INTRODUCCIÓN al Lenguaje de Modelado Unificado

1. ¿Qué es el UML?

El "Lenguaje de Modelado Unificado" – del inglés *Unified Modeling Language (UML)* – es un lenguaje basado en diagramas para la especificación, visualización, construcción y documentación de cualquier sistema complejo, aunque nosotros nos centraremos en el caso específico de *sistemas software*.

> Nota: otro de los ámbitos en los que UML se utiliza habitualmente es el modelado de los *procesos de negocio* de una organización. Por ejemplo, se puede hacer un modelo de cómo funciona (cómo desarrolla su labor diaria) el Departamento de Compras de una determinada empresa.

Por tanto, UML es un lenguaje para describir modelos. Básicamente, un modelo es una simplificación de la realidad que construimos para comprender mejor el sistema que queremos desarrollar. Un modelo proporciona los "planos" de un sistema, incluyendo tanto los que ofrecen una visión global del sistema como los más detallados de alguna de sus partes. Para comprender el objetivo del modelado con UML, es muy útil compararlo con otras áreas de ingeniería, como es la construcción de edificios o automóviles, con sus diferentes planos y vistas; o incluso con la industria cinematográfica, donde la técnica del *storyboarding* (representación de las secuencias de un película con viñetas dibujadas a mano) constituye un modelado del producto[1].

Si bien UML es independiente de las metodologías de análisis y diseño y de los lenguajes de programación que se utilicen en la construcción de los sistemas *software*, es importante destacar que se basa en el paradigma de la **orientación a objetos**. Por tanto, es especialmente adecuado cuando se ataca la construcción de sistemas *software* desde la perspectiva de la orientación a objetos.

La especificación, visualización, construcción y documentación de cualquier sistema software requiere que el sistema pueda ser estudiado desde diferentes puntos de vista, ya que un usuario final necesita una visión diferente del sistema de la que necesita un analista o un programador. UML incorpora toda una serie de diagramas y notaciones gráficas y textuales destinadas a mostrar el sistema desde las diferentes perspectivas, que pueden utilizarse en las diferentes fases del ciclo de desarrollo del software.

En este documento presentamos uno de estos puntos de vista, concretamente el conocido como "modelado estático", que se concreta en los *diagramas de clases*. Los diagramas de clases muestran para nosotros, pues, la estructura estática de un sistema software (en las diferentes fases de su construcción, por ejemplo, se suele hablar de diagramas de clases "de análisis", que posteriormente se convierten en diagramas de clases "de diseño"). Más concretamente, describen los elementos que en él existen (por ejemplo, clases), la estructura interna de estos elementos y cómo estos se interrelacionan entre sí. Es importante destacar que no examinaremos de manera exhaustiva todos los elementos que se pueden incorporar en un diagrama de clases, sino que únicamente estudiaremos aquellos que son de especial interés en esta asignatura.

Debemos tener claro también que un diagrama de clases UML (esto es aplicable a cualquier tipo de diagrama UML) es tan sólo una *vista* del modelo estático del sistema en cuestión, esto quiere decir que una misma clase puede aparecer en varios diagramas y que podemos crear más de un diagrama, bien si es el número de clases es elevado o bien si queremos mostrar dos o más partes del sistema que tienen poco que ver en diagramas separados. Hay que tener en cuenta que los diagramas son herramientas de comunicación con otras personas que quizá mañana vean nuestro trabajo, y lo importante es que la información sea legible a la vez que completa.

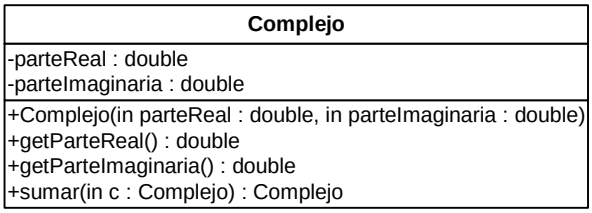
El término "estructura estática" está relacionado con el hecho de que los diagramas de clases muestran todas las relaciones y datos que el sistema necesita durante su operación. Esta vista se complementaría con la "estructura dinámica", que muestra las relaciones entre los elementos teniendo en cuenta la dimensión temporal, que no tratamos en este documento.

[1] Recomendamos la lectura del primer capítulo "¿Por qué modelamos?" del libro (Booch 1999).

2. Clases y Objetos

2.1. Clases, atributos y operaciones

Una clase es una descripción de un conjunto de objetos que comparten la misma estructura y semántica y que presentan el mismo comportamiento y relaciones. En UML se presentan mediante un rectángulo con tres compartimentos. En el primero de estos compartimentos se indica cual es el nombre de la clase, en el segundo se indican cuáles son las propiedades de la clase (*atributos* en terminología UML), mientras que en el último se indica el comportamiento de la clase, formado por una colección de servicios (*operaciones* en terminología UML y métodos en algunos lenguajes de programación como Java). Por ejemplo, el siguiente diagrama muestra una clase *Complejo*, con dos atributos, un constructor y algunas operaciones para la manipulación sencilla de números complejos.



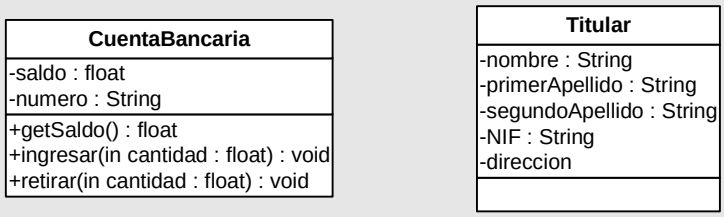
En relación con el nombre de la clase, es importante destacar las siguientes recomendaciones (convenciones de nombrado):

- El nombre de una clase debería ser un sustantivo en singular y debería comenzar con mayúsculas.
- El nombre de una clase debe de estar de acuerdo a su semántica o intención, en otras palabras, debe ser significativo a la vez que simple. De no ser así, se dificultará la posible reutilización posterior de esa clase.

Ejemplo. Cuentas Bancarias: Entidades Fundamentales.

Por ejemplo, adentrándonos en el dominio bancario, podemos pensar en cuentas bancarias y sus titulares como entidades importantes. Simplificando las cosas, podemos pensar que una cuenta bancaria es un objeto que guarda el valor del saldo y que del titular de la cuenta lo único que nos interesa almacenar es su nombre y su dirección postal.

El diagrama resultante para representar las dos entidades sería el siguiente:



Nótese que hemos dejado la clase Titular parcialmente especificada, no aparece el tipo del atributo dirección, y tampoco las operaciones sobre la clase. Tampoco hemos especificado los constructores de las clases. Posteriormente se podrán añadir al diagrama o pueden dejarse a la elección del programador (quizá encontremos una clase Dirección para manipular direcciones postales entre nuestras bibliotecas de clases Java que podamos reutilizar).

Las propiedades de las clases se representan mediante una lista de atributos, que figuran en el segundo compartimento. Para cada atributo se pueden indicar, entre otras características, su nombre, su tipo, y su visibilidad.

El nombre de cada atributo debe ser también significativo, pero a diferencia de los nombres de clase, suelen comenzar en minúscula. El tipo de cada atributo puede ser cualquier clase o bien puede ser un tipo de datos básico. El nombre de cada atributo y su tipo quedan separados mediante dos puntos (':'). Finalmente, la visibilidad de cada atributo nos indica hasta que punto las operaciones de otras clases pueden acceder al atributo. Tenemos tres posibilidades:

- Público: el atributo puede ser accedido desde otras clases. Para indicar en UML que un atributo es de acceso público, es necesario anteponer al nombre del atributo el símbolo '+'. Cabe recordar que en orientación a objetos los atributos no suelen definirse como públicos (por el principio de ocultación de la información), y si es necesario que sean accesibles se definen operaciones de acceso, conocidas como *getters* (como las operaciones getParteReal y getParteImaginaria del ejemplo anterior) y *setters* (como podrían ser operaciones setParteReal y setParteImaginaria que podríamos añadir en el ejemplo anterior).
- Protegido: el atributo puede ser accedido desde las clases descendientes. Para indicar en UML que un atributo es de acceso protegido, es necesario anteponer el símbolo '#' al nombre del atributo.
- Privado: el atributo no puede ser accedido desde otras clases. Para indicar en UML que un atributo es de acceso privado, es necesario anteponer al nombre del atributo el símbolo '-'.

