

1.	A01. Introducción a UML e á Orientación a Obxectos.....	2
1.1	Introdución.....	2
1.2	Actividade.....	2
	Orientación a obxectos.....	2
	Clases e obxectos.....	2
	Construtores e destrutores.....	4
	Abstracción.....	7
	Encapsulamento.....	8
	Herdanza.....	9
	Polimorfismo.....	10
	Envío de mensaxes.....	12
	Relacións entre obxectos.....	12
	Métodos de modelaxe.....	14
	Diagrama de fluxo de datos.....	14
	Diagramas entidade-relación.....	15
	Diagramas HIPO.....	16
	Diagramas VTOC.....	17
	Diagramas Warnier-Orr.....	17
	Á Linguaxe Unificada de Modelado (UML).....	18
	Por que é necesario o UML?.....	18
	A concepción do UML.....	18
	Diagramas do UML.....	19
1.3	Tarefas.....	27
1.3.1	Tarefa 1. Identificación de elementos da programación orientada a obxectos (POO).....	27

1. A01. Introducción a UML e á Orientación a Obxectos

1.1 Introducción

Na actividade que nos ocupa aprenderanse os seguintes conceptos e manexo de destrezas:

- Manexar os conceptos básicos da programación orientada a obxectos.
- Distinguir as principais técnicas de modelaxe empregadas en proxectos software.
- Coñecer as principais características do modelo UML e identificar os seus diagramas mais empregados.

1.2 Actividade

Orientación a obxectos

A programación orientada a obxectos (POO) é un paradigma. Un paradigma de programación representa un enfoque particular ou filosofía para a construción 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 máis 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 realízase 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* aínda 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 tivéssemos que crear un sistema que manexase información acerca das persoas sería moi probable que incorporásemos algúns dos seus atributos e accións no noso software.

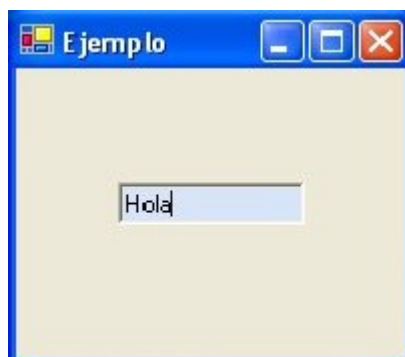
Para poñer un exemplo¹ de clase software imos a empregar a plataforma *.Net*, nela temos a clase `TextBox` que ten como propiedades `BackColor` para indicar a cor de fondo, `Cursor` para indicar a forma do punteiro do rato cando nos situamos sobre un `TextBox`, etc. Como métodos ten por exemplo `Clear` 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.



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

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

Construtores e destrutores.

O *constructor* 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.

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 enteiras), 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
```

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

```

        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 Tempo`; isto non é máis que a declaración da nosa clase. As clases polo xeral son de tipo `public`, aínda 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
{
    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: `nome_Clase nome_obxecto = new nome_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: `obxecto.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 crave `value` para asignar un valor ao atributo.

Na seguinte clase utilízase 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 según 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 facer fronte a sistemas complexos que debemos modelizar. 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 vantaxe 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 como *interfaz*.

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 o seu

interfaz (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, lavaplatos, radios, licuadoras e planchas 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 herdán 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 facilítanos a implementación de algoritmos que cumpren 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 sobreescibindo outro se utiliza a palabra crave `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 traballador
```

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 elas 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 poden 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 cómo 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 constructor haga
    }
    public void setCreditCard(CreditCard creditCard)
    {
        this.creditCard = creditCard;
    }
    // Más código aquí
}
```

Como implementar Composición

Representaremos a relación: o portátil ten un teclado.

```
public class Laptop
{
    private String manufacturer;
    private String model;
```

