UNIVERSIDAD POLITÉCNICA DE MADRID
ESCUELA UNIVERSITARIA DE INFORMÁTICA
LENGUAJES, PROYECTOS Y SISTEMAS INFORMÁTICOS
Luis Fernández Muñoz
setillo@eui.upm.es

CAPITULO 3

PROGRAMA ORIENTADO A OBJETOS: COLABORACION ENTRE OBJETOS

> *"Cuando los programadores piensan en los problemas, en términos de comportamientos y responsabilidades de los objetos, traen con ellos un caudal de intuición, ideas y conocimientos provenientes de su experiencia diaria".* [Budd, 94]

> *"En lugar de un saqueador de bits que saquea estructuras de datos, nosotros tenemos un universo de objetos con buen comportamiento que cortésmente se solicitan entre sí cumplir diversos deseos".* [Ingalls, 81]

> *"Un objeto en si mismo no es interesante. Los objetos contribuyen al comportamiento de un sistema colaborando con otros objetos".* [Booch, 96]

De estas citas bibliográficas, se pueden extraer ciertas características a las que debe responder un programa orientado a objetos correctamente construido:

- cada objeto de un programa debe asumir ciertas responsabilidades: mantener datos y realizar operaciones; las operaciones que realiza el objeto deben estar estrechamente ligadas a los datos que soporta el objeto, de forma equivalente a cómo el ser humano entiende el mundo.

  Pero esta equivalencia no debe dar pie a tratar de "codificar el mundo entero"; el **carácter antropomórfico** de la programación orientada a objetos ("…traen con ellos un caudal de intuición, ideas y conocimientos provenientes de su experiencia diaria …") se refiere a que cada cosa ocupe su lugar esperado.

  Por ejemplo, al igual que nadie concibe que en cada aula de una universidad haya una secretaría; que sea el alumno el que maneje el ordenador de la secretaría para obtener un certificado de estudios; que un profesor corrija los exámenes sentado en el rectorado; etc. en los mismos términos debe evaluarse el código de una clase. Simplemente, cada clase debe ser un bloque comprensible en sí mismo, tanto en los datos que reúne como en las operaciones asignadas a la clase.

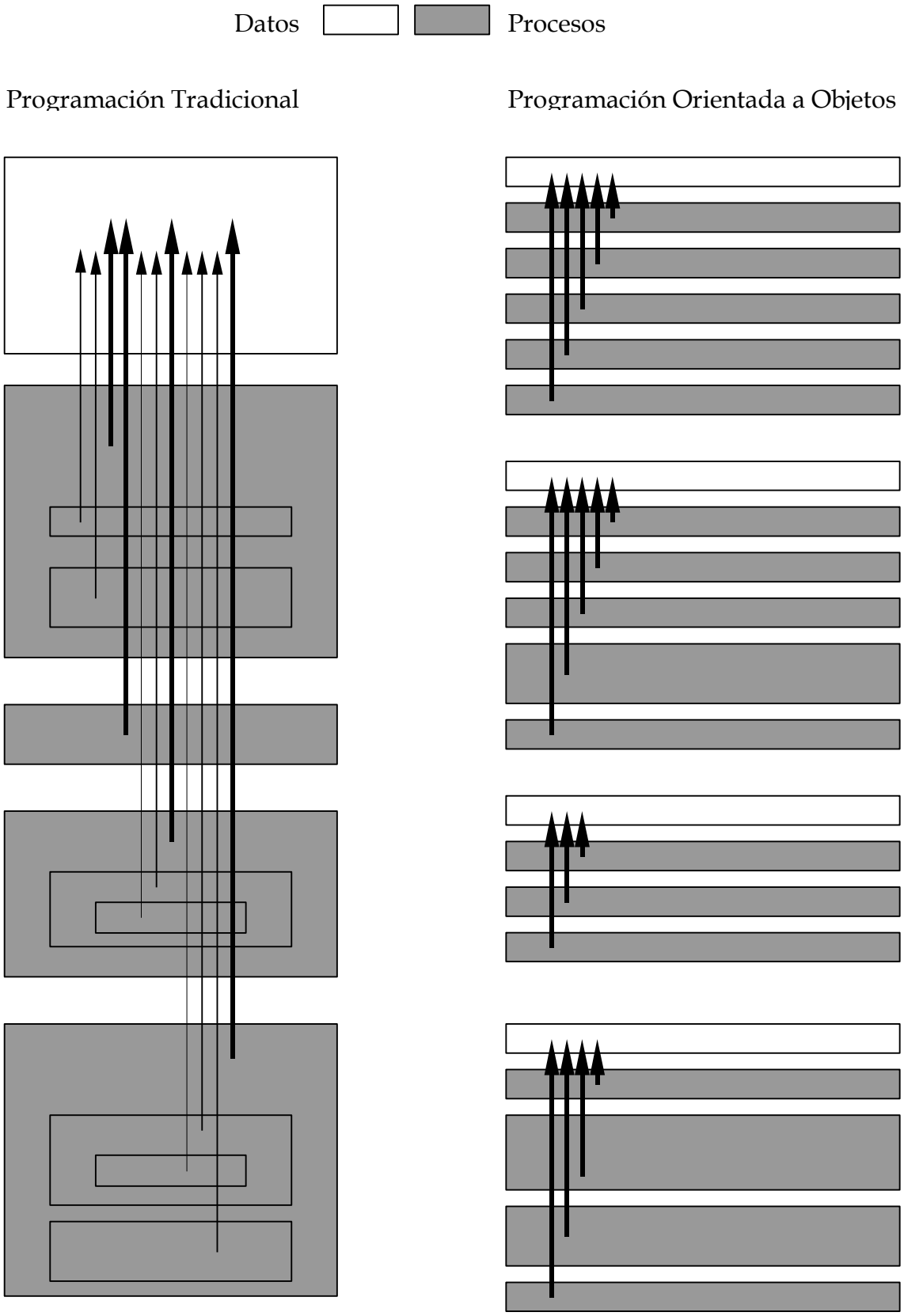


Figura 3. 1. Relación entre datos y procesos en las distintas programaciones.

- cada objeto de un programa debe **asumir directamente cierta parte de sus responsabilidades y saber delegar otras en otros objetos colaboradores**. No se trata de que un objeto asuma y "controle todo" y el resto se dispongan a darle la información y aceptarla cuando el "controlador" la haya procesado; se trata de que cada objeto almacene su propia información y, por ende, que cada objeto la procese.

  Por tanto, no es admisible "codificar un programa orientado a objeto de un diseño orientado a procesos". No se trata de que constantemente un objeto "extraiga" toda la información de sus objetos "colaboradores", mediante "correctos" mensajes, procese "sus" datos y se los "empotre" de vuelta, mediante "correctos" mensajes.

  Un objeto debe dar ordenes para que sus objetos colaboradores procesen ellos mismos sus datos a gusto del objeto que emitió el mensaje. En el caso "problemático" de que cierto proceso necesite de datos repartidos entre dos objetos, la "tendencia" en programación orientada a objetos debe ser que un objeto dé la orden al otro objeto de lo que precisa junto con la información que él contiene para que el otro pueda realizar la orden.

3.1. Método de Desarrollo de Programas.

Un **programa orientado a objetos en ejecución es una colección jerárquica de objetos** contenidos en otros objetos sucesivas veces que colaboran todos entre sí. Paralelamente, **un programa en edición/compilación es una colección de las clases** de estos objetos relacionadas entre sí.