Finalmente, en el tercer compartimento se indica el comportamiento de la clase, que queda representado mediante una lista de operaciones. Para cada operación es necesario especificar su signatura, que tiene la siguiente sintaxis:

```
visibilidad nombreMetodo (Parámetro, Parámetro, ...)[: tipo de retorno]
```

Tal y como figura en la especificación de la signatura, hay que indicar la visibilidad de las operaciones de la clase, anteponiendo los símbolos '+' (operación pública), '#' (operación protegida) o '-' (operación privada) al nombre de la operación. La semántica de estos símbolos es idéntica a la de los atributos.

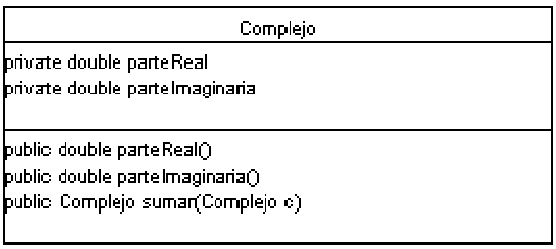
Los parámetros (se pueden especificar cero, uno o más) responde a la sintaxis:

```
{[dirección] nombre: tipo [=valor por defecto]}
```

donde la dirección puede tomar uno de estos valores:

- `in`, para especificar que se trata de un parámetro de entrada que no se puede modificar.
- `out`, para especificar los parámetros de salida, que pueden modificarse para comunicar información al invocador.
- `inout`, que especifica un parámetro de entrada que puede modificarse.

Nótese que UML permite adaptar las notaciones gráficas anteriores a lenguajes de programación concretos, por lo cual podríamos también dibujar la clase anterior cambiando la notación de atributos y operaciones por la sintaxis propia de Java, como se muestra en la siguiente figura:



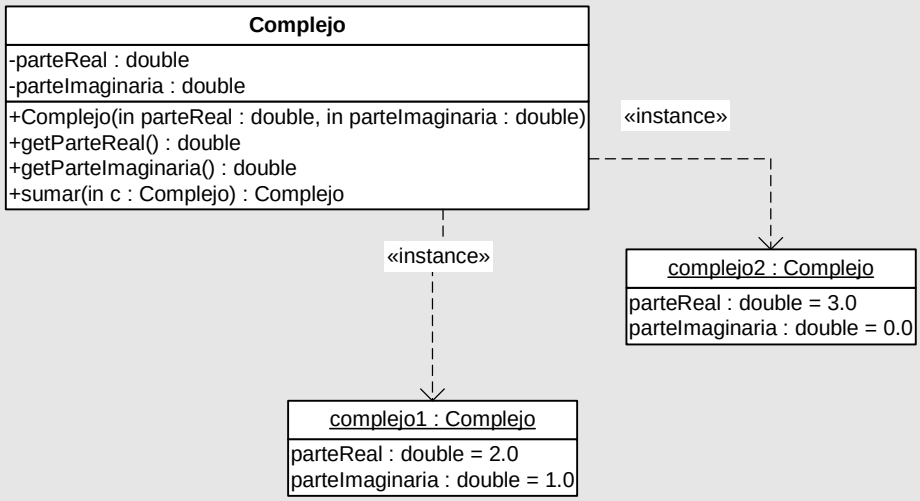
También es posible ocultar alguno de los compartimentos o alguno de los detalles de los atributos u operaciones si creemos que no añaden información importante en el diagrama o bien están visibles en otro diagrama, en el caso de que una clase aparezca en varios de ellos.

Cuadro 1. Cómo representar instancias de las clases en UML.

Otro tipo de diagrama UML que muestra la estructura estática del sistema son los *diagramas de objetos*. Un diagrama de objetos muestra un grafo de objetos (instancias de las clases) con valores concretos para sus atributos y enlaces entre objetos (instancias de las relaciones de asociación).

Por ejemplo, el siguiente diagrama de clases muestra dos instancias de la clase *Complejo*, de nombre *complejo1* y *complejo2* (en UML, el nombre aparece subrayado en el primer compartimento), con valores

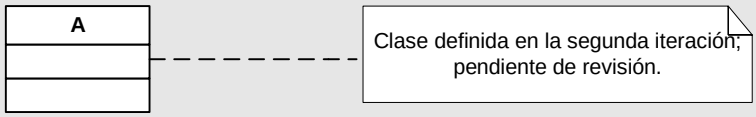
concretos para sus atributos. Puede representarse la relación con su clase mediante dependencias etiquetadas con <<instance>> o <<instanceOf>> aunque es redundante con la especificación del nombre, que ya incluye la clase a la que pertenecen.



Estas representaciones no suelen ser muy utilizadas por sí solas. Suelen encontrarse en otros diagramas UML como los diagramas de secuencia o de clases. Se considera que un diagrama de objetos es simplemente un diagrama de clases en el que sólo aparecen objetos.

Cuadro 2. Notas en UML

El lenguaje UML permite incluir *notas* con información textual asociadas a cualquier elemento de cualquier diagrama (clases, operaciones, atributos, etc.) para cualquier uso: desde escribir el pseudocódigo de una operación o una restricción del elemento hasta una referencia bibliográfica, pasando por notas sobre el estado de la clase, por ejemplo, indicando que es provisional y hay que revisarla. La notación es la que se muestra en la siguiente figura.



2.2. Implementación de una clase UML sin relaciones en Java

La implementación de una clase sin relaciones en Java es un proceso bastante directo; de hecho, prácticamente todas las herramientas de modelado UML proporcionan la opción de *generar el código automáticamente* a partir del diagrama de clases, dejando al programador la tarea de escribir el cuerpo (código) de los métodos Java que resultan de las operaciones. Por ejemplo, la clase *Complejo* anteriormente definida se traduciría a la siguiente definición:

```
public class Complejo {

  // Atributos
  private double parteReal;
  private double parteImaginaria;

  // Operaciones
  public double parteReal() {
  }
  public double parteImaginaria() {
  }
```

```
  public Complejo sumar(Complejo c) {
  }
}
```

Merece la pena resaltar los siguientes aspectos del esqueleto de código anterior:

- El modificador `public` aplicado a la clase indica que la clase está disponible a las demás clases sin restricciones (si no ponemos `public`, se restringirá la disponibilidad de las clases a otras del mismo paquete).
- Las declaraciones de atributos y métodos pueden aparecer en cualquier orden, aunque es recomendable que siempre se siga una misma convención.
- En ocasiones los diagramas de clases no contienen todos los detalles, por ejemplo, se puede omitir el/los constructor/es, e incluso el tipo de los atributos. En general, hay "decisiones" que se pueden posponer a la fase de codificación. Dependiendo del nivel de detalle elegido, el programador que escribe la clase Java tendrá más o menos trabajo a la hora de realizar la especificación de la clase.
- El código anterior debe ser completado con:
  - La implementación de los métodos.
  - Los comentarios `javadoc` necesarios para la posterior generación automática de la documentación (aunque en algunos de los ejemplos de este documento no los incluyamos para abreviar, es muy importante incluirlos). Hay herramientas UML que permiten incluir estos comentarios en el propio diagrama, de modo que se pasan directamente al código generado de manera automática.