⁴ Padilla Donato, Humberto. *Desarrolla Sw utilizando POO*. <https://sites.google.com/a/cecylteg.edu.mx/desarrolla-sw-utilizando-poo/atributos-compositivo-y-asociativo>

```

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.



Realizar a Tarefa 1 consistente en identificar os conceptos básicos da orientación a obxectos en distintos listados de código fonte.

Métodos de modelaxe

A día de hoxe os diagramas máis utilizados nas fases de análise e deseño de calquera proxecto software son os propostos por UML, pero antes de comezar a falar desta linguaxe de modelado convén realizar un repaso por outras técnicas de representación que foron utilizadas e que en nalgúns casos, como os diagramas entidade-relacion, 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 estratéxico e de negocios.

■ Compoñentes

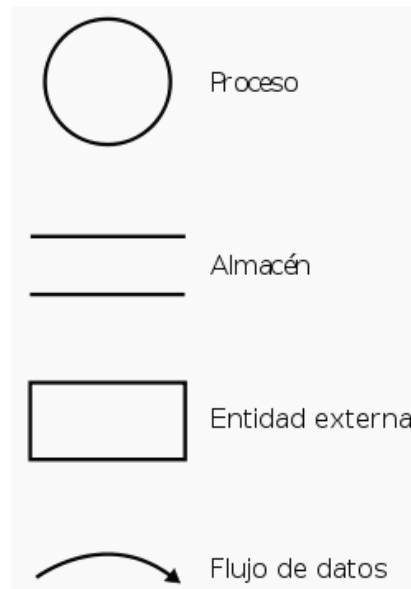


Fig. 3

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

Algúns analistas prefiren 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-objeto 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.

- *Almacéns*. - O almacén utilízase 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

⁵ <http://www.virtual.unal.edu.co/cursos/sedes/manizales/4060030/lecciones/Capitulo%204/dfd.htm>



Fig. 4

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.

■ Componentes 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. Representáase 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. Representáanse con círculos ou óvalos.