Las funciones del "programa principal" las asumirá un único objeto que englobe todo el sistema de objetos (mediante un desencadenamiento de instanciaciones) al que lanzándole un único mensaje realice todas las funciones deseadas (mediante un desencadenamiento de mensajes). Fuera de ese objeto no puede existir nada, no tiene sentido que reciba parámetros, no tiene sentido devolver un valor a nadie.

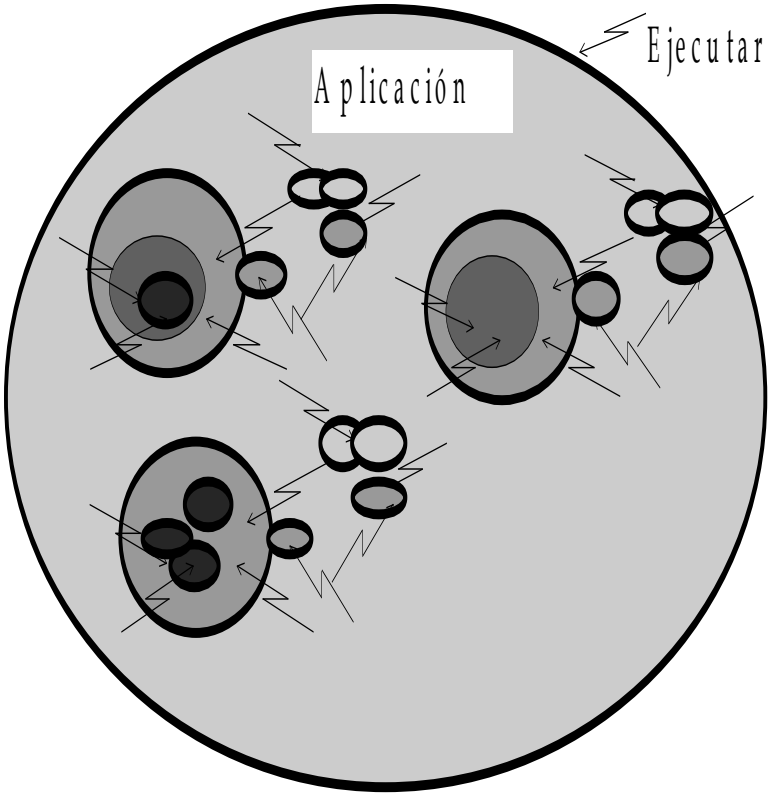


Figura 3.7. Programa Orientado a Objetos.

La clase de este objeto "principal" suele responder a nombres como *Aplicación*, *Sistema*, *Gestor*, *Simulador*, *Juego*, etc. El mensaje "principal" suele responder a nombres como *ejecutar*, *gestionar*, *simular*, *jugar*, etc. Por tanto, de forma genérica para cualquier programa orientado a objetos:

```
class Aplicacion
{

...

public void ejecutar ()
{
      ...
}

public static void main (string arg[])
{
      Aplicacion aplicacion = new Aplicacion();
      aplicacion.ejecutar();
}

}
```

Una vez establecido el objeto "principal" e inicial queda por determinar cuántos otros objetos existen en el programa: ¿hasta qué punto se realizan descomposiciones sucesiva de objetos en objetos que forman la jerarquía de composición, fundamental en un programa orientado a objetos? La respuesta es de la misma naturaleza que en programación tradicional respecto a su bloque de construcción: el subprograma. De forma paralela, ¿hasta que punto se realizan descomposiciones sucesiva de subprogramas en subprogramas que forman la jerarquía de llamadas, fundamental en un programa tradicional?

La respuesta en ambos casos es la misma:

> *"Dado el mismo conjunto de requisitos, más módulos significa menor tamaño individual del módulo. Sin embargo, a medida que crece el número de módulos, el esfuerzo asociado con la integración de módulos también crece. Esta características llevan a una curva de coste total. Hay un número, M, de módulos que resultaría en un coste de desarrollo mínimo, pero no tenemos la sofisticación necesaria para predecir M con seguridad"*
> [Pressman, 93]

Entonces, dado que no se puede establecer el **número de distintas clases que tiene un programa**, se recurre a la siguiente regla de desarrollo: aceptar una clase más si existen fundamentos para pensar que se reducen los costes de desarrollo totales que favorezca mayores cotas de abstracción, encapsulación, modularización y jerarquización que faciliten la legibilidad, mantenimiento, corrección, etc.

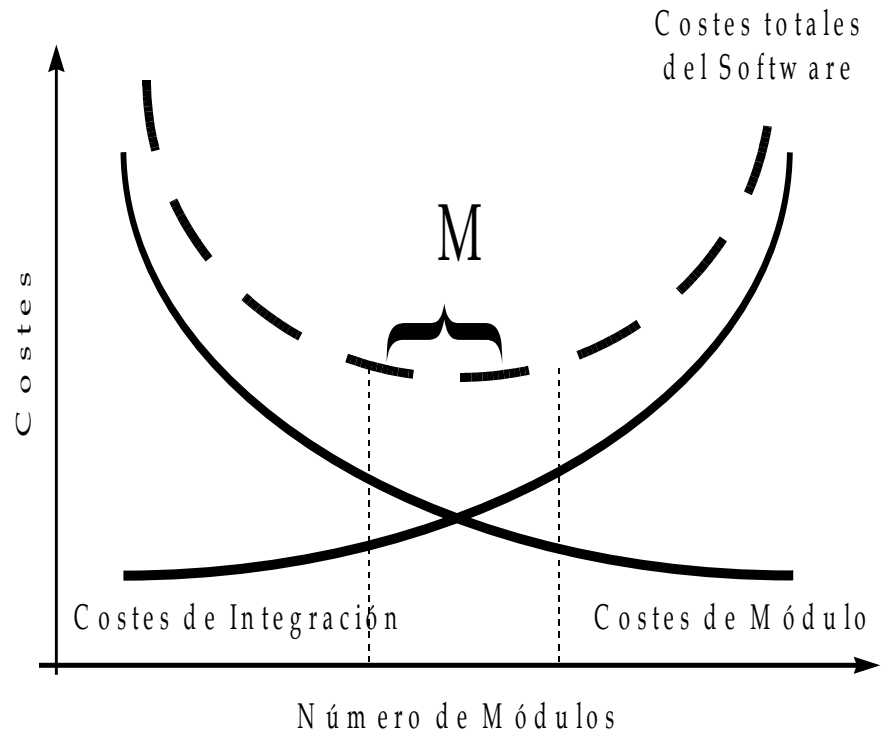


Figura 3.8. Relación de Costes y Número de Módulos

Una vez que se conoce de dónde partir (la clase del objeto "principal") y hasta dónde llegar (hasta que nuevas clases no reduzcan los costes), resta saber qué camino seguir para el desarrollo del programa.

El **método** propuesto es una simplificación extrema y variación de HOOD (diseño orientado a objetos jerárquico). El resultado es un método muy sencillo y utilizable en pequeñas aplicaciones. Este método contiene 5 fases que se aplican cíclicamente para cada clase del programa:

1. Tomar una clase
2. Definir el comportamiento
3. Definir los atributos
4. Codificar los métodos
5. Recapitular nuevas clases y métodos.