La clase quedaría finalmente como muestra el siguiente fragmento de código, tras añadir algunos comentarios `javadoc` e incluir algún método adicional útil (el método `toString`, sobrescrito del que tiene la clase `Object` de Java, que convierte el complejo a cadena).

```
/**
 * La clase Complejo permite crear y manipular instancias que representan
 * números complejos.
 * @author Miguel Angel Sicilia
 */

 public class Complejo {

  /** Parte real del número complejo */
  private double parteReal;

  /** Parte imaginaria del número complejo */
  private double parteImaginaria;

  /**
   * Constructor de un número complejo a partir de dos reales.
   */
  public Complejo(double parteReal, double parteImaginaria){
       this.parteReal       = parteReal;
       this.parteImaginaria = parteImaginaria;
  }

  /**
   * Getter para la parte real del complejo.
   */
  public double parteReal() {
       return parteReal;
  }

  /**
   * Getter para la parte imaginaria del complejo.
   */
  public double parteImaginaria() {
```

```
        return parteImaginaria;
    }

    /**
     * Devuelve un nuevo objeto de la clase <code>Complejo</code>
     * con el valor de sumar el complejo c con el complejo que
     * recibe este mensaje.
     */
    public Complejo sumar(Complejo c) {
        double nuevaParteReal = this.parteReal() + c.parteReal();
        double nuevaParteImaginaria = this.parteImaginaria()
                                      + c.parteImaginaria();
        return new Complejo (nuevaParteReal, nuevaParteImaginaria);
    }

    /**
     * Convierte el complejo a cadena de caracteres con la notación
     * habitual (a + bi).
     */
    public String toString(){
        return "(" + parteReal + "+" + parteImaginaria + "i)";
    }
}
```

Ejemplo. Cuentas Bancarias: implementación en Java

Siguiendo el ejemplo anterior, la clase *CuentaBancaria* se implementaría en Java de la siguiente manera:

```
/**
 * La clase CuentaBancaria representa cuentas bancarias en las
 * que se mantiene un saldo.
 * @author Miguel Angel Sicilia
 */

public class CuentaBancaria {

    /** Número de cuenta*/
    private String numero;

    /** Saldo en euros de la cuenta*/
    protected float saldo;

    /**
     * Constructor de una cuenta bancaria.
     * @param numero numero identificativo de cuenta
     * @param saldoInicial saldo inicial de la cuenta
     */
    public CuentaBancaria(String numero, float saldoInicial){
        this.numero = numero;
        saldo = saldoInicial;
    }

    /**
     * Getter para el saldo de la cuenta.
     */
    public float getSaldo() {
        return saldo;
    }

    /**
     * Getter para el numero de la cuenta.
     */
    public String getNumero(){
        return numero;
    }

    /**
     * Aumenta el saldo en la cantidad indicada.
```

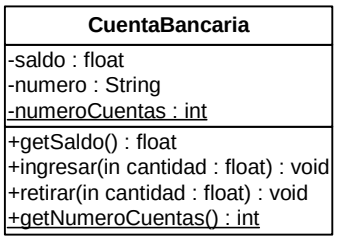
```
   * @param cantidad la cantidad a ingresar
   */
  public void ingresar(float cantidad){
       saldo += cantidad;
  }

  /**
   * Disminuye el saldo en la cantidad indicada.
   * @param cantidad la cantidad a retirar
   */
  public void retirar(float cantidad)
                      throws SaldoResultanteNoPermitido{
       saldo -= cantidad;
  }
}
```

Nótese que hemos definido el atributo saldo `protected` al darnos cuenta que posibles futuras subclases podrían requerir su manipulación. También hemos declarado que el método *retirar* puede lanzar una excepción definida por nosotros. Aunque en el código de esta clase no lo hace, posiblemente lo hará el código de subclases que por ejemplo, no permitan que una cuenta de un tipo determinado quede en "números rojos".

2.3. Atributos y operaciones de las clases

UML proporciona una notación especial para distinguir los miembros (atributos y operaciones) de las instancias y de los miembros de las clases. Hasta ahora, solo hemos utilizado propiedades de las instancias, pero vamos a suponer ahora que quisiéramos añadir un atributo de la clase a *CuentaBancaria* con el número de cuentas que se han creado, y un método de la clase para tener acceso a ese atributo.

En UML, se subrayan los atributos y operaciones de la clase para diferenciarlos de los de las instancias, como se muestra en la siguiente figura:



Ejemplo: Cuentas Bancarias: Atributos y Operaciones de clase

Los atributos y operaciones de la clase pueden considerarse como atributos y operaciones de una "instancia de una clase especial" que está visible desde todas las instancias de la clase (pero no a la inversa) – véase el acceso desde código del constructor al atributo de clase `numeroCuentas`. Esto es especialmente apropiado en Java, donde tenemos una clase `java.lang.Class` cuyas instancias representan a las clases cuyas instancias se usan en la aplicación. A esa "instancia especial" es a la que se accede cuando hagamos la invocación sobre el nombre de la clase `CuentaBancaria.getNumeroCuentas()`.

```
public class CuentaBancaria {

  public CuentaBancaria(String numero, float saldoInicial){
       this.numero = numero;
       saldo = saldoInicial;
       numeroCuentas ++;
  }
  private static int numeroCuentas = 0;
  static{
      numeroCuentas = 0;
  }

  public static int getNumeroCuentas(){
       return numeroCuentas;
```

```
  }

// resto de atributos y métodos de instancia (vistos anteriormente)
}
```

3. Relaciones entre clases

Una relación entre clases se puede definir como una conexión entre dos o más clases. En orientación a objetos, las principales relaciones que se pueden establecer entre clases son las de generalización (o herencia), asociación, agregación y dependencia (o uso).

3.1. Asociaciones y dependencias

Las asociaciones representan relaciones estructurales entre clases. En UML, se define una asociación como una relación que describe un conjunto de enlaces, donde cada enlace define una interconexión semántica entre instancias de las clases que participan en la asociación. Se dice que las instancias de una de las clases "conocen" a las instancias de la otra clase a las que están enlazadas.

A grandes rasgos, podemos distinguir dos tipos de asociaciones:

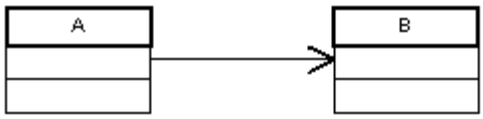
- Asociaciones binarias: en este caso, el vínculo se establece entre dos clases.
- Asociaciones n-arias: en este caso, el vínculo se establece entre tres o más clases.

3.1.1. Asociaciones binarias

En UML, las asociaciones binarias se representan mediante una línea continua que conectan dos clases (aunque puede darse el caso particular en que las dos clases involucradas en la asociación binaria sean la misma. En este caso, hablamos de asociaciones reflexivas).

A la representación de las asociaciones se le pueden añadir ciertos elementos que aportan información adicional, como la direccionalidad, los nombres de rol o la cardinalidad, que veremos a continuación.

Una asociación binaria representa una interconexión entre las instancias (u objetos) de dos clases A y B. Esta interconexión puede ser o no bidireccional. En caso que la interconexión sea **bidireccional**, (opción por defecto) será posible navegar desde las instancias de la clase A a las instancias de la clase B, y a la inversa, es decir, desde las instancias de la clase B podremos acceder a las instancias de la clase A. Si decidimos que la interconexión debe ser **unidireccional**, entonces será necesario añadir una flecha abierta encima de la línea de asociación; esta flecha indica el sentido de la navegación.



Las asociaciones pueden tener nombre. Dado que por defecto las asociaciones son bidireccionales, se pueden especificar dos nombres en la asociación. En UML, estos nombres son los **roles** que juegan las clases en la asociación.

Ejemplo: Cuentes Bancarias: Roles

Volvamos al ejemplo anterior de las cuentas bancarias. Según el modelo que tenemos hasta ahora, hemos modelado cuentas y usuarios pero no qué cuentas pertenecen a qué usuarios. Esta última información se representa mediante una asociación entre ambas clases.