- *Relación*. Describe certa dependencia entre entidades ou permite a asociación destas. Representáase cun rombo.

■ Exemplo⁶:

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

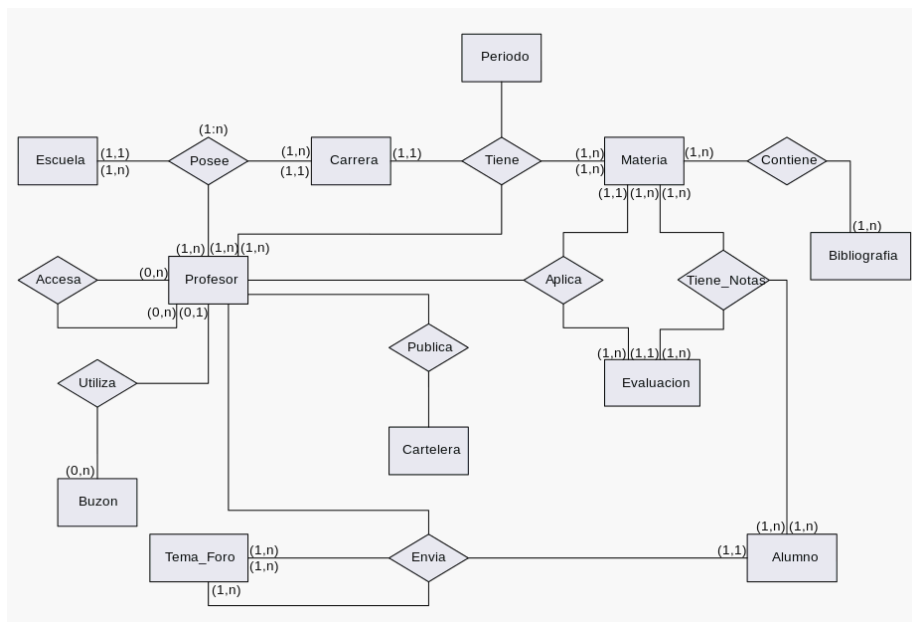
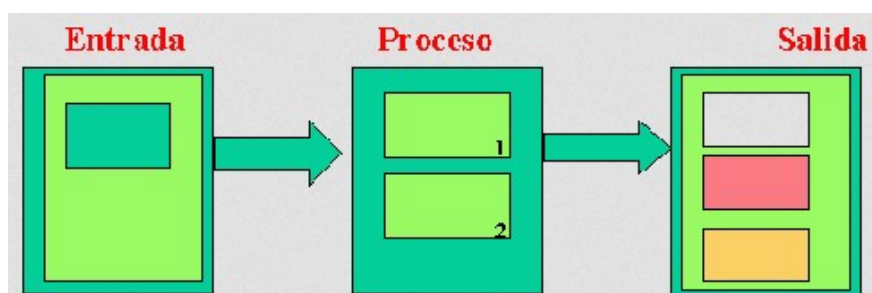


Fig. 5

Diagramas HIPO⁷

HIPO 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

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 doado 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 folia en que se debuxa o

⁷ http://www.sites.upiicsa.ipn.mx/polilibros/portal/Polilibros/P_externos/Administracion_informatica_de_las_organizaciones/Ramon_E_Enriquez_Gonzalez/AIO3_FlujoDatos.html

diagrama déixase un espazo para unha descrición mais detallada dos cadros.

■ **Exemplo:**

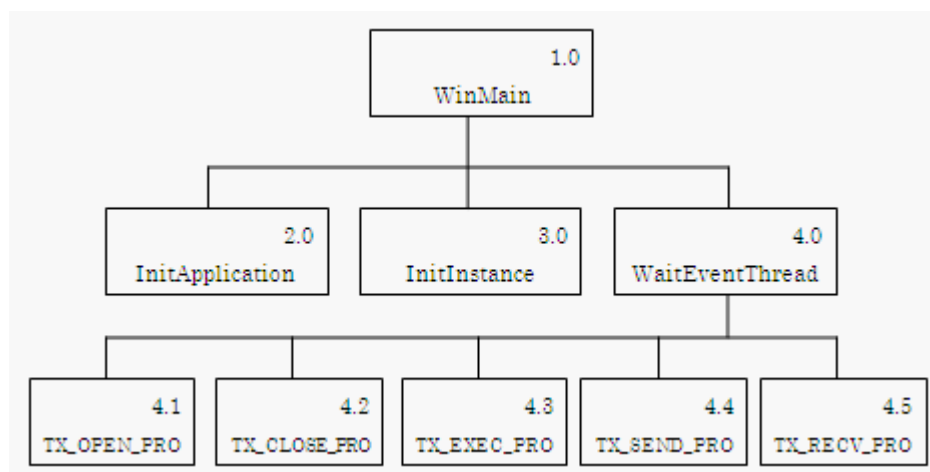


Fig. 6

Diagramas Warnier-Orr

Os diagramas de Warnier/Orr (tamén coñecidos como construción lóxica de programas/construción 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 máis claros os movementos dos datos nos devanditos niveis.

■ **Exemplo⁸:**

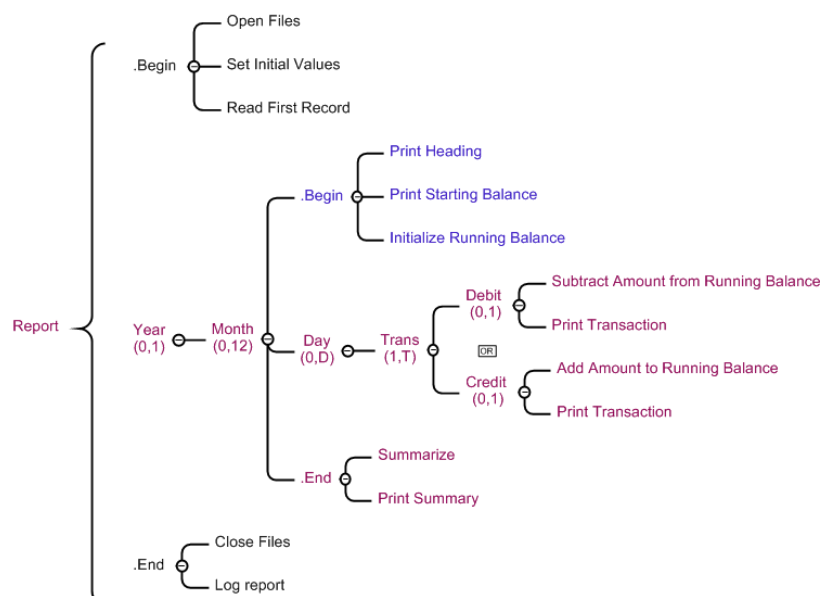


Fig. 7

⁸ <http://mr-ahmadsyofiya.blogspot.com.es/>

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 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 comprendano 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 é 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 é 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/>. Destacan:

- *UML 1.4.2*. que é o estándar ISO/IEC 15939 edición 2005