A continuación se aplicará el método fase a fase bajo la siguiente nomenclatura: "F.C" siendo 'F' la fase actual (con valores de 1 a 5) y 'C' el ciclo de aplicación (con valores de 1 a N clases del programa). Posteriormente, se sintetizarán esquemáticamente todas las fases del método.

- o O o -

El programa a resolver debe permitir a dos usuarios jugar a las "Tres En Raya" (*TER*):

- inicialmente el tablero estará vacío;
- los jugadores pondrán las fichas alternativamente, siendo posible alcanzar las "Tres en Raya" con la quinta puesta;

- en caso contrario, una vez puestas las 6 fichas, se moverán las fichas alternativamente;
- tanto en las puestas como en los movimientos, se deben requerir las coordenadas a usuario validando la corrección de la puesta/ movimiento;
- se debe detectar automáticamente la finalización del juego al lograrse "Tres en Raya": tras la quinta y sexta puesta y tras cada movimiento.

1.1) Tomar una clase: al principio de todo el desarrollo del programa será la clase del objeto que engloba todo el sistema - *TER*-.

```
class TER
{
public static void main (string arg[])
{
      TER partida = new TER();
      partida.jugar();
}

}
```

2.1) Determinar el comportamiento: al principio de todo el desarrollo del programa será el método de la clase que engloba todo el sistema que dispare toda la funcionalidad deseada *-jugar-:* no tendrá parámetros y no devolverá nada; el juego es "todo" el universo de objetos.

```
class TER
{

public void jugar ()
{
      ...
}

public static void main (string arg[])
{
      TER partida = new TER();
      partida.jugar();
}

}
```

3.1) Determinar los atributos con sus respectivas clases; no es estrictamente necesario determinar todos y cada uno de los atributos, sino que se deben establecer aquellos que sean "obvios" en el momento; si se omitiese alguno surgirá su necesidad en el siguiente punto 4.1).

Los atributos soportarán los datos que posibiliten el proceso del juego: memorizar dónde están las fichas puestas/movidas, saber a qué jugador le corresponde poner/mover y su color de fichas, etc.

En programación tradicional, se definirían todas las estructuras de datos necesarias (tablas bidimensionales, ...) para que las manipulen todos los subprogramas. En programación orientada a objetos, se recurre a atributos que son objetos de otras clases responsables de los detalles del soporte y manipulación de esos datos;

Para el programa de las "Tres en Raya":

- se crea un objeto de la clase *Tablero*; responsable de aparecer vacio, guardar las fichas que se pongan, detectar las "Tres en Raya", mostrar la situación de las fichas, etc.
- se crean dos objetos de la clase *Jugador*; responsables de memorizar el color de cada uno, comunicarse con cada usuario y controlar automáticamente sus puestas y movimientos, etc.
- se crea un objeto de la clase *Turno*; responsable de memorizar a qué jugador le corresponde poner/mover, alternar el jugador, etc.

```
class TER
{
private final Tablero tablero = new Tablero();
private final Jugador jugadores[] = new Jugador[2];
private final Turno turno = new Turno();

public TER
{
      jugadores[0]=new Jugador('o');
      jugadores[0]=new Jugador('x');
}

public void jugar ()
{
      ...
}

public static void main (string arg[])
{
      TER partida = new TER();
      partida.jugar();
}

}
```

4.1) Codificar métodos: no debe ser un impedimento incluir en la codificación de los métodos el paso de mensajes correspondientes a métodos que no se encuentren definidos previamente en sus correspondientes clases. Hay que codificar los métodos de la clase actual, *TER*, delegando tareas a los objetos de las clases de los atributos, *Tablero*, *Jugador* y *Turno*.

```
class TER
{
private final Tablero tablero = new Tablero();
private final Jugador jugadores[] = new Jugador[2];
private final Turno turno = new Turno();

public TER
{
      jugadores[0]=new Jugador('o');
      jugadores[0]=new Jugador('x');
}

public void jugar ()
{
      tablero.mostrar();
      for (int i=0; i<5 ; i++)
      {
            jugadores[turno.toca()].poner(tablero);
            turno.cambiar();
            tablero.mostrar();
      }
      if (tablero.hayTER())
            jugadores[turno.noToca()].victoria();
      else
      {
            jugadores[turno.toca()].poner(tablero);
            tablero.mostrar();
            while (!tablero.hayTER())
            {
                  turno.cambiar();
                  jugadores[turno.toca()].mover(tablero);
                  tablero.mostrar();
            }
            jugadores[turno.toca()].victoria();
      }
}

public static void main (string arg[])
{
      TER partida = new TER();
      partida.jugar();
}

}
```

5.1) Recapitular: nuevas clases aparecidas en los puntos 2.1), 3.1) y 4.1) y métodos correspondientes a los mensajes enviados a objetos de estas clases en el punto

4.1). El valor devuelto y parámetros de cada método se extrae del contexto del envío del mensaje.

```
class Tablero
{
public void mostrar () {...}
public boolean hayTER () {...}
}
```

```
class Jugador
{
public Jugador (char color) {...}
public void poner (Tablero tablero) {...}
public void mover (Tablero tablero) {...}
public void victoria () {...}
}
```

```
class Turno
{
public void cambiar() {...}
public int toca() {...}
public int noToca() {...}
}
```

La recapitulación abre de nuevo la aplicación de las 5 fases del método para cada una de las clases, pero con los siguientes matices:

- la primera fase, tomar una clase, que "anulada" al limitarse a escoger una de las clases recapituladas anteriormente. Se recomienda que la elección sea la que inspire más riesgos en su implantación;
- la segunda fase, determinar el comportamiento público de clases anteriores queda "anulada" por la quinta fase de recapitulación anterior.

1.2) Tomar una clase: *Jugador* de la fase 5.1)
2.2) Determinar el comportamiento: de la fase 5.1)
3.2) Determinar los atributos con sus respectivas clases; para *poner*, *mover* y cantar *victoria* un *Jugador* solo precisa saber el color de sus fichas.

```
class Jugador
{
private char color;

public Jugador (char color) {...}
public void poner (Tablero tablero) {...}
public void mover (Tablero tablero) {...}
public void victoria () {...}
}
```

4.2) Codificar métodos: no debe ser un impedimento incluir en la codificación de los métodos nuevas clases "auxiliares" no previstas hasta el momento.

```
class Jugador
{
private char color;

public Jugador (char color)
{
      this.color=color;
}
public void poner (Tablero tablero)
{
      (new GestorIO()).out("juega: "+color);
      Coordenada destino=new Coordenada();
      do
      {
            destino.recoger("Coordenada destino de puesta");
      } while(!destino.valida() || tablero.ocupado(destino));
      tablero.poner(destino,color);
}

public void mover (Tablero tablero)
{
      Coordenada origen=new Coordenada();
      do
      {
            origen.recoger("Coordenada origen de movimiento");
      } while (!origen.valida() || tablero.vacio(origen));
      tablero.sacar(origen);
      this.poner(tablero);
}

public void victoria ()
{
      (new GestorIO()).out ("las "+color+" han ganado y ...");
}

}
```

Esta solución opta por la inclusión de una nueva clase *Coordenada* que aglutine (encapsule, modularice, reutilice, ...) el tratamiento de la fila y la columna de una casilla del tablero donde poner/mover fichas.