En la figura anterior hemos omitido los compartimentos de atributos y operaciones y hemos incluido una asociación con los roles en ambos extremos, cambiando el nombre de la clase *Titular* por otro más significativo, ya que un *Cliente* del banco juega el rol de *titular* en su relación con las *cuentas* que posee. Los roles no son obligatorios, pero en ocasiones alguno de ellos aporta claridad a la asociación.

Nótese que en la figura anterior la asociación es bidireccional y que no hemos dicho nada sobre qué número de instancias participan, por ejemplo, si una cuenta puede tener varios titulares.

Dado que una asociación binaria representa una interconexión de instancias de dos clases, es necesario indicar cuántos objetos de cada clase pueden estar involucrados en dicha interconexión. En otras palabras, es necesario indicar la **cardinalidad** (o multiplicidad) de la asociación. Podemos indicar el valor máximo y mínimo de esta cardinalidad. Dadas dos clases A y B, la cardinalidad indica con cuántas instancias de B se puede relacionar cada instancia de A (valor mínimo y máximo). Los casos más generales de cardinalidad son los siguientes:

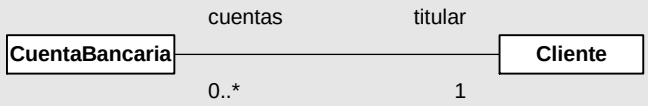
- Cardinalidad "*uno a uno*": en este caso una instancia de la clase A se relaciona con una única instancia de la clase B, y cada instancia de la clase B se relaciona con una única instancia de la clase A.
- Cardinalidad "*uno a muchos*": en este caso una instancia de la clase A se relaciona con varias instancias de la clase B, y cada instancia de la clase B se relaciona con una única instancia de la clase A.
- Cardinalidad "*muchos a muchos*": en este caso una instancia de la clase A se relaciona con varias instancias de la clase B, y cada instancia de la clase B se relaciona con varias instancias de la clase A.

Ejemplo. Cuentas Bancarias: Cardinalidad de las asociaciones

Pensemos en la cardinalidad de la asociación en nuestro ejemplo. En el caso de las cuentas bancarias, podemos pensar las tres situaciones siguientes, que se corresponden a los tres casos generales anteriores:

- Una persona, de tener alguna, sólo puede tener una cuenta, y no se permiten cuentas de titularidad múltiple. Diremos que una cuenta tiene un único titular – nótese que no se puede tener cuentas sin titular – y que un cliente puede tener una cuenta o ninguna (quizá es sólo cliente de servicios).

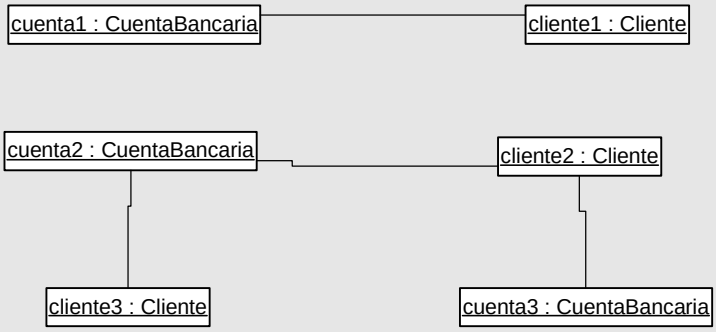
- Una persona puede tener varias cuentas, pero no se permiten cuentas de titularidad múltiple. En este caso, un cliente puede tener una, varias (indicado con el asterisco '*') o ninguna cuenta, pero las cuentas tienen un único titular.



- Una persona puede tener varias cuentas, y éstas pueden ser de titularidad múltiple. Diremos que una cuenta puede tener de uno a muchos clientes como titulares y que un cliente puede tener una cuenta, varias o ninguna.

Aunque sólo la última situación es la que sucede en la realidad del entorno bancario, contemplamos las dos anteriores por motivos didácticos.

A veces es útil considerar los enlaces entre las instancias de las clases que participan en la asociación. Esto puede visualizarse mediante diagramas de objetos que muestren una situación de ejemplo. Por ejemplo, el siguiente diagrama muestra enlaces entre tres clientes y tres cuentas bancarias.



En el diagrama de objetos anterior ¿a cuál de los tres casos anteriores de asociación se corresponde? Si nos fijamos en él, hay cuentas con más de un titular (*cuenta2*) y clientes con más de una cuenta (*cliente2*), por lo cual, el único diagrama que se ajusta es la situación de "muchos a muchos".

A continuación se muestra una tabla resumen de la posible multiplicidad de las cardinalidades y sus notaciones:

NOTACIÓN	DESCRIPCIÓN	EJEMPLO
`1`	Exactamente uno	Una persona tiene un único país de origen
`0..1`	Cero o uno	Un empleado tiene un responsable directo

		en una empresa excepto si es un director, en cuyo caso tiene cero.
`0..*`	Cero o más	Una persona puede no tener casa (vivir alquilado), tener una o tener más de una.
`1..*`	Uno o más	Un alumno de un centro está matriculado de una o más asignaturas (si no tuviera ninguna no sería alumno del centro).
`número exacto`	El número exacto indicado	Un segmento tiene 2 puntos asociados: el origen y el destino del segmento.

Se pueden establecer combinaciones de lo anterior, llegándose a dar multiplicidades como: 0..1, 4..6, 8..*, que significaría "cualquier número de objetos exceptuando el 2, el 3 y el 7".

3.1.2. Implementación de asociaciones en Java

La técnica esencialmente consiste en indicar la multiplicidad de la cardinalidad para cada sentido de una asociación navegable (en el caso anterior, para los sentidos *CuentaBancaria* → *Cliente* y *Cliente* → *CuentaBancaria*) de la siguiente forma:

- Las cardinalidades de tipo "muchos" se deben implementar mediante una estructura de datos que será un atributo cuyo tipo es la clase Java que está "al otro extremo".
- Las cardinalidades de tipo "uno" se deben implementar mediante una referencia de objeto cuyo tipo es la clase que participa de la asociación con cardinalidad uno, que será un atributo de la clase Java "que está en el otro extremo".

Lógicamente, también habrá que añadir los métodos pertinentes para la *gestión* de la asociación, es decir, añadir y eliminar enlaces.

Ejemplo. Cuentas Bancarias: Implementación de asociaciones

Tomemos la segunda situación de nuestro ejemplo ("uno a muchos") para ver cómo se debería implementar la asociación en Java.

```
public class CuentaBancaria{

    private Cliente titular;

    // resto de atributos...

    public CuentaBancaria (Cliente titular){
          if (titular == null)
             throw new Exception("No se puede crear cuentas sin titular");
          this.titular = titular;
         // resto de código del constructor
    }

    public Cliente obtenerTitular(){
         return titular;
    }

    public boolean cambiarTitular(Cliente c){
        // No debemos permitir que una cuenta se quede sin titular.
         if (c!=null) {
             titular = c;
         }
        return c!=null;
    }

    // resto de operaciones
}


import java.util.Set;
import java.util.HashSet;
import java.util.Iterator;

public class Cliente{
```

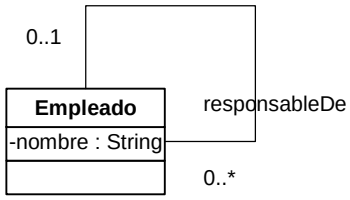
```
     private Set cuentas;

     public Cliente(...){
        cuentas = new HashSet();
        // resto de código del constructor
     }

    public void nuevaCuenta(CuentaBancaria c){
        cuentas.add(c);
    }

    public void eliminarCuenta(CuentaBancaria c){
        cuentas.remove(c);
    }

    public Iterator obtenerCuentas(){
        return cuentas.iterator();
    }

    // resto de operaciones y atributos...
}
```

En la implementación anterior se han tomado algunas decisiones de implementación por parte del programador:

- Una cuenta nunca debe quedar sin titular, esta regla del negocio se asegura en los métodos que manipulan la implementación de la asociación, incluyendo el constructor que obliga a especificar un titular válido.
- Hemos elegido una estructura `Set` (conjunto) de entre las que nos ofrece Java, ya que las cuentas de un titular ni requieren orden ni pueden estar duplicadas. En general, siempre que se implementan asociaciones se deben tomar decisiones de este tipo.
- Normalmente, los atributos que implementan la asociación toman como nombre el nombre del rol que aparecía en el diagrama, si es que había alguno.
- En implementaciones de asociaciones *"a muchos"*, como la de cliente con sus cuentas, hay que estudiar cómo damos acceso a buscar enlaces dentro de la asociación. En el ejemplo anterior, ofrecemos el método `obtenerCuentas` que devuelve un iterador con el cual se puede manipular las cuentas mediante bucles (véanse las clases correspondientes en `java.util`).