- *UML 2.4.1*. que é o estándar ISO/IEC 15939-1 e ISO/IEC 15939-2 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 última versión lanzada por OMG en agosto de 2011.
- *UML 2.5* é unha versión que OMG ten en proceso actualmente (Novembre 2014) e que empezou en outubro de 2012.

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 máis detallada do modelo pero non é unha ferramenta cómoda de comunicación, sería máis 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 táboa⁹:

Diagrama	Descrición	Prioridade
Diagrama de Clases	Mostra unha colección de elementos de modelado declarativo (estáticos), tales como clases, tipos e os seus contidos e relacións.	Alta
Diagrama de Componentes	Representa os compoñentes dunha aplicación, sistema ou empresa. Os compoñentes, as súas relacións, 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 construción interna pode ser representada por nodos ou artefactos embebidos.	Media
Diagrama de Obxectos	Un diagrama que presenta os obxectos e as súas relacións nun punto do tempo. Un diagrama de obxectos pódese considerar como un caso especial dun diagrama de clases ou un diagrama de comunicacións.	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 máis 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 relacións entre os actores e o suxeito (sistema), e os casos de uso.	Alta

Estes diagramas permítenos abarcar as perspectivas máis relevantes dun sistema:

⁹ *Introdución a UML 2.0.* http://www.epidataconsulting.com/tikiwiki/tiki-read_article.php?articleId=15#Breve_descripci_n_sobre_los_diagramas