5.2) Recapitular

```
class Tablero
{
// fase recapitulación 5.1
public void mostrar () {...}
public boolean hayTER () {...}

// fase recapitulación 5.2
public boolean ocupado (Coordenada coordenada) {...}
public void poner (Coordenada coordenada, char color) {...}
public boolean vacio (Coordenada coordenada) {...}
public void sacar (Coordenada coordenada, char color) {...}
}
```

```
class Coordenada
{
public void recoger (String titulo) {...}
public boolean valida () {...}
}
```

1.3) Tomar una clase: *Tablero* de la fase 5.1)
2.3) Determinar el comportamiento: de las fases 5.1) y 5.2)
3.3) Determinar los atributos con sus respectivas clases; para *poner*, *sacar* y detectar si *hayTER* o si una casilla está *ocupado* o *vacia* un *Tablero* solo precisa soportar 3*x*3 caracteres que memoricen las posiciones de las fichas (el carácter '-' representa la ausencia de fichas) .

```
class Tablero
{
private final char fichas[][] = new char[3][3];
private final char VACIO = '-';

public void mostrar () {...}
public boolean hayTER () {...}
public boolean ocupado (Coordenada coordenada) {...}
public void poner (Coordenada coordenada, char color) {...}
public boolean vacio (Coordenada coordenada) {...}
public void sacar (Coordenada coordenada, char color) {...}
}
```

4.3) Codificar métodos: no debe ser un impedimento incluir en la codificación de los métodos nuevas clases "auxiliares" no previstas hasta el momento.

```
class Tablero
{
private final char fichas[][] = new char[3][3];
private final char VACIO = '-';

public Tablero ()
{
   for(int i=0;i<3;i++)
      for(int j=0;j<3;j++)
         fichas[i][j] = VACIO;
}
```

```
public void mostrar ()
{
      GestorIO gestorIO = new GestorIO();
      for(int i=0;i<3;i++)
      {
            for(int j=0;j<3;j++)
            {
                  gestorIO.out(fichas[j][i]);
                  gestorIO.out(' ');
            }
            gestorIO.out('\n');
      }
      gestorIO.out('\n');
}

public boolean hayTER ()
{
      return this.hayTER('o') || this.hayTER('x');
}

private boolean hayTER (char color)
{
      boolean victoria = false;
      int diagonal = 0;
      int inversa = 0;
      int filas[] = new int[3];
      int columnas[] = new int[3];

      for(int i=0;i<3;i++)
      {
            filas[i] = 0;
            columnas[i] = 0;
            for(int j=0;j<3;j++)
                  if(fichas[i][j]==color)
                  {
                        if (i==j)
                              diagonal++;
                        if (i+j==2)
                              inversa++;
                        filas[i]++;
                        columnas[j]++;
                  }
      }
      if ((diagonal==3)||(inversa==3))
         victoria = true;
      else
            for(int i=0;i<3;i++)
                  if ((columnas[i]==3)||(filas[i]==3))
                        victoria = true;
      return victoria;
}


public boolean ocupado (Coordenada coordenada)
{
    return fichas[coordenada.getFila()]
                  [coordenada.getColumna()] != VACIO;
}
```

```
public boolean vacio (Coordenada coordenada)
{
    return (!this.ocupado(coordenada));
}

public void poner (Coordenada coordenada, char color)
{
     fichas[coordenada.getFila()]
           [coordenada.getColumna()] = color;
}

public void sacar (Coordenada coordenada)
{
     this.poner(coordenada, VACIO);
}

}
```

5.3) Recapitular

```
class Coordenada
{
// fase recapitulación 5.2
public void recoger (String titulo) {...}
public boolean valida () {...}

// fase recapitulación 5.3
public int getFila () {...}
public int getColumna () {...}
}
```

1.4) Tomar una clase: *Coordenada* de la fase 5.2)
2.4) Determinar el comportamiento: de las fases 5.2) y 5.3)
3.4) Determinar los atributos con sus respectivas clases; para *recoger*, *validar* y obtener fila y columna de una coordenada, solo se precisa memorizar la fila y la columna.

```
class Coordenada
{
private int fila;
private int columna;

public void recoger (String titulo) {...}
public boolean valida () {...}
public int getFila () {...}
public int getColumna () {...}
}
```

4.3) Codificar métodos:

```
class Coordenada
{
private int fila;
private int columna;
```

```
public void recoger (String titulo)
{
      GestorIO gestorIO=new GestorIO();
      gestorIO.out(titulo+"\n");
      gestorIO.out("Dame fila: ");
      fila=gestorIO.inInt();
      gestorIO.out("Dame columna: ");
      columna=gestorIO.inInt();
}

public boolean valida ()
{
      return fila<=2 && fila>=0 && columna<=2 && columna>=0;
}

public int getFila ()
{
      return fila;
}

public int getColumna ()
{
      return columna;
}

}
```

5.4) Recapitular: no surgen ni nuevas clases ni nuevos métodos de clases ya recapituladas.

1.5) Tomar una clase: *Turno* de la fase 5.1)
2.5) Determinar el comportamiento: de las fases 5.1)
3.5) Determinar los atributos con sus respectivas clases; para *cambiar* y obtener a quien *toca* y el contrario solo se precisa memorizar un valor alternativo entre 1 y 2.

```
class Turno
{
private int valor=0;

public void cambiar() {...}
public int toca() {...}
public int noToca() {...}
}
```

4.5) Codificar métodos:

```
class Turno
{
private int valor=0;

public void cambiar()
{
      this.valor=(this.valor+1)%2;
}

public int toca()
{
      return valor;
}
public int noToca()
{
      return (valor+1)%2;
}

}
```

5.5) Recapitular: no surgen ni nuevas clases ni nuevos métodos de clases ya recapituladas.

- o O o -

El **método aplicado queda esquematizado** en:

1) Tomar una clase.

Al principio de todo el desarrollo del programa será la clase que engloba todo el sistema - *Ajedrez, SimuladorVuelo, Gestor, etc.-*; posteriormente será una de las clases "recapituladas" en el punto 5.x).

↓

2) Determinar las cabeceras de los métodos públicos requeridos a dicha clase.

O bien es un único método - *jugar, arrancar, gestionar, etc.-* en el caso de la clase que engloba todo el sistema - *Ajedrez, Simulador, Gestor, etc.-*, o bien los métodos vienen previamente determinados por la recapitulación de mensajes del punto 5.x) para las demás clases.

↓

3) Determinar los atributos con sus respectivas clases.

No es estrictamente necesario determinar todos y cada uno de los atributos, sino que se deben establecer aquellos que sean "obvios" en el momento para la codificación de los métodos; si se omitiese alguno surgirá su necesidad en el siguiente punto 4).

No debe ser un impedimento incluir nuevas clases en los atributos que no se encuentren definidas previamente.

↓

4) Definir los métodos del punto 2).

No debe ser un impedimento incluir nuevas clases en las declaraciones locales que no se encuentren definidas previamente.

No debe ser un impedimento incluir en la codificación de los métodos el paso de mensajes correspondientes a métodos que no se encuentren definidos previamente en sus correspondientes clases.

↓

5) Recapitular las nuevas clases aparecidas en los puntos 2), 3) y 4) y mensajes enviados a objetos de estas clases en el punto 4).

