de diagramas. É software propietario pero dispón dunha versión de proba (FREE Trial) completamente funcional pero limita o seu uso a 30 días. Tamén dispón dunha versión gratuíta para uso non comercial chamada: *Community Edition*. Para a súa descarga desta versión iremos á páxina de descargas e seleccionaremos esta edición *community* para o sistema operativo que estamos empregando (<http://www.visual-paradigm.com/download/community.jsp>):

Unha vez descargada, a executamos e comenzará o proceso guiado de instalación que non ten ningunha dificultade:



Unha vez instalado, a primeira vez que o executamos pediranos que solicitemos unha clave de activación a través do enderezo web. Este paso non e necesario aínda que si recomendable xa que se non a aplicación estará recordándonos continuamente que traballamos cunha versión sen activar:



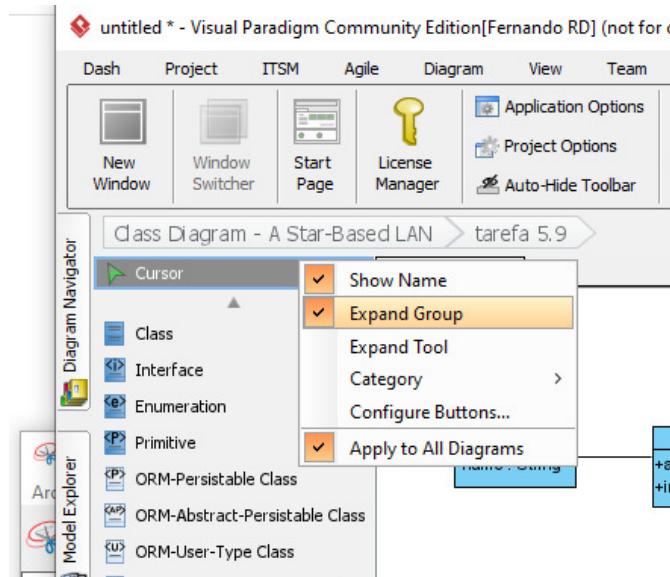
Traballando con Visual Paradigm

Sobre a ventá de comezo "Welcome to Visual Paradigm CE"

En UML prememos o botón "+"

Seleccionamos "Class Diagram" e logo partimos dun dos "Templates" que nos ofrece. Dámolle un nome ao diagrama e começamos a súa edición.

Na barra lateral de Ferramentas, prememos na súa cabeceira e seleccionamos a acción "Expand group" e teremos todas as opcións á vista: interfaces, agregacións...



Para agregar unha asociación, agregación ou composición, seleccionámola la caixa de ferramentas, prememos na clase orixe, e sen soltar, arrastramos ata a clase destino. Unha vez, engadida, sobre os seus extremos, botón dereito > *Multiplicity* podemos engadir a cardinalidade: 0..1, 1..*, etc.

Nas clases e interfaces, para engadir atributos e operacións (métodos), podemos premer nella clase, botón dereito > *Add*. Pero é máis rápido: **Alt+May+A** e **Alt+May+O** respectivamente para atributos e operacións.

UML

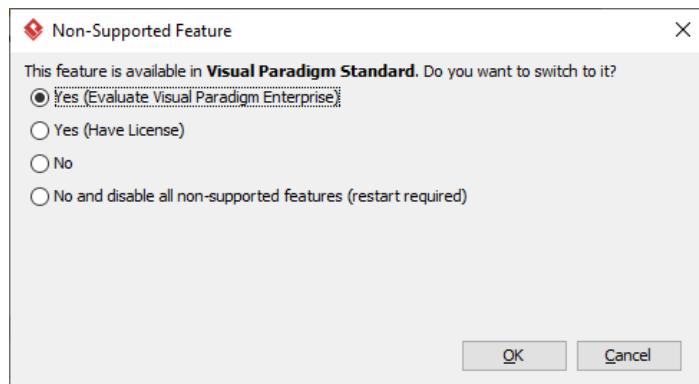
Design software system by drawing UML diagrams

- Use Case
- Deployment
- Class
- Package
- Sequence
- Object
- Communication
- Composite Structure
- State Machine
- Timing
- Activity
- Interaction Overview
- Component