- 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 obrigatorio que un modelo inclúa todos os diagramas.

Na seguinte figura¹⁰ indícanse as correspondencias máis evidentes entre diagramas mediante frechas, aínda que case todos eles teñen relación cos demais, 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.

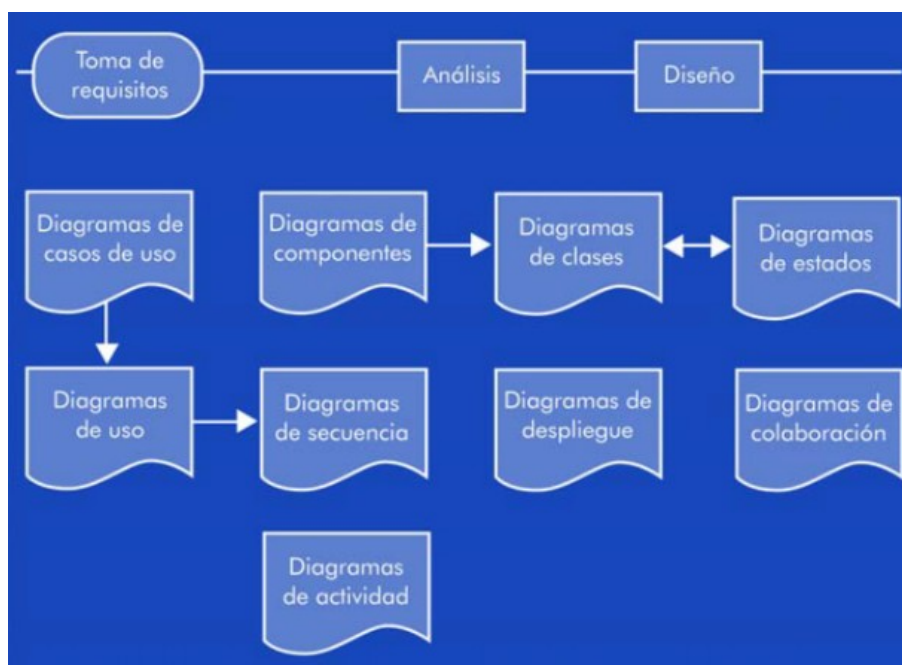


Fig. 8

A continuación veremos algúns exemplos¹¹ dos principais diagramas UML:

¹⁰ AYCART PÉREZ, David. GIBERT GINESTA, Marc HERNÁNDEZ MATÍAS, Martín, MAS HERNÁNDEZ, Jordi. *Ingeniería de software en entornos de SL*. Universitat Oberta de Catalunya.

¹¹ Os exemplos da Lavadora foron extraídos do libro “*Aprendiendo UML en 24 Horas*” de Joseph Schmuller.

- **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 máis 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 relacións. Por cada clase indícase o seu nome, atributos e métodos.

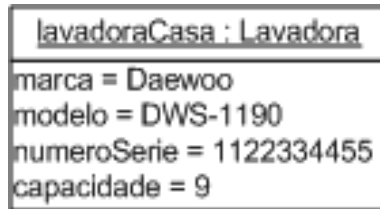
Exemplos de posibles diagramas da clase Lavadora:

Lavadora
-marca -modelo -numeroSerie -capacidade
+agregarRoupa() +agregarDeterxente() +activarLavado() +sacarRoupa()

 Lavadora
<i>Attributes</i> private String marca private String modelo private int numeroserie private int capacidade
<i>Operations</i> public void agregarRoupa() public void agregarDeterxente() public void activarLavado() public void sacarRoupa()

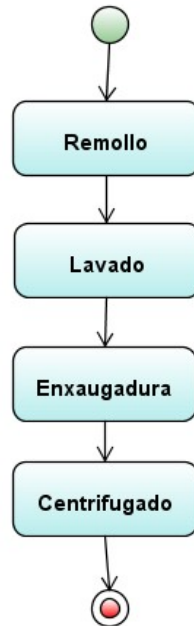
 Lavadora
<i>Attributes</i> private String marca private String modelo private int numeroserie private int capacidade