- Si quedan métodos de clases por definir volver al punto 4) hasta que estén completamente definidas todas las clases recapituladas.

- Si existen nuevas clases por definir volver al punto 1) hasta que estén definidas todas las clases recapituladas.

- o O o -

El diseño de la solución anterior se basa en 5 clases: *TER*, *Jugador*, *Tablero*, *Coordenada* y *Turno*. El método aplicado no establece pautas para el número idoneo de clases. Éste número, como establece Pressman, depende de los costes de desarrollo y, hoy en día, no hay fórmulas que lo determine.

A continuación se presenta una nueva solución al mismo problema de las "Tres en Raya" pero eliminando dos clases, *Jugador* y *Turno* de las 5 anteriores. Decantarse por una u otra solución dependerá de la experiencia del programador, de la legibilidad del código deseada, de la potencialidad de cambios, etc.

```
class TER
{
private final Tablero tablero=new Tablero();

public void jugar ()
{
      GestorIO gestorIO=new GestorIO();
      char turno = 'o';
      tablero.mostrar();
      for (int i=0; i<5 ; i++)
      {
            this.poner(turno);
            turno = (turno == 'o'?'x':'o');
            tablero.mostrar();
      }
      if (tablero.hayTER())
            if (turno == 'o')
                  gestorIO.out ("las crucecitas han ganda....");
            else
                  gestorIO.out ("las pelotitas han ganda....");
      else
      {
            this.poner(turno);
            tablero.mostrar();
            while (!tablero.hayTER())
            {
                  turno = (turno == 'o'?'x':'o');
                  this.poner(turno);
                  tablero.mostrar();
            }
            if (tablero.hayTER())
                  if (turno == 'o')
                        gestorIO.out("las pelotitas han ganda.");
                  else
                        gestorIO.out("las crucecitas han ganda.");
      }
}

private void poner (char turno)
{
      (new GestorIO()).out("pone: "+turno);
      Coordenada destino=new Coordenada();
      do
      {
            destino.recoger("Coordenada destino de puesta");
      } while(!destino.valida() || tablero.ocupado(destino));
      tablero.poner(destino,turno);
}
```

```
private void mover (char turno)
{
      (new GestorIO()).out("saca: "+turno);
      Coordenada origen=new Coordenada();
      do
      {
            origen.recoger("Coordenada origen de movimiento");
      } while (!origen.valida() || tablero.vacio(origen));
      tablero.sacar(origen);
      this.poner(turno);
}

public static void main (String arg[])
{
   TER partida=new TER();
   partida.jugar();
}

}

// las clases Tablero y Coordenada persisten intactas
```

Comparando esta nueva solución con la anterior se aprecia que en la medida que decrece el número de clases de un diseño, disminuye la legibilidad (p.e.: `turno = (turno == 'o'?'x':'o')`) y aumenta el tamaño líneas (p.e.: *jugar* aumenta en un 75%) y métodos de las clases (p.e.: en *TER* aumenta en un 66%). Por tanto, una pauta para decantarse por nuevas clases es pretender "clarificar/aligerar" a otras clases.

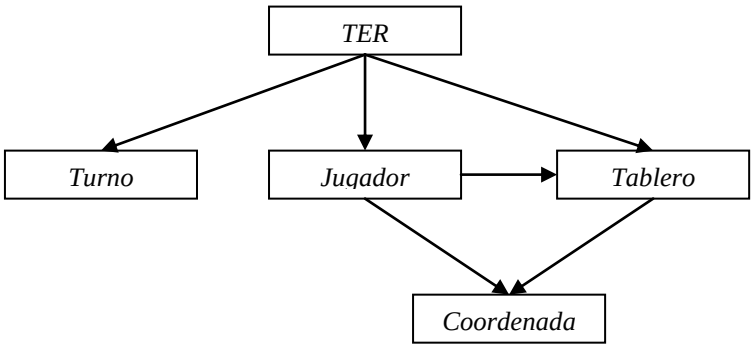
- o O o -

A continuación se establecerá un **puente entre los conceptos del capítulo anterior y el actual** en los siguientes términos:

- visión estática: un programa orientado a objetos es una colección de clases que permiten la abstracción, la encapsulación, la modularidad y la jerarquía;
- visión dinámica: un programa orientado a objetos en ejecución es un universo de objetos que colaboran entre sí gracias al desencadenamiento de objetos y mensajes;

La **visión estática** del código muestra una jerarquía de clases que se "apoyan" en otras clases: usan objetos de otras clases como atributos, parámetros o locales auxiliares.

Para las "Tres en Raya" quedaría:



Cada clase ofrece un nombre y un interfaz público de métodos que permite la abstracción de un concepto: esencialemente qué es y qué puede hacer. Por otro lado, la vista privada encapsula los detalles: cómo se soportan los datos y cómo se manipulan para realizar las operaciones del interfaz público.

La combinación de abstracción y encapsulación permite la modularidad de un programa. Todas las clases forman una colección de elementos poco acoplados (únicamente por el interfaz público sin acceso a la implantación) y cohesivos (autocomprensibles). De esta manera, los cambios en una clase no repercuten en todo el programa.