Xeración de código a partir de diagramas de clases

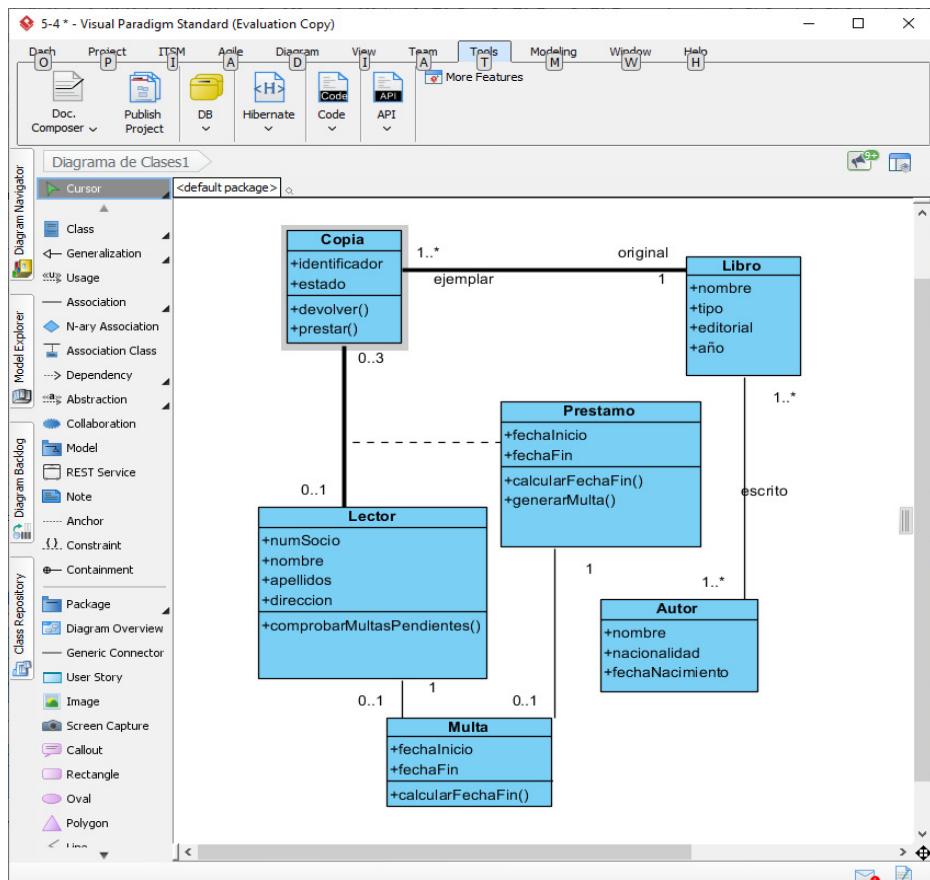
Tanto para a xeración de código como para o traballo con enxeñería inversa, Visual Paradigm permite traballar cunha tecnoloxía que chaman *Round-Trip* de tal forma que calquera modificación que fagamos no código das nosas clases ou no propio diagrama UML van verse reflectidos automaticamente nos diagramas UML ou no código fonte respectivamente. Esta funcionalidade permite manter ó código fonte e a documentación de deseño sincronizadas.

A versión gratuita de Visual Paradigm non permite o traballo con estas funcionalidades de Enxeñería de Código (Submenú *Tools->Code-> Reverse Java Code*) polo que teremos que utilizar a versión de proba de 30 días. Dende a propia versión *Community Edition* permítensenos facer o cambio (non é necesario descargar e instalar outra versión). Cando accedemos a unha funcionalidade non soportada pola versión gratuita sairanos esta pantalla:

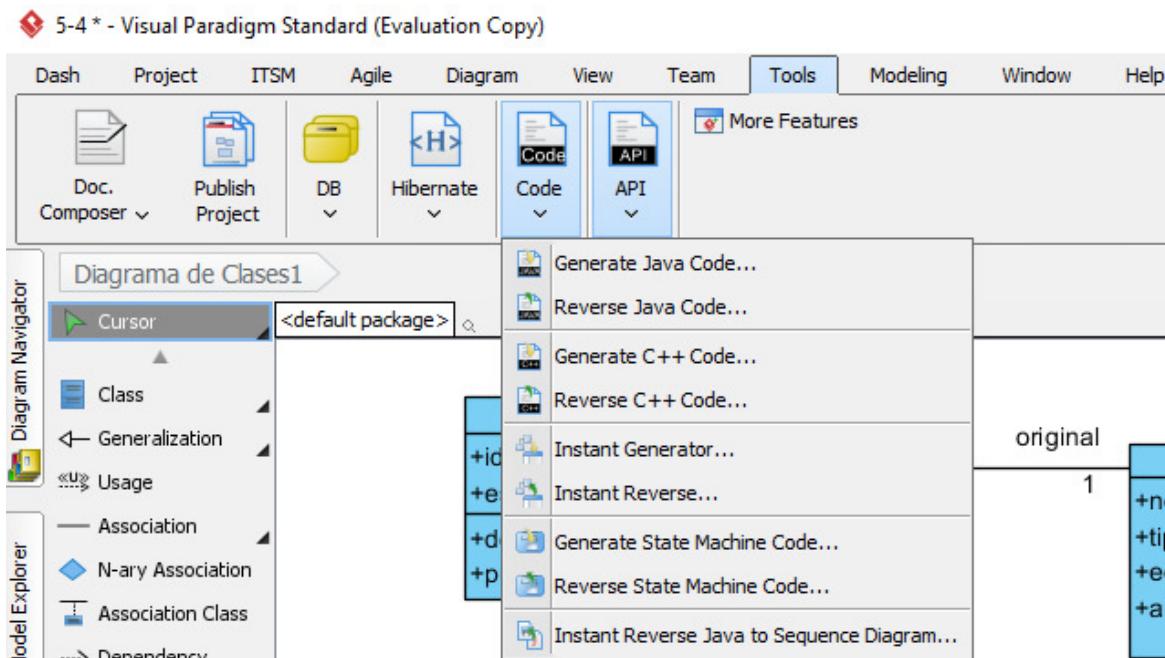


Se confirmamos o cambio guiaranos para empezar a utilizar a versión de proba (semre poderemos volver a versión gratuita cando queiramos).

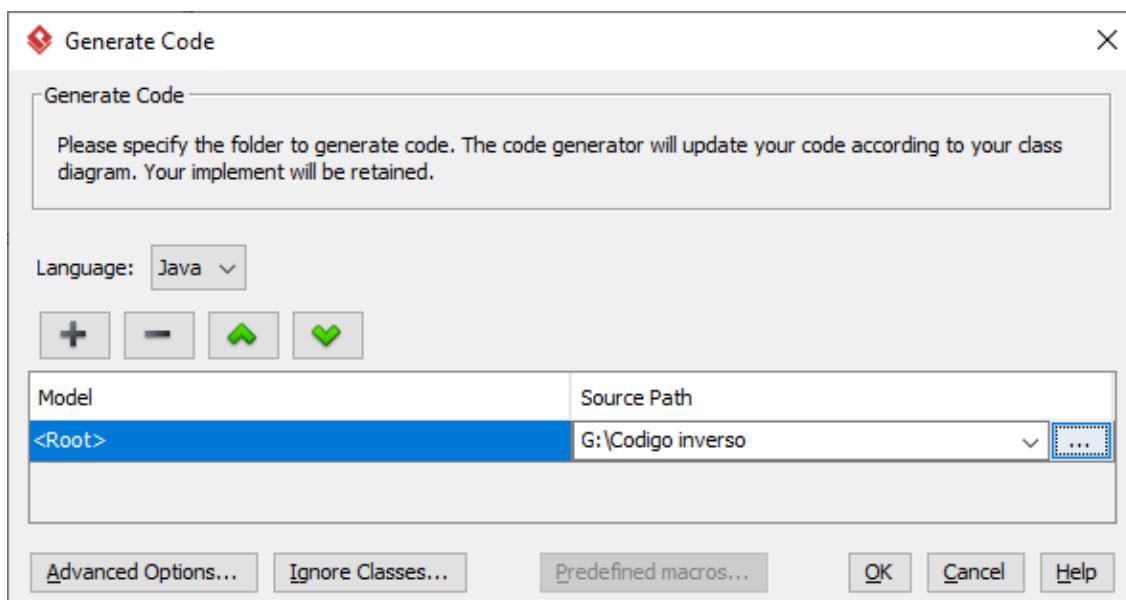
Unha vez na versión de proba, seleccionamos o proxecto que contén o diagrama de clases a partir do cal queremos xerar código:



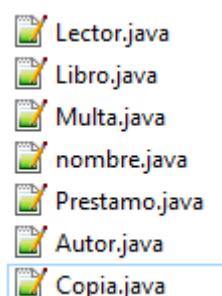
Imos a opción *Tools->Code->Generate Java Code*:



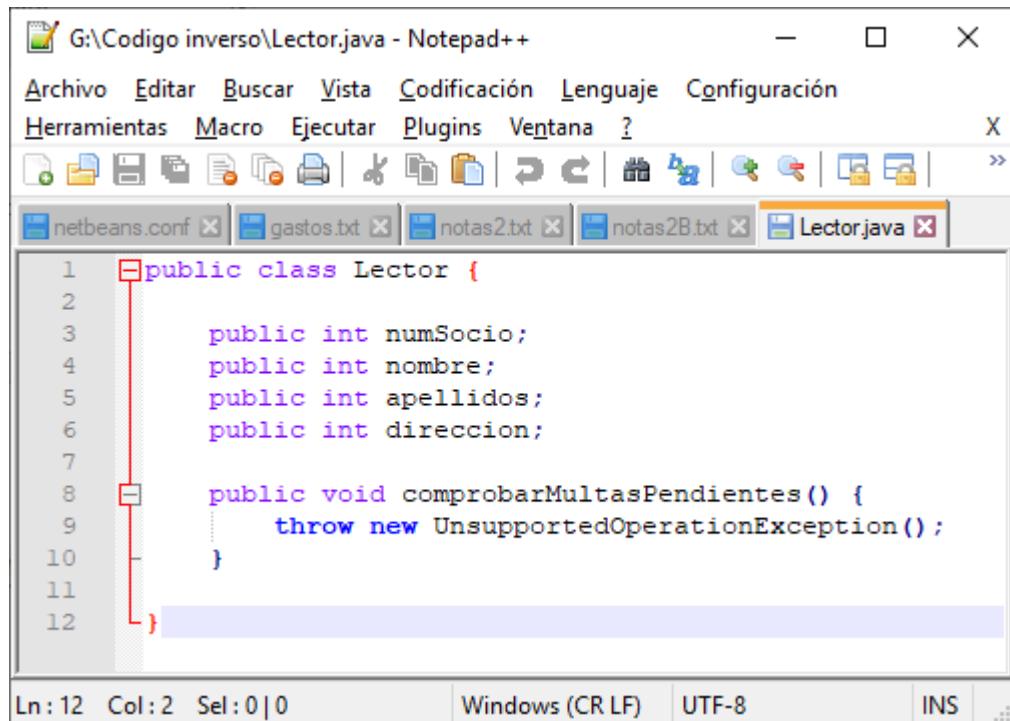
Seleccionamos o directorio destino onde queremos que se garden os ficheiros xerados e pulsamos *OK*. Se queremos personalizar o código xerado podemos modificar varios parámetros no botón *AdvancedOptions*.



Podemos ver no directorio os ficheiros creados:



E o código dalgún dos ficheiros:



The screenshot shows a Notepad++ window with the title "G:\Codigo inverso\Lector.java - Notepad++". The menu bar includes Archivo, Editar, Buscar, Vista, Codificación, Lenguaje, Configuración, Herramientas, Macro, Ejecutar, Plugins, Ventana, and ?.

The toolbar contains icons for file operations like Open, Save, Print, and Find.

The tab bar shows several files: netbeans.conf, gastos.txt, notas2.txt, notas2B.txt, and Lector.java (which is currently selected).