<i>Operations</i> public Lavadora() public String getMarca() public void setMarca(String val) public String getModelo() public void setModelo(String val) public int getNumeroserie() public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

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 correspóndese coa ocorrencia dun evento.

Por exemplo o diagrama de estados dun obxecto da clase Lavadora no proceso de

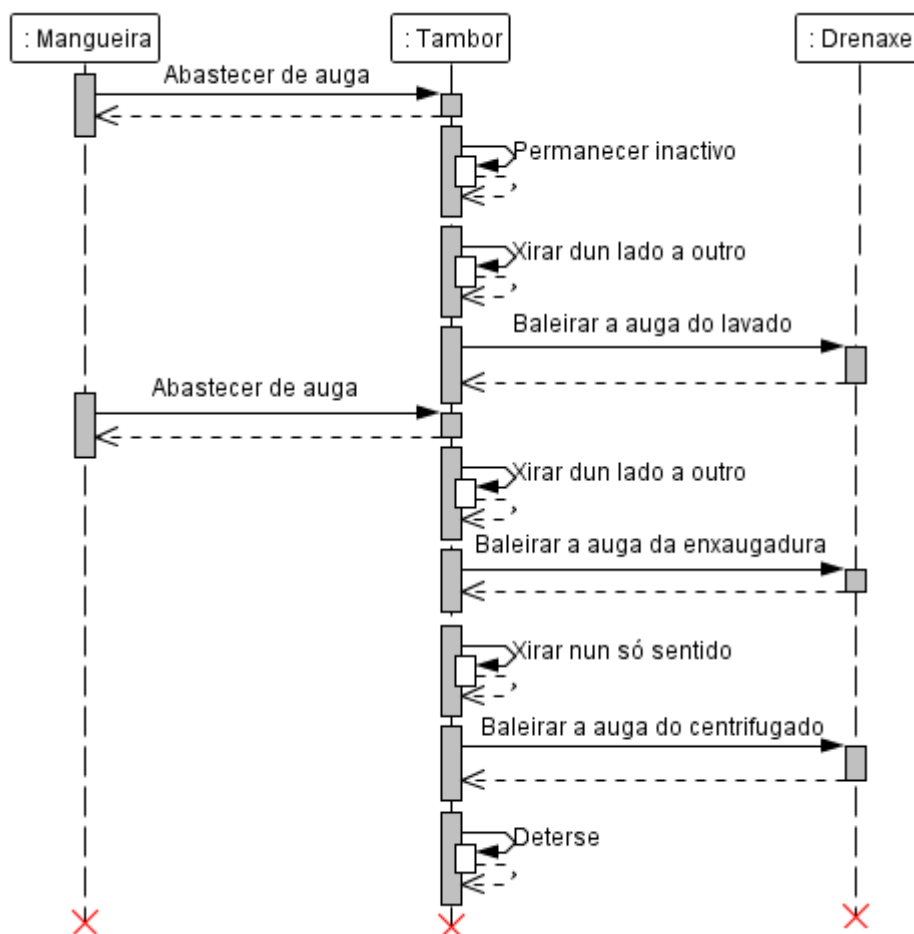


activar lavado podería ser:

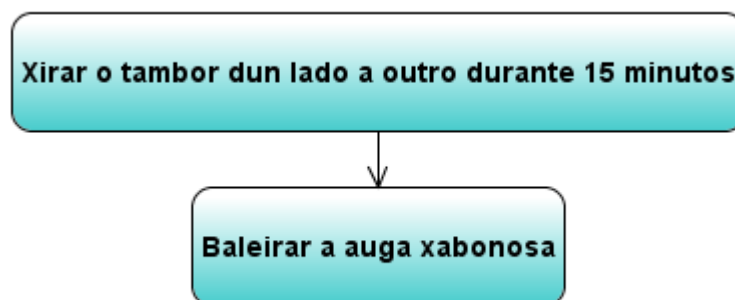
- **Diagrama de secuencias.** Representa a interacción entre obxectos ao longo do tempo. Por exemplo, o proceso de lavado de roupa despois de haber completado as operacións “agregar roupa”, ”agregar deterxente” e “activar” podería estar formado polas actividades representadas textualmente como:
 - A auga empeza a encher o Tambor mediante unha manguera.
 - A manguera 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 descríbese cun diagrama de secuencia.

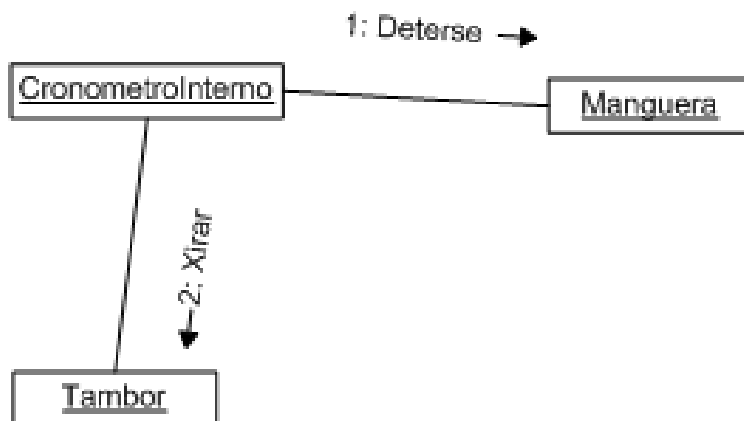


A representación das actividades para o proceso de lavado anterior podería ser:

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

Por exemplo a existencia dunha nova clase para o cronómetro interno que actúa

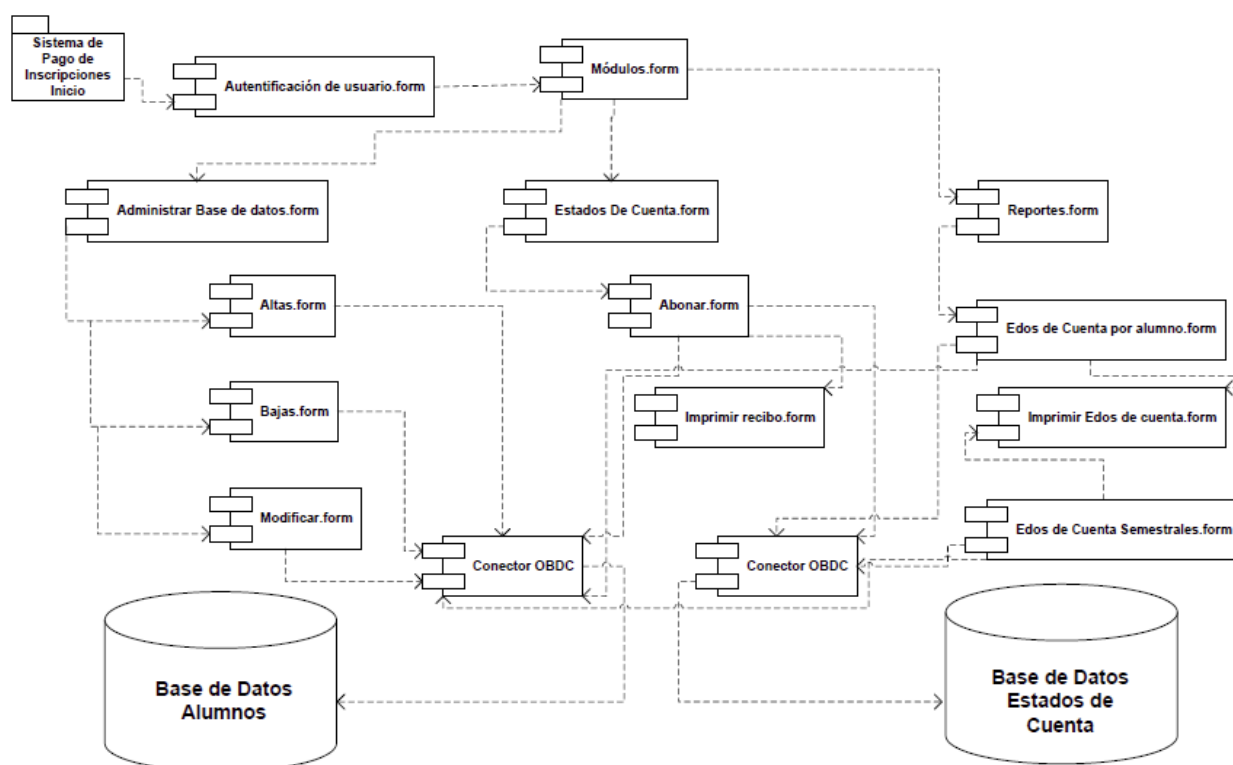
entre a manguera de auga e o tambor para controlar o tempo. O diagrama de colaboracións podería ser:



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

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