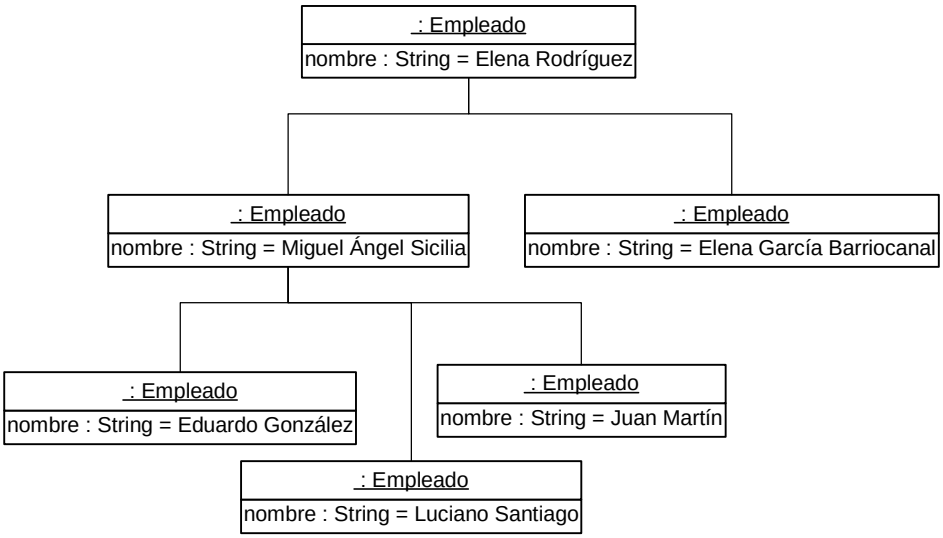
3.1.3. Asociaciones binarias reflexivas

Las asociaciones reflexivas son un tipo de asociación binaria en el cual la clase de origen y destino de la asociación es la misma. Como ejemplo típico, trataremos la organización de una empresa, definida mediante la relación "es responsable de" entre instancias de la clase Empleado.



En el ejemplo anterior hemos dibujado una asociación bidireccional donde un empleado puede tener cero, uno o muchos empleados bajo su responsabilidad, mientras que todos los empleados tienen un único responsable (excepto el Director General, que no tiene ningún responsable).

Es importante "visualizar" la estructura de instancias que se desprenden de este diagrama. Esta estructura, en nuestro caso particular, representará un árbol de instancias, por la restricción de que una persona tiene un solo responsable como mucho.



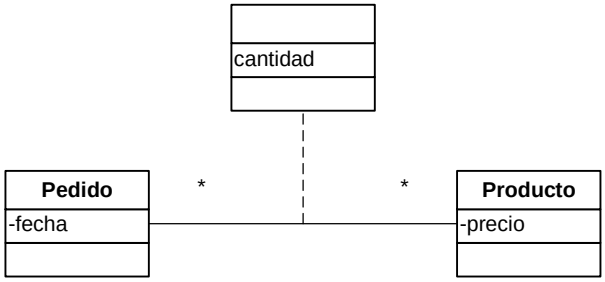
Nótese que si la asociación reflexiva es de cardinalidad "muchos" en ambas direcciones, representa potencialmente un grafo de instancias. Dejamos al lector que compruebe este hecho trazando un diagrama de objetos de ejemplo.

Las técnicas de implementación en Java son las mismas que en las asociaciones no reflexivas.

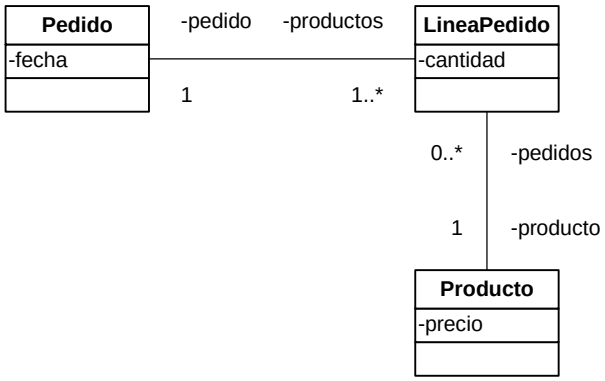
Nótese que en general las asociaciones, incluyendo las reflexivas, pueden formar grafos dirigidos con ciclos de instancias (imaginemos por ejemplo un sistema que quisiese guardar las relaciones "conoce a" entre personas), aunque en casos como el ejemplo anterior la implementación debe impedirlo para garantizar que la semántica del dominio del problema se cumple.

3.1.4. El concepto de Clase de la Asociación

En una asociación entre dos clases, es posible que la propia asociación tenga atributos. La información que tienen los valores de estos atributos no tiene sentido propio excepto en el concepto del enlace. Un ejemplo típico es el modelado de un pedido de múltiples productos. La cantidad pedida de cada producto no pertenece ni al pedido en sí (a diferencia, por ejemplo, de la fecha del pedido) ni al producto, sino a la relación del pedido con cada uno de los productos solicitados en él. En UML, este ejemplo se modelaría como una clase de la asociación, que se conecta con una línea discontinua a la asociación de la cual depende.

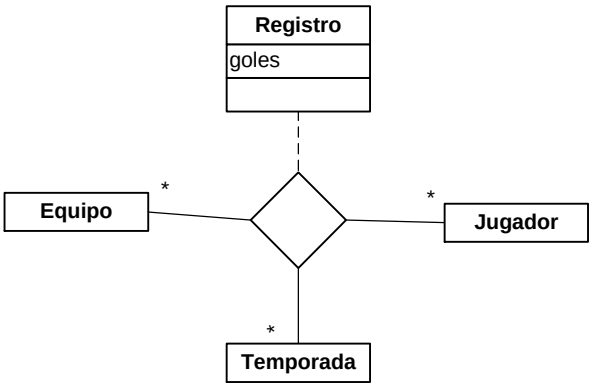


Los lenguajes de programación no proporcionan un mecanismo directo para implementar estas asociaciones, sino que se implementan creando una tercera clase que posee dos asociaciones normales, como se muestra en la siguiente figura.



3.1.5. Asociaciones n-arias

Una asociación n-aria constituye una relación que se establece entre tres o más clases. Un ejemplo típico es el de la cuenta de los goles marcados por jugadores de los distintos equipos en el campeonato de Liga. En este caso, hay que enlazar a un jugador con un equipo en una temporada, ya que en diferentes temporadas puede estar en diferentes equipos (incluso quizá en la misma). Estas relaciones se representan en UML mediante un rombo, y pueden tener una clase de la asociación con la información que depende de la existencia del propio enlace, en nuestro caso, una clase Registro con los datos del jugador en el equipo y temporada indicado en el enlace.