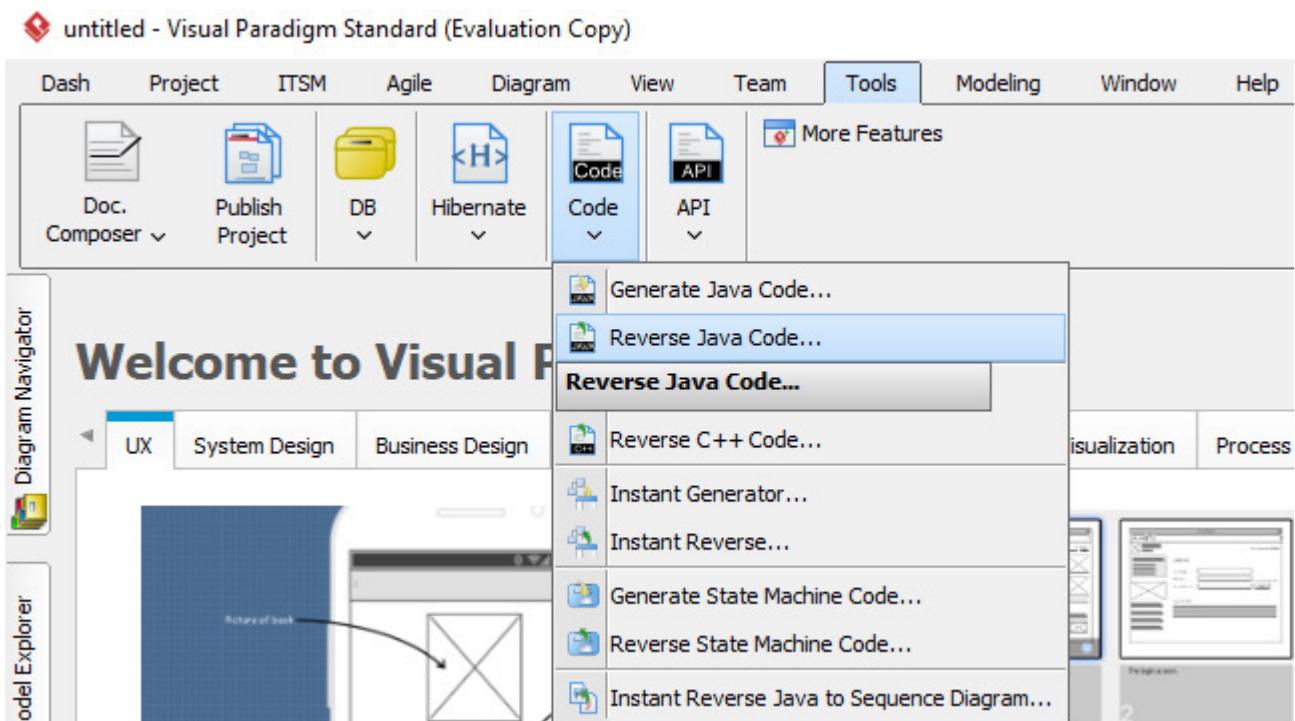
The code editor displays the following Java code:

```
1 public class Lector {  
2     public int numSocio;  
3     public int nombre;  
4     public int apellidos;  
5     public int direccion;  
6  
8     public void comprobarMultasPendientes() {  
9         throw new UnsupportedOperationException();  
10    }  
11}  
12
```

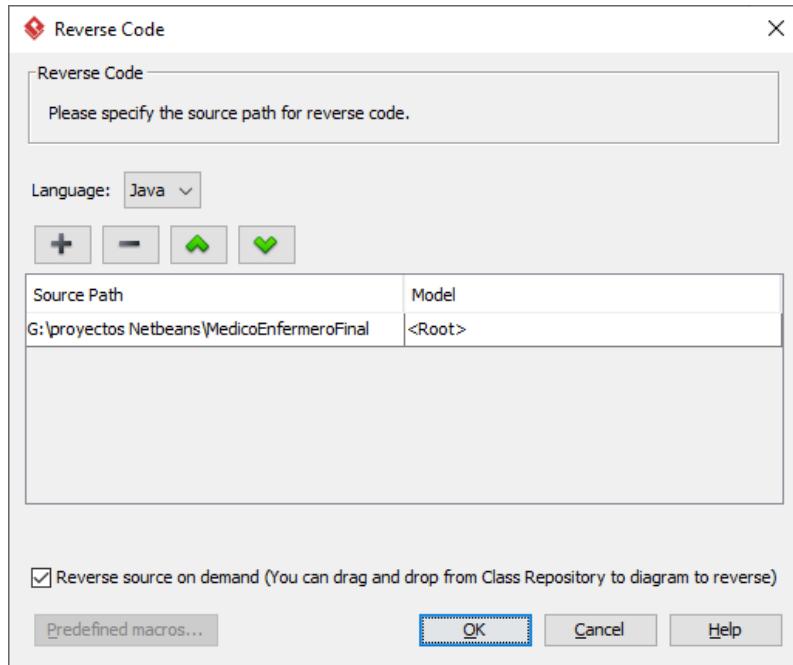
The status bar at the bottom shows Ln:12 Col:2 Sel:0|0, Windows (CR LF), UTF-8, INS, and a few other icons.