Prueba de la modularidad en el ejemplo de las "Tres en Raya" es optar por una implantación alternativa de la clase *Tablero* sin repercutir en las clases *TER, Jugador* y *Turno*; tan solo exige ampliar el comportamiento de la clase *Coordenada* en la que se "apoya". La alternativa propuesta propone que en vez de almacenar como atributo los 3*x*3 caracteres para memorizar las posiciones de las fichas, se puede soportar la misma información con las 3*x*2 coordenadas donde se encuentran éstas.

```
class Tablero
{
private Coordenada fichas[][]=new Coordenada[2][3];

public Tablero ()
{
      for (int i=0; i<fichas.length; i++)
            for (int j=0; j<fichas[i].length; i++)
                  fichas[i][j] = null;
}
```

```
public void mostrar ()
{
      for (int i=0; i<3; i++)
      {
            String fila=" ";
            for (int j=0; j<3; j++)
            {
                  Coordenada actual=new Coordenada(i,j);
                  if (this.vacio(actual))
                        fila=fila+"  ";
                  else if (this.ocupado(actual,'x'))
                        fila=fila+"x ";
                  else
                        fila=fila+"o ";
       }
            GestorIO gestorIO=new GestorIO();
            gestorIO.out(fila+'\n');
      }
}

public boolean hayTER ()
{
      boolean resultado=false;
      for (int i=0; i<fichas.length; i++)
            resultado= resultado ||
                  fichas[i][0].enFila(fichas[i]) ||
                  fichas[i][0].enColumna(fichas[i]) ||
                  fichas[i][0].enDiagonal(fichas[i]);
      return resultado;
}

public boolean ocupado (Coordenada coordenada)
{
      return this.ocupado(coordenada,'x') ||
                  this.ocupado(coordenada,'o');
}

private boolean ocupado (Coordenada coordenada, char color)
{
      int fila = (color=='o'? 0 : 1);
      boolean resultado = false;
      for (int j=0; j<fichas[fila].length; j++)
            resultado = resultado || fichas[fila][j]!=null &&
                              fichas[fila][j].iguales(coordenada);
      return resultado;
}

public void poner (Coordenada coordenada, char color)
{
      int fila = (color=='o'? 0 : 1);
      int j = 0;
      while (fichas[fila][j] != null)
            j++;
      fichas[fila][j] = coordenada;
}
```

```
public boolean vacio (Coordenada coordenada)
{
      return !this.ocupado(coordenada);
}
public void sacar (Coordenada coordenada)
{
      for (int i=0; i<fichas.length; i++)
            for (int j=0; j<fichas[i].length; j++)
                  if (coordenada.iguales(fichas[i][j]))
                        fichas[i][j] = null;
}

}
```

```
class Coordenada
{
private int fila;
private int columna;

public Coordenada ()
{
}

public Coordenada (int fila, int columna)
{
      this.fila = fila;
      this.columna = columna;
}

public void recoger (String titulo)
{
      GestorIO gestorIO=new GestorIO();
      gestorIO.out(titulo+"\n");
      gestorIO.out("Dame fila: ");
      fila=gestorIO.inInt();
      gestorIO.out("Dame columna: ");
      columna=gestorIO.inInt();
}

public boolean valida ()
{
      return fila<=2 && fila>=0 && columna<=2 && columna>=0;
}

public boolean iguales (Coordenada coordenada)
{
      return fila==coordenada.fila &&
                  columna==coordenada.columna;
}
```

```
public boolean enFila (Coordenada coordenadas[])
{
     boolean resultado=true;
     for (int i=0; i<coordenadas.length; i++)
          resultado=resultado && coordenadas[i]!=null &&
                    this.fila==coordenadas[i].fila;
     return resultado;
}

public boolean enColumna (Coordenada coordenadas[])
{
     boolean resultado=true;
     for (int i=0; i<coordenadas.length; i++)
          resultado=resultado && coordenadas[i]!=null &&
                    this.columna==coordenadas[i].columna;
     return resultado;
}

public boolean enDiagonal (Coordenada coordenadas[])
{
 boolean resultado=true;
     if (this.fila+this.columna==2)
          for (int i=0; i<coordenadas.length; i++)
               resultado = resultado &&
                                   (coordenadas[i].fila+
                                    coordenadas[i].columna==2);
     else if (this.fila-this.columna==0)
          for (int i=0; i<coordenadas.length; i++)
          resultado = resultado &&
                                   (coordenadas[i].fila-
                                    coordenadas[i].columna==0);
     else
          resultado = false;
  return resultado;
}

}
```

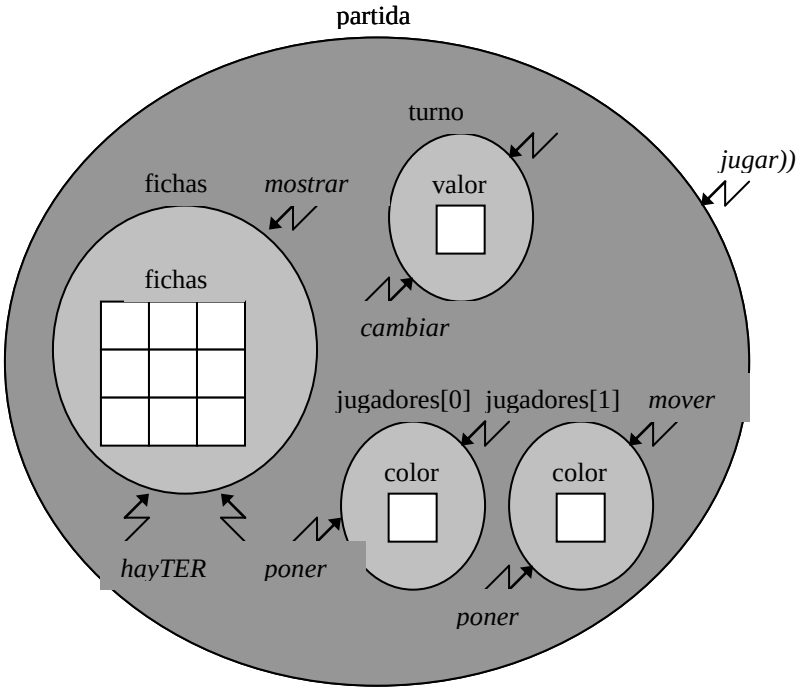
Por otro lado, la **visión dinámica** del código muestra un universo jerarquizado de objetos construido "automáticamete" gracias al desencadenamiento de instaciaciones. Estas, a su vez, que colaboran entre sí gracias al desencadenamiento de mensajes.

Para la primera solución del programa de las "Tres en Raya":

- *main*:
  - crea un objeto *partida* que:
    - crea un objeto *tablero* que:
      - crea nueves *fichas* caracter;
    - crea dos objetos *jugadores*[i] que:
      - crean dos *color* carácter;
    - crea un objeto *turno* que:
      - crea un *valor* entero.
  - lanza el mensaje *jugar* a *partida* que:
    - lanza el mensaje *mostrar* a *tablero* que:

- lanza muchos mensajes *out* a *gestorIO*;
- el mensaje *poner* a *jugadores[0]* que:
  - crea un objeto *coordenada* que:
    - crea una *fila* entera;
    - crea una *columna* entera;
  - lanza el mensaje *recoger* a *coordenada*;

...



3.2. Relaciones entre Clases.

Con lo anteriormente expuesto, se considerará que existe una **colaboración entre dos objetos** cuando un objeto lanza mensajes a otro objeto; por otro lado, se considerará que existe una **relación entre dos clases** si dos objetos de respectivas clases colaboran entre sí.

De esta forma, de forma paralela al gráfico de la jerarquía de clases de las "Tres en Raya", tenemos que:

- la clase *TER* está relacionada con *Tablero, Jugador* y *Turno*; pero no con *Coordenada*;
- la clase *Jugador* está relacionada con *Tablero* y *Coordenada* y nada más;
- la clase *Tablero* está relacionada con *Coordenada* y nada más;
- la clase *Coordenada* no está relacionada con ninguna;
- la clase *Turno* no está relacionada con ninguna;

De alguna forma, los conceptos de colaboración entre objetos y relación entre clases se sitúan en dos planos proyectables: uno tangible para los objetos y el otro intangible para las clases; uno particular entre 2 objetos y el otro genérico entre 2 clases.

Una posible analogía sería que, de forma genérica, el *Profesor* se relaciona con el *Alumno*. Esto permite que objetos concretos como *donAurelio* y *elSapo* colaboren con objetos concretos como *elChato, Ramirez* y *cuatroOjos*.

- o -

A continuación, se exponen tres tipos diferentes de relaciones entre clases: **composición**, **asociación** y **uso**. Posteriormente, se enfrentarán comparativamente, a través de sus características, para una mejor comprensión. Por último, se establecerán las vías oportunas para "aterrizar" el diseño de las relaciones de clases al código concreto de un programa orientado a objetos.

Por tanto, el estudio de las diferentes tipos de relaciones entre clases abre un abanico para nuevas soluciones de un mismo programa. Tras el estudio teórico sobre los tipos de relaciones entre clases se propondrán diseños alternativos sobre el ejercicio anterior de las "Tres en Raya".

3.1.1. Relación de Composición.

Es la relación que se constituye entre **el todo y la parte**. Se puede determinar que existe una relación de composición entre la clase A, el todo, y la clase B, la parte, si un objeto de la clase A "*tiene un*" objeto de la clase B. La relación de composición no abarca simplemente cuestiones físicas (libro -todo- y páginas -parte-) sino, también, a relaciones lógicas que respondan adecuadamente al todo y a la parte como "*contiene un*" (aparato digestivo -todo-

y bolo alimenticio -parte-), *"posee un"* (propietario -todo- y propiedades -parte-), etc.

La colaboración vendrá dada a través de la delegación oportuna, del todo a las partes, de ciertas responsabilidades particulares que permitan llevar a cabo la responsabilidad global: almacenar ciertos datos de todo el conjunto de datos, operar sobre éstos para realizar la parte correspondiente de una operación requerida al todo, etc. (una página sirve de índice del resto de las páginas del libro; el bolo alimenticio transportar los nutrientes a otras zonas del aparato digestivo; una propiedad produce beneficios si el propietario la alquila; etc.)

Al ser un objeto de la clase B parte del objeto de las clase A, la responsabilidad de manejar este objeto de la clase B será, fundamentalmente, de la clase A, respetándose el principio de encapsulación y modularidad por parte de otras clases (lo que le ocurra a las páginas depende, fundamentalmente, de lo que le ocurra al libro; lo que le ocurra al bolo alimenticio depende, fundamentalmente, del aparto digestivo; lo que le ocurra a las propiedades depende, fundamentalmente, del propietario; etc.).

Esta relación establece una jerarquía por grado de composición entre las clases que **constituye un esqueleto fundamental** de un programa orientado a objetos. Esta jerarquía debe responder a un grafo dirigido acíclico cuyos nodos representan a las clases y cuyos arcos representan las relaciones de composición que parte de la clase "todo" hasta la clase "parte".

3.1.2. Relación de Uso.

Es la relación que se establece **momentáneamente entre un cliente y cualquier servidor.** Se puede determinar que existe una relación de uso entre la clase A, el cliente, y la clase B, el servidor, si un objeto de la clase A disfruta de los servicios de un objeto de la clase B (mensajes lanzados) para llevar a cabo la responsabilidad del objeto de la clase A en un momento dado sin dependencias futuras. La relación de uso no abarca simplemente cuestiones tangibles (ciudadano -cliente- y autobús -servidor-) sino, también a cuestiones lógicas que respondan adecuadamente al cliente y al servidor momentáneo cualquiera que sea como "*goce*" (espectador -cliente- y actor -servidor-), "*beneficio*" (viajante -cliente- y motel -servidor-), etc.

La colaboración vendrá dada a través de la delegación oportuna, del cliente al servidor, de ciertas responsabilidades particulares que permitan llevar a cabo una parte de su responsabilidad: almacenar ciertos datos momentáneamente, operar sobre éstos para realizar la parte oportuna de una operación requerida en cierto momento, etc. (un autobús desplaza a un ciudadano que desea ir al cine; un actor representa un papel para conmover a un espectador; un motel alberga a un viajante para descansar; etc.)

Al ser un objeto de la clase B usado por un objeto de la clase A, la responsabilidad de manejar este objeto de la clase B no tiene porqué depender, únicamente, de la clase A (lo que le ocurra al autobús no tiene porqué depender, únicamente, del ciudadano; lo que le ocurra al actor no tiene porqué depender, únicamente, del espectador; lo que le ocurra al motel no tiene porqué depender, únicamente, del viajante; etc.).

Esta relación establece una jerarquía por grado de uso entre las clases que **no constituye un esqueleto fundamental** de un programa orientado a objetos. Esta jerarquía debe responder a un grafo dirigido cuyos nodos representan a las clases y cuyos arcos representan las relaciones de uso que parte de la clase "cliente" hasta la clase "servidor"; por tanto, es posible la existencia de ciclos en el grafo: un ciclo entre dos clases determinará el uso de los objetos de una clase por parte de los objetos de la otra, alternando las papeles de cliente y servidor (un camarero "usa" a un policía tras un atraco y un policía "usa" a un camarero para comer; etc.) .

3.1.3. Relación de Asociación.

Es la relación que **perdura entre un cliente y un servidor determinado.** Se puede determinar que existe una relación de asociación entre la clase A, el cliente, y la clase B, el servidor, si un objeto de la clase A disfruta de los servicios de un objeto determinado de la clase B (mensajes lanzados) para llevar a cabo la responsabilidad del objeto de la clase A en diversos momentos creándose una dependencia del objeto servidor. La relación de asociación no abarca simplemente cuestiones tangibles (procesador -cliente- y memoria -servidor-) sino, también a cuestiones lógicas que respondan adecuadamente al cliente y al servidor determinado como "*provecho*" (socio -cliente- y club -servidor-), "*beneficio" (*empresa -cliente- y banca -servidor-), etc.

La colaboración vendrá dada a través de la delegación oportuna, del cliente al servidor, de ciertas responsabilidades particulares que permitan llevar a cabo parte de su responsabilidad: almacenar ciertos datos por necesidades posteriores, operar sobre éstos para realizar las operaciones requeridas en diversos momentos, etc. (la memoria almacena información cuando el procesador edita, compila y ejecuta programas; el club presta instalaciones al socio para jugar, descansar y reunirse; la banca atiende al empresario cuando solicita y rescinde créditos; etc.)

Al estar un objeto de la clase B asociado a un objeto de la clase A, la responsabilidad de manejar este objeto de la clase B no tiene porqué depender, únicamente, de la clase A (lo que le ocurra a la memoria no tiene porqué depender, únicamente, del procesador; lo que le ocurra al club no tiene porqué depender, únicamente, del socio; lo que le ocurra a la banca no tiene porqué depender, únicamente, del empresario; etc.).

Esta relación establece una jerarquía por grado de asociación entre las clases que **constituye un esqueleto importante** de un programa orientado a objetos. Esta jerarquía debe responder a un grafo dirigido cuyos nodos representan a las clases y cuyos arcos representan las relaciones de asociación que parte de la clase "cliente" hasta la clase "servidor"; por tanto, es posible la existencia de ciclos en el grafo: un ciclo entre dos clases determinará la asociación bidireccional entre los objetos de dos clases (un matrimonio, esposa y esposo, están "asociados" bidireccionalmente en el mantenimiento de la familia; etc.) .

- o O o -

Una vez expuestas las diferentes relaciones que se establecen entre clases, se estudiarán atendiendo a las siguientes **características**:

VISIBILIDAD: carácter privado o público de la colaboración entre dos objetos.

Un ejemplo válido del carácter público de una colaboración: un profesor (objeto agente) debe colaborar con la red del centro de cálculo (objeto pasivo) para dar alcance a los alumnos (otros objetos agentes) de un compilador de PIIPOO.

Un ejemplo válido del carácter privado de una colaboración: el medio (objeto pasivo) por el que un profesor (objeto agente) hace llegar el compilador a la red: ¿mediante un disquete? ¿mediante Internet? ¿mediante la red de área local? etc.; nadie (otros objetos) lo conoce y a nadie le interesa.

TEMPORALIDAD: mayor o menor duración de la colaboración entre dos objetos.

Un ejemplo válido de escasa duración en una colaboración: un profesor (objeto pasivo) colabora momentáneamente con un alumno (objeto agente) para explicarle una duda y, posteriormente, no volver a colaborar jamás.

Un ejemplo válido de larga duración en una colaboración: un profesor (objeto pasivo) colabora por mucho tiempo, incluso años, con el responsable de la asignatura (objeto agente) realizando distintas tareas de las asignatura: en febrero, junio y septiembre redactar propuestas de examen, preparar los capítulos y ejercicios que le correspondan para el año siguiente, etc.

VERSATILIDAD: intercambiabilidad de los objetos en la colaboración con otro objeto.

Un ejemplo válido de posible intercambiabilidad en una colaboración: un profesor (objeto agente) colabora con tizas (objetos pasivos) en su quehacer diario: dar clase, discutir con otros profesores, etc.; la tiza particular con que colabore un profesor en un momento dado, es completamente indiferente para la responsabilidad que tenía asumida.

Un ejemplo válido de ausencia de intercambiabilidad en una colaboración: un profesor (objeto agente) colabora con un ordenador (objeto pasivo) en su quehacer diario: redactar apuntes, exámenes, etc.,

gestionar notas, listados, actas, etc.; en términos generales, el ordenador no puede ser cualquiera, tiene que ser el que tenga cargado en el disco duro toda la información necesaria y los programas que la manejen, de hecho, cambiar de ordenador suele significar "perder" varios días de trabajo.

A través de estas características en una colaboración particular entre objetos, se puede determinar qué relación entre sus clases es la más adecuada cuando responde conceptualmente a más de una relación: ¿un paciente está asociado o usa un médico? ¿un motor es parte o está asociado a un coche?:

- la **relación de composición** responde eminentemente a una visilidad privada; frecuentemente su temporalidad no es momentánea y su versatilidad es nula.
- la **relación de uso** responde eminentemente a una temporalidad momentánea y total versatilidad; en cuanto a la visibilidad es indistintamente privada o pública.
- la **relación de asociación** responde eminentemente a una visibilidad pública y temporalidad no momentánea; en cuanto a la versatilidad es frecuentemente reducida.

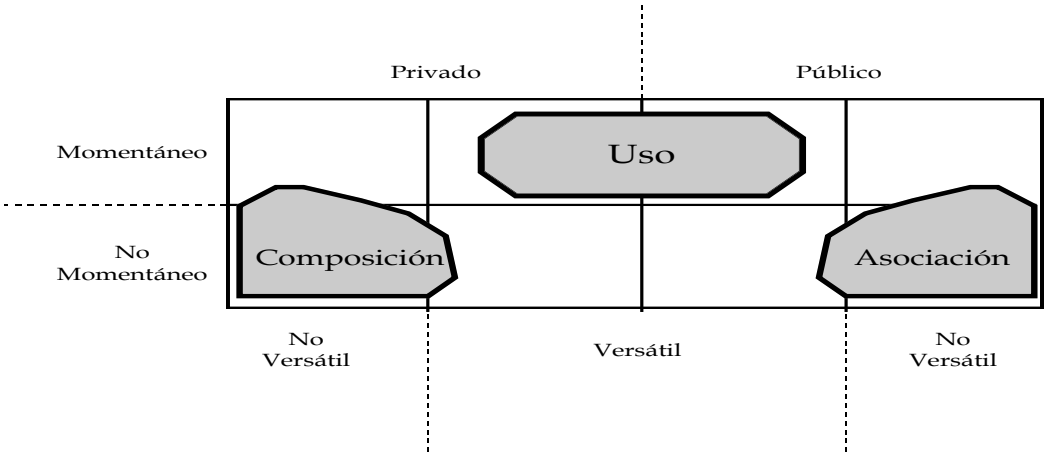


Figura 3. 2 Comparativa de las Relaciones entre clases.

Sin duda, falta mencionar el factor más determinante a la hora de decidir la relación entre las clases. El **contexto** en el que se desenvuelvan los objetos, determinará de forma categórica qué grados de visibilidad, temporalidad y versatilidad se producen en su colaboración.

Por tanto, retomando ejemplos anteriores, si el contexto de los objetos paciente y médico es un hospital de urgencias la relación se decantaría por un uso mientras que si es el médico de cabecera que conoce su historial y tiene pendiente algún tratamiento, la relación se inclinaría a una asociación; si el contexto de los objetos motor y coche es un taller mecánico (se accede al motor de un coche, se cambian motores a los coches, etc.) la relación se inclinaría a una asociación, mientras que si el contexto es la gestión municipal del parque automovilísticos (se da de alta y de baja al coche, se denuncia al coche, etc. y el

motor se responsabiliza de ciertas características que dependen del ministerio de industria como su potencia fiscal, etc.) la relación se inclinaría a una composición.

Finalmente, señalar que no existe para toda colaboración un relación ideal categórica. Es muy frecuente que sean varias relaciones candidatas, cada una con sus ventajas y desventajas. Por tanto, al existir diversas alternativas, será una decisión de ingeniería, un compromiso entre múltiples factores no cuantificables: costes, modularidad, legibilidad, eficiencia, etc., la que determine la relación final.

El objetivo principal de establecer relaciones entre clases es reconocer, estudiar, analizar, etc. la forma en qué colaboran los objetos, siendo secundario, e imposible, determinar a qué relación responden exactamente en todas las ocasiones. Prueba de ello es cómo cierra Rumbaugh una comparación entre las relaciones de agregación (como él denomina la composición) y asociación: *"La decisión de utilizar una agregación es discutible y suele ser arbitraria. Con frecuencia, no resulta evidente que una asociación deba ser modelada en forma de agregación, En gran parte, este tipo de incertidumbre es típico del modelado; este requiere un juicio bien formado y hay pocas reglas inamovibles. La experiencia demuestra que si uno piensa cuidadosamente e intenta ser congruente la distinción imprecisa entre asociación ordinaria y agregación no da lugar a problemas en la práctica."* [Rumbaugh, 91].

Una vez que se conoce cómo colaboran dos objetos, se facilitarán las decisiones de cómo implantar dicha colaboraciones. Normalmente, quien pasa por alto este diseño de las relaciones entre clases se encuentra violando principios de encapsulación o desarrollando una maraña incomprensible de mensajes hinchados de objetos o, sencillamente, sin ningún camino abierto para poder escribir una sola sentencia más. Si no se diseñan las relaciones entre clases, no se pondrá a unos objetos al alcance de otros para que realicen las colaboraciones necesarias.

- o -

Toda la exposición de relaciones entre clases se ha referido únicamente a relaciones binarias (una clase se relaciona con otra por composición, asociación o uso), pero esto no anula la posibilidad de que un objeto colabore con muchos otros muchos objetos dos a dos y, por tanto, una clase se relaciona con otras clases .

Por ejemplo: una persona está asociado a su cónyuge, posee un coche y una lavadora y usa un ordenador (¡qué triste!); una empresa posee ordenadores para desarrollar sus aplicaciones y está asociado a otro ordenador proveedor de Internet; etc.

- o O o -

Al igual que no existe una fórmula para determinar cual es la relación dada en una colaboración entre objetos, no existen fórmulas para **traducir la relación escogida a un código particular**. Por tanto, sólo se establecerán pautas

de actuación que ayudarán al programador a formar un esqueleto del programa (no todos los detalles) en un amplio abanico de casos pero será responsabilidad última del programador saber cuándo romper las reglas:

- un relación de composición debe traducirse en que la clase "todo" contemple como atributo un referencia y creación de un objeto de clase "parte". Por tanto, el objeto "parte" nace con el nacimiento del objeto "todo" de forma encapsulada.