Dado un jugador concreto que juega en un equipo concreto tendrá varios registros de goles en cada temporada. Dada una temporada concreta y un equipo concreto, habrá un registro de goles para cada jugador del equipo. Finalmente, dada una temporada y jugador concreto, puede meter goles en diversos equipos (asumimos que un jugador dentro de una misma temporada puede cambiar de equipo).

A la hora de implementar en Java, se suele introducir una clase adicional y se expresan los enlaces n-arios mediante enlaces binarios.

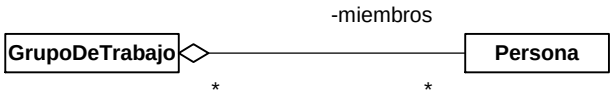
3.1.6. Relaciones de Agregación y Composición

Las relaciones de agregación y composición son tipos especiales de asociación. En las asociaciones binarias, se asume que las instancias de ambas clases son independientes, en el sentido de que no dependen de la existencia de las instancias de la otra clase. Sin embargo, las relaciones de **agregación** asumen una subordinación conceptual del tipo “todo/parte”, o bien “tiene un”.

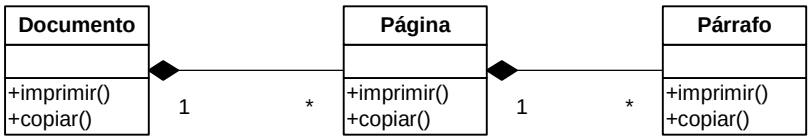
A efectos de implementación en Java, las técnicas son las mismas, con la salvedad de que no se pueden crean ciclos de enlaces, es decir, una “parte” no puede estar agregada en sí misma, ni directa ni indirectamente. Se puede eliminar la instancia que representa al “todo” y seguir existiendo las instancias que eran sus “partes”.

Las agregaciones se representan mediante un rombo en la parte del objeto agregado, para distinguirlo de los objetos “más pequeños” que contiene. Por ejemplo, puede considerarse que un Grupo de Trabajo es un agregado de sus miembros, lo cual no implica que cuando el grupo se disuelva, las instancias que

representan a sus miembros deban eliminarse también. Nótese que las estructuras de objetos resultantes son en general un grafo dirigido acíclico.



La **composición** es un tipo de agregación más específico en el cual se requiere que una instancia de la clase que representa a las partes esté asociada como mucho con una de las instancias que representan el "todo" (nótese que no puede haber multiplicidad "muchos" en el extremo del agregado). De esta manera, el objeto compuesto gestiona sus partes en exclusiva. Por lo general, la estructura de objetos resultantes es siempre un árbol. Un ejemplo típico es un procesador de textos que estructure los documentos en páginas, y éstas en párrafos. La notación en UML es utilizar un rombo relleno en el lado de la clase que representa al "todo".



Otro ejemplo clarificador es el de un Directorio y las entradas de directorio que contiene (ficheros, otros directorios), que se deben eliminar cuando se elimina el directorio. Por otro lado, un fichero está en un solo directorio (aunque los enlaces simbólicos puedan hacer parecer que esto no se cumple, internamente es así en los sistemas operativos). Invitamos al lector a realizar el modelo como ejercicio.

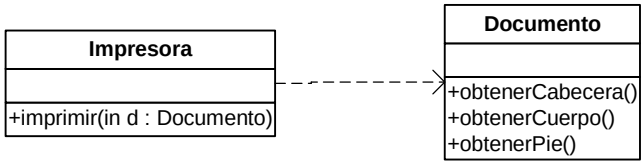
En Java, las composiciones se implementan con las mismas técnicas que las asociaciones, con la salvedad de que en la composición la gestión de la memoria de las "partes" debe obligatoriamente ser responsabilidad de la clase agregada. En Java, esto significa que el documento guardará referencias a sus páginas *en exclusiva*, es decir, no las cederá a otros objetos, y deberá proveer mecanismos para la creación y el borrado de páginas. Nótese que en Java el borrado solo requiere "perder" la referencia, ya que si borramos una página de este modo, el recolector de basura la eliminará y a su vez eliminará las instancias de los párrafos contenidos en la página, puesto que ninguna otra instancia podía contener referencias a esos párrafos. Por ello, se dice que los *ciclos de vida* de las "partes" están supeditados al ciclo de vida del "todo".

Las composiciones implican la propagación a las "partes" de las invocaciones a las operaciones sobre el "todo". Siguiendo el ejemplo anterior, si pedimos que se copie o imprima un documento, el documento hará copia o mandará imprimir a sus páginas y a su vez, las páginas harán lo mismo con los párrafos que contienen.

3.1.7. Relaciones de Dependencia o Uso

Una dependencia entre clases denota una relación de uso entre las mismas. Esto quiere decir que si la clase de la que se depende cambia, es posible que se tengan que introducir cambios en la clase dependiente. Por ejemplo, cuando un método de la clase A tiene como parámetro un objeto de la clase B, decimos que A depende de B, usa sus servicios.

Por ejemplo, si tenemos una clase *Impresora* que tiene una operación imprimir que toma como parámetro una instancia de la clase *Documento*, diremos que *Impresora* depende de *Documento*. Nótese que las impresoras no mantienen enlaces con los documentos que imprimen, sólo los procesan con la operación correspondiente, pero después de cada llamada a imprimir, no "recuerdan" los documentos que imprimieron. Se dice que este es el tipo de relación entre clases "más débil", y como recomendación general, sólo debería mostrarse en los diagramas cuando aporten alguna información importante..



Imaginemos que la clase impresora invoca a las tres operaciones de la instancia de *Documento* para imprimirlo. Pues bien, si por ejemplo, eliminamos la operación "obtenerPie" (o cambiamos los parámetros o el tipo de retorno de alguna de las operaciones de *Documento*), habrá que cambiar la implementación de imprimir.

Las dependencias no se reflejan en ningún elemento específico de Java, sólo que la clase dependiente hará referencia a la clase de la que depende en algún punto (variable local, parámetro), y por tanto suele ser habitual encontrar declaraciones `import` de las clases de las que se depende.

Nótese que toda relación de asociación implica lógicamente una relación de dependencia (una para cada uno de los sentidos de la asociación), pero no al contrario. Recordemos que una relación de asociación implica enlaces entre las clases que participan en la asociación; esto no es necesario en una dependencia.

3.2. Relaciones de generalización

La generalización es una relación taxonómica entre una clase más general (denominada superclase o clase padre) y una clase más específica (denominada subclase o clase hija). Lo fundamental de esta relaciones es que tienen que reflejar relaciones *"es un tipo de"*, es decir, las instancias de la subclase son un tipo específico de instancias de la clase padre. De ello se deriva una propiedad importante: los objetos de la clase hija pueden emplearse en cualquier lugar en que se requiera una instancia de la clase padre, pero no a la inversa. La subclase hereda las propiedades, el comportamiento y las relaciones de la superclase, a la vez que puede añadir sus propias propiedades, relaciones y comportamiento.

Nótese que la relación de herencia es transitiva "de abajo hacia arriba", es decir, si C es un tipo de B y B es un tipo de A, entonces C es un tipo de A.

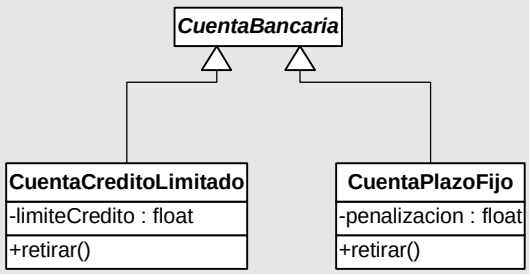
En UML la relación de generalización se representa mediante una línea sólida que une superclase y subclase. Además, esta línea sólida incorpora una flecha con punta triangular que queda localizada al lado de la superclase. Un conjunto de clases relacionadas entre si mediante relaciones de generalización constituye una jerarquía de herencia que puede ser un grafo o un árbol dependiendo, respectivamente, de si existe o no herencia múltiple.