Xeración de diagramas de clase a partir de código (enxeñería inversa)

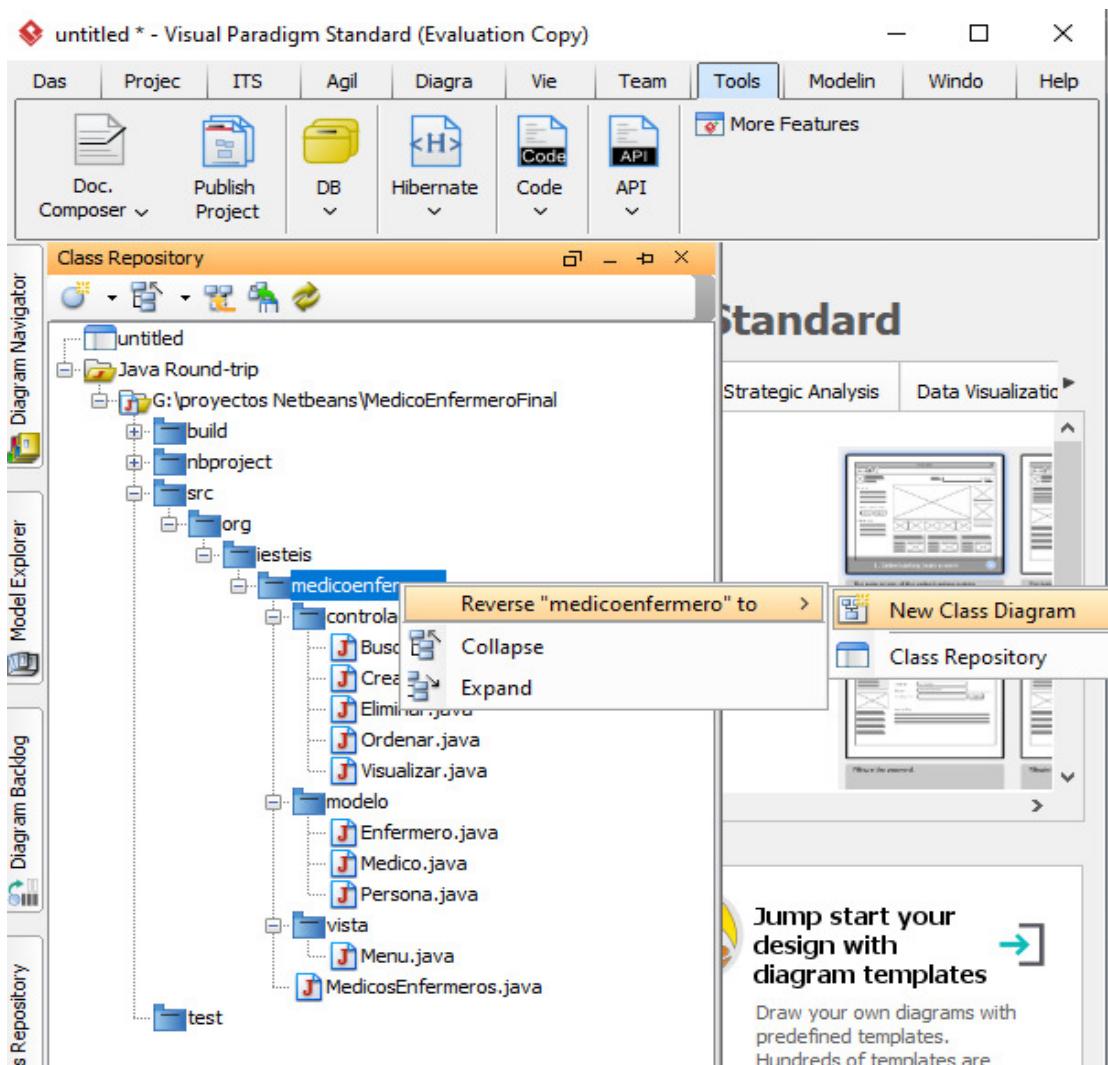
Para xerar diagramas de clase a partir de código fonte xa existente primeiramente creamos un proxecto novo no que imos crear os nosos diagramas e a continuación iremos ó submenú *Tools->Code ->Reverse Code* (neste caso crearemos os diagramas de clases do código Java do proxecto Médico-Enfermeiro):