Exemplo¹²:



- **Diagrama de despregue.** O diagrama de despregue mostra a disposición física dos distintos nodos que compoñen o sistema e o reparto de compoñentes sobre eses nodos.

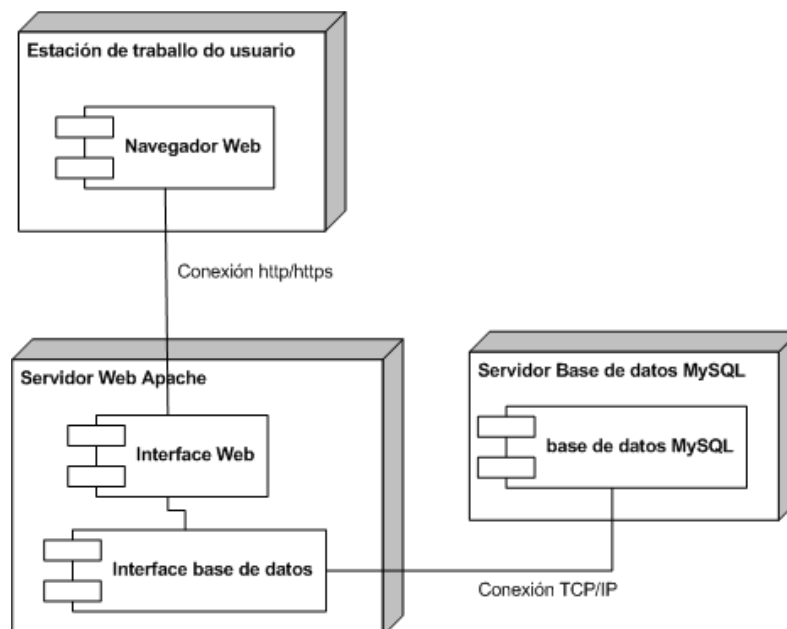
Enlazando o diagrama de despregue co de compoñentes:

- Os compoñentes son os elementos que participan na execución do sistema.

¹² <http://gzloluna8sm.blogspot.com.es/2010/06/diagrama-de-componentes.html>

- Os nodos son os elementos onde se executan os compoñentes e pode ser unha linguaxe de programación, un sistema operativo ou un ordenador.

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

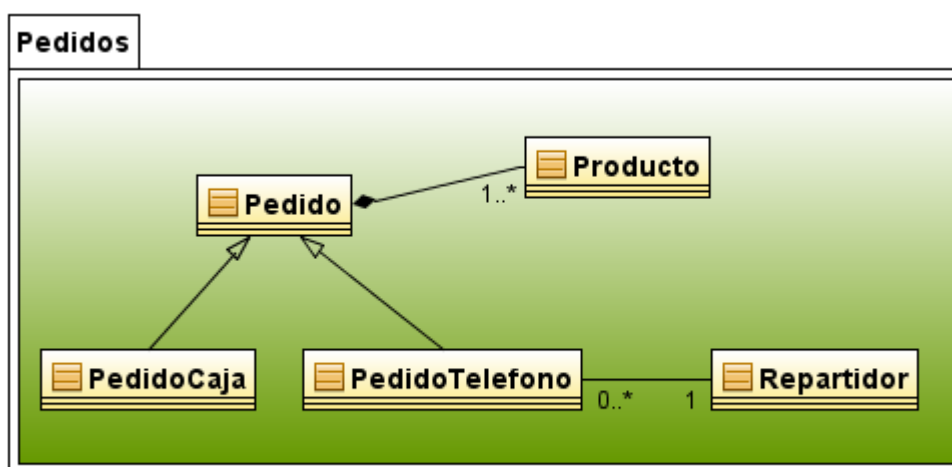


- **Paquetes.** Un paquete mostra agrupacións lóxicas de elementos de modelado. Determina un espazo de nomes e por tanto os nomes dos elementos dun paquete non poden repetirse.

Graficamente represéntase 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:



1.3 Tarefas

1.3.1 Tarefa 1. Identificación de elementos da programación orientada a obxectos (POO)

Nos seguintes exemplos de código identifica todos os conceptos e termos que **podas** dos vistos nos apuntamentos: clase, obxecto, construtor, atributos, métodos, encapsulado de campos, abstracción, encapsulamento, herdanza, polimorfismo, sobrecarga, envío de mensaxes, agregación/composición, xeneralización/especialización, asociación.

Por cada un dos que atopes copia e pega algún anaco de código ou ben explica o concepto utilizando os anacos de código involucrados.

■ Código 1