Ejemplo. Cuentas Bancarias: Relación de generalización

Vamos a extender la clase CuentaBancaria anterior para soportar dos nuevos tipos de cuentas (una vez más, es un ejemplo no realista, sólo con propósitos didácticos):

1. Cuentas a plazo fijo, en las cuales si se realiza una retirada de dinero se produce una penalización sobre el importe retirado (por simplicidad, no consideramos la fecha de vencimiento). En estas cuentas, el saldo nunca puede ser negativo.
2. Cuentas con crédito limitado, en las cuales no se permite un nivel de "números rojos" superior a uno especificado al abrir la cuenta.

En ambos casos redefiniremos la operación retirar de la clase CuentaBancaria, y añadiremos el código necesario para cubrir los requisitos de los nuevos tipos de cuentas.



Con esta nueva extensión, podríamos considerar abstracta la clase CuentaBancaria (si es que en nuestra entidad bancaria imaginaria nuestra entidad bancaria sólo considera dos tipos de cuentas). En el diagrama anterior hemos marcado la clase CuentaBancaria como abstracta, lo que se visualiza e UML con el nombre de la clase en cursiva (aunque no lo mostramos aquí,

la cursiva se utiliza también para diferenciar los métodos abstractos en UML). Para cambiar la implementación en Java anterior, bastaría con poner el cualificador `abstract` antes de la palabra clase `class`.

El código correspondiente a la subclase CuentaPlazoFijo es el siguiente (el de la clase CuentaCreditoLimitado se deja como ejercicio al lector).

```
public class CuentaPlazoFijo extends CuentaBancaria {

     /** Porcentaje de penalizacion en retiradas */
     private float penalizacion;

     /**
      * Crea una cuenta a plazo fijo indicando
      * numero, saldo inicial y penalizacion sobre la
      * cantidad retirada en caso de sacar dinero.
      * @param penalizacion porcentaje entre 0 y 1  de penalizacion
      */
     public CuentaPlazoFijo(String numero, float saldoInicial,
                            float penalizacion){
        super(numero, saldoInicial);
        this.penalizacion = penalizacion;
     }

     /**
      * Redefinicion de retirar para imponer una penalizacion
      * a la cantidad retirada.
      */
     public void retirar(float cantidad) throws SaldoResultanteNoPermitido {
       if (saldo - cantidad - cantidad*penalizacion >=0 ){
           saldo -= cantidad;
           saldo -= cantidad * penalizacion;
       }
       else
          throw new SaldoResultanteNoPermitido();
     }
}
```

4. Otros Elementos de Modelado

4.1. Interfaces

Una interfaz en UML es la especificación de un conjunto de operaciones relacionadas que describen un servicio que puede implementar una clase (o un componente, aunque no lo vemos aquí). El uso de interfaces permite separar la implementación de las especificaciones, y por tanto es un concepto de diseño orientado a objetos muy importante. El esfuerzo de recopilación de conocimiento en diseño orientado a objetos –véase el primer capítulo de (Gamma 1995)– se fundamenta en un uso intensivo de las interfaces. Las interfaces en UML se representan como clases estereotipadas con el estereotipo `<<interface>>`, que se muestra encima del nombre de la interfaz.

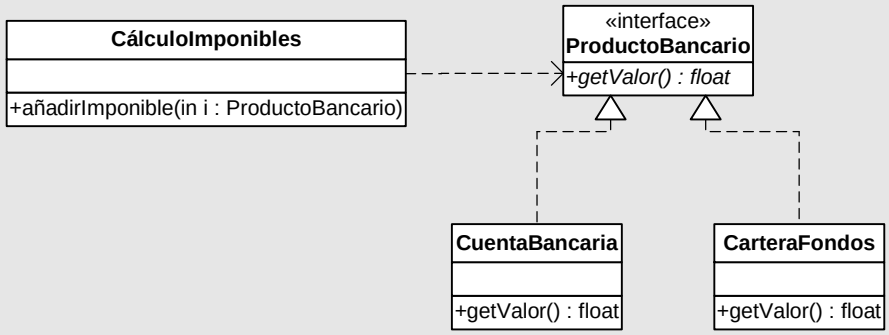
> Esto quiere decir que son un tipo especial de otro elemento de modelado de UML (en este caso, las interfaces se consideran un tipo de clase, pero esto queda fuera del alcance de este documento)

Las interfaces sólo poseen operaciones abstractas (en cursiva). Las interfaces pueden derivar de otras, en cuyo caso "heredan" la especificación de todas sus interfaces antecesoras.

Las clases que implementan la interfaz (es decir, que proveen la implementación para todas las operaciones especificadas en ella) se indican en el diagrama mediante un símbolo de generalización pero con línea discontinua.

Ejemplo. Cuentas Bancarias: Utilización de Interfaces

Siguiendo nuestro ejemplo anterior, consideremos ahora que se requiere una clase para el cálculo de impuestos. Esta clase necesita tener en cuenta todos los productos bancarios sujetos a impuestos que tiene el contribuyente. En este caso, nuestra clase *CálculoImponibles* proveerá un método para que la aplicación le indique los productos del cliente. Nótese que el servicio que requiere esta clase es solamente preguntar la base imponible en Euros del producto. Para ello, diseñamos una interfaz *ProductoBancario* con un método a tal efecto. Así, clases de jerarquías disjuntas, como cuentas bancarias y carteras de fondos, por ejemplo, pueden implementar esa interfaz y aparecer a la clase *CálculoImponibles* como *ProductosBancarios*, gracias al polimorfismo. Esto permite que en el futuro añadamos nuevos productos sin más que implementar la interfaz.



En Java, las interfaces son una construcción sintáctica parecida a las clases, pero que no permite implementar operaciones ni incluir atributos de instancia. Aunque en ocasiones se utilizan para "simular" la herencia múltiple que no posee el lenguaje, éste no es su cometido principal. Por ejemplo, la interfaz *ProductoBancario* se codificaría de la siguiente forma:

```
public interface ProductoBancario {
       public float getValor();
}
```

Y las clases que implementan la interfaz la incluirían en su cláusula *implements*, como por ejemplo:

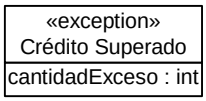
```
public class CuentaPlazoFijo extends CuentaBancaria implements ProductoBancario{
   //métodos y operaciones, incluyendo la implementación de getValor().
}
```

4.2. Excepciones

En UML, las excepciones son clases "estereotipadas" mediante `<<exception>>`.

> Como ya se explicó, esto quiere decir que son un tipo especial de otro elemento de modelado de UML (en este caso, las excepciones se consideran un tipo de señal, que a su vez es un tipo de clase, pero esto queda fuera del alcance de este documento)

Visualmente, se pueden representar igual que una clase, pero mostrando el estereotipo `<<exception>>` encima del nombre de la excepción. Por ejemplo, la siguiente figura muestra una excepción que podría aparecer en nuestra aplicación bancaria.

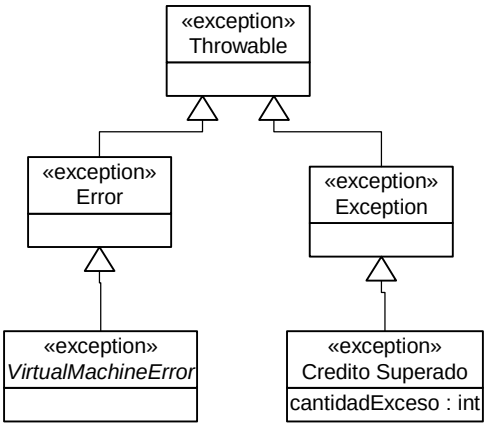


Una excepción puede tener atributos (que en este caso se suelen llamar *parámetros* de la excepción) que aportan más información sobre por qué se ha lanzado una determinada instancia de una excepción. En el ejemplo anterior, si se intenta hacer una operación que supera el crédito asignado a una cuenta, la cantidad en la cual se supera se puede incluir como información adicional. Las excepciones también pueden tener operaciones, aunque no es el caso de nuestro ejemplo.