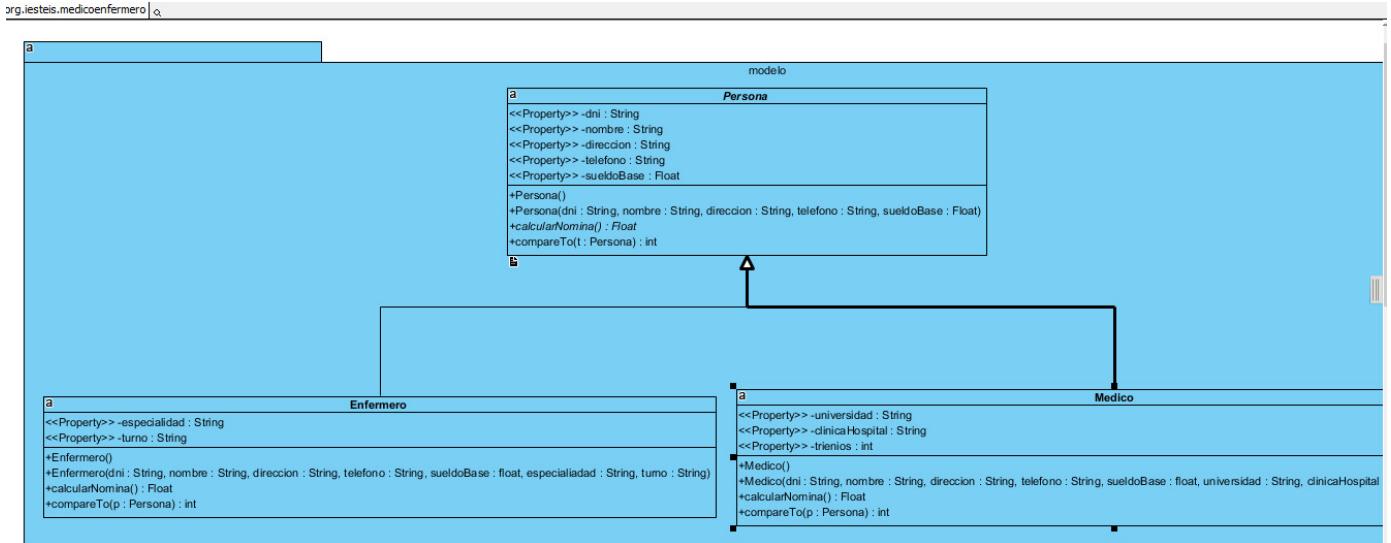
Seleccionamos o directorio onde están os ficheiros có código fonte e pulsamos *OK*:



Despois disto, no repositorio de clases do menú esquierdo aparecerán as clases atopadas. Seleccionamos todas e co botón derecho do rato dicímoslle que xere un novo diagrama de clases:



Finalmente vemos o diagrama xerado:



Tarefa 5.11. Instalación e manexo de Visual Paradigm

Seguindo as indicacións dadas no texto da actividade, instala a versión gratuíta de Visual Paradigm e despois desenvolve os diagramas de clases das tarefas 5 á 9 deste tema.