```
using System;
using System.Windows.Forms;
namespace ClasesAbstractas
{
    abstract class FiguraGeometrica
    {
        public abstract double Area();
    }
    class Circulo : FiguraGeometrica
    {
        private int radio;
        public int Radio
        {
            get
            {
                return radio;
            }
            set
            {
                radio = value;
            }
        }
        public Circulo(int r)
        {
            Radio = r;
        }
        public override double Area()
        {
            return (double)(Radio * Radio * 3.14);
        }
    }
    class Triangulo : FiguraGeometrica
    {
```

```

private int ladA;
private int ladB;
private int ladC;
public int LadoA
{
    get
    {
        return ladA;
    }
    set
    {
        ladA = value;
    }
}
public int LadoB
{
    get
    {
        return ladB;
    }
    set
    {
        ladB = value;
    }
}
public int LadoC
{
    get
    {
        return ladC;
    }
    set
    {
        ladC = value;
    }
}
public Triangulo(int a, int b, int c)
{
    LadoA = a;
    LadoB = b;
    LadoC = c;
}
public override double Area()
{
    double num, mPer;
    mPer = (double) (LadoA + LadoB + LadoC) / 2;
    num = mPer * (mPer - LadoA) * (mPer - LadoB) * (mPer - LadoC);
    if (num <= 0)

```

```

        {
            MessageBox.Show("posiblemente el triangulo no exista,
intentelo nuevamente");
            return 0;
        }
        else
            return Math.Pow(num, 0.5);
    }
}

class Cuadrado : FiguraGeometrica
{
    private int lado;
    public int Lado
    {
        get
        {
            return lado;
        }
        set
        {
            lado = value;
        }
    }

    public Cuadrado(int l)
    {
        Lado = l;
    }

    public override double Area()
    {
        return Lado * Lado;
    }
}
}

```

■ Código 2

```

class Clase1 {
    public void prueba()
    {
        System.out.println("\n Metodo sin Argumentos:()");
    }

    public void prueba(int x)
    {
        System.out.print("\n Metodo con un Argumento:");
        System.out.println(" x= " +x);
    }

    public void prueba(double x, double y, double z)
    {
        System.out.print("\n Metodo con tres Argumentos:");
    }
}

```

```
        System.out.println(" x= " +x+ "      y= " + y + "      z= " + z + "\n");
    }
}
class UsarClase1 {
    public static void main (String [] var)
    {
        Clase1 objeto= new Clase1();
        objeto.prueba();
        objeto.prueba(30);
        objeto.prueba(-3.5,20.0,18.6);
    }
}
```