Ejemplo. Cuentas Bancarias: Excepciones

Las excepciones que puede lanzar una determinada operación forman parte de la cabecera de la operación en muchos lenguajes, y en ocasiones, puede resultar clarificador indicarlo explícitamente en el modelo. Por ejemplo, el siguiente diagrama indica que la operación retirar de la clase *CuentaBancaria* puede lanzar instancias de la excepción *Crédito Superado* mediante una asociación con el estereotipo <<send>> entre la operación y la clase que representa la excepción.

El concepto de excepción UML es muy similar al de excepción en Java, por ejemplo, la siguiente figura muestra una parte de la jerarquía de excepciones en Java, incluyendo una definida por nosotros mediante herencia.



Como muestra la figura anterior, en Java las excepciones de nuestras aplicaciones deben derivar directa o indirectamente de `Exception`, mientras que las excepciones de la máquina virtual – como por ejemplo, cuando la máquina virtual se queda sin memoria – , predefinidas en el lenguaje, derivan de `Error`. Por tanto, nuestra excepción se implementaría derivando de `Exception` o de alguna de sus subclases, como se esquematiza en el siguiente fragmento de código:

```
public class CreditoSuperado extends Exception {

   private int cantidadExceso;

   public CreditoSuperado(int cantidad){
       cantidadExceso = cantidad;
   }

  //
  // otros métodos...
  // se suele redefinir al menos getMessage() y toString()
  //
}
```

5. Estructuración de las Clases: Paquetes y Múltiples Diagramas

Los paquetes en UML son agrupaciones de elementos del modelo de nuestro sistema, útiles cuando estamos modelando problemas grandes, que pueden llegar a contener cientos de clases y de otros elementos de UML.

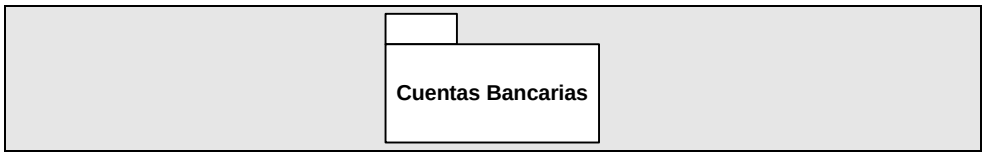
Un paquete contiene un conjunto de elementos cualesquiera del modelo (clases, relaciones, etc.) y puede contener otros paquetes, de modo que el modelo del sistema se puede estructurar en una jerarquía de paquetes anidados unos dentro de otros, formando un árbol.

La norma para agrupar en paquetes es poner juntos en el mismo paquete los elementos que estén semánticamente relacionados. Por otro lado, las propiedades generales que debe cumplir la estructuración en paquetes son alcanzar el mínimo acoplamiento entre paquetes (mínimo número de dependencias entre las clases o elementos de diferentes paquetes) y máxima cohesión entre los elementos de un mismo paquete.

Gráficamente, un paquete se representa con forma de carpeta. Existen elementos notacionales adicionales para representar visibilidad dentro de los paquetes y otros elementos, pero quedan fuera del alcance de este documento.

Ejemplo. Cuentas Bancarias: Paquetes UML

En nuestro ejemplo podríamos agrupar las clases vistas anteriormente sobre cuentas bancarias en un paquete denominado *"Cuentas Bancarias"*.



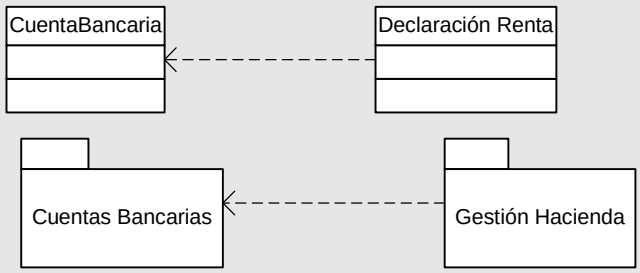
5.1. Dependencias entre paquetes

Pueden definirse dependencias entre paquetes que son el reflejo de las dependencias entre las clases que están contenidas en los paquetes.

Ejemplo. Cuentas Bancarias: Dependencias entre paquetes

Por ejemplo, si añadimos la gestión de las declaraciones de la Renta por el banco, tendremos una clase *DeclaraciónRenta* –que incluiremos en un nuevo paquete *"Gestión Hacienda"*– que tendrá una dependencia (y quizá una asociación, pero eso no nos interesa aquí) con la clase *CuentaBancaria*, indicando la cuenta en la que se debe hacer el reintegro o pago (dependiendo de si es positiva o negativa) del resultado de la declaración.

Así, la dependencia entre las clases, se traducirá en una dependencia entre los paquetes.

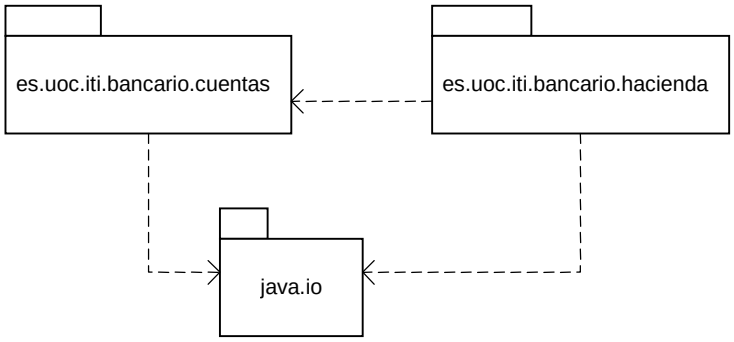


5.2. Paquetes UML y paquetes Java

Aunque tienen el mismo nombre, son conceptos diferentes, ya que un paquete en UML es un agrupador de elementos del modelo, mientras que un paquete Java agrupa clases Java, que no necesariamente tienen que guardar una correspondencia uno a uno con los elementos del modelo.

Podemos representar la estructura de las clases Java resultantes del modelo mediante un diagrama de paquetes aparte.

Así, nuestro anterior diagrama podría traducirse a la siguiente estructuración de paquetes en Java (siguiendo las convenciones de nombrado de paquetes Java típica de acuerdo al dominio de la organización donde se desarrolla el software, en nuestro caso, las Ingenierías Técnicas en Informáticas–ITIs):



También hemos incluido el paquete de las librerías estándar `java.io` mostrando que la implementación en Java de las clases se haría guardando en disco la información mediante las bibliotecas de flujos de Java.

Ejemplo. Cuentas Bancarias: Paquetes en Java

Como ejemplo ilustrativo, la clase *DeclaracionRenta* podría tener la siguiente cabecera que refleja de manera explícita sus dependencias (nótese que la clase *CuentaBancaria* debe haberse declarado como `public class` para poder ser vista desde otros paquetes Java):

```
package es.uoc.iti.bancario.hacienda;

import java.io.*;
import es.uoc.iti.bancario.cuentas.CuentaBancaria;

public class DeclaracionRenta{
  // operaciones que usan CuentaBancaria y clases de java.io
  // y otras operaciones...
}
```

6. Recursos

Object Managment Group (OMG). Unified Modeling Language (UML), version 1.3, Document *formal/00-03-01*, disponible en http://www.omg.org/technology/documents/formal/uml.htm

7. Bibliografía

(Booch 1999) **Booch, G., Rumbaugh, J., Jacobson, I.** El Lenguaje Unificado de Modelado, Addison Wesley, 1999.

(Gamma 1995) **Gamma, E., Helm, R., Johnson, R. Vlissides, J.** Addison-Wesley, 1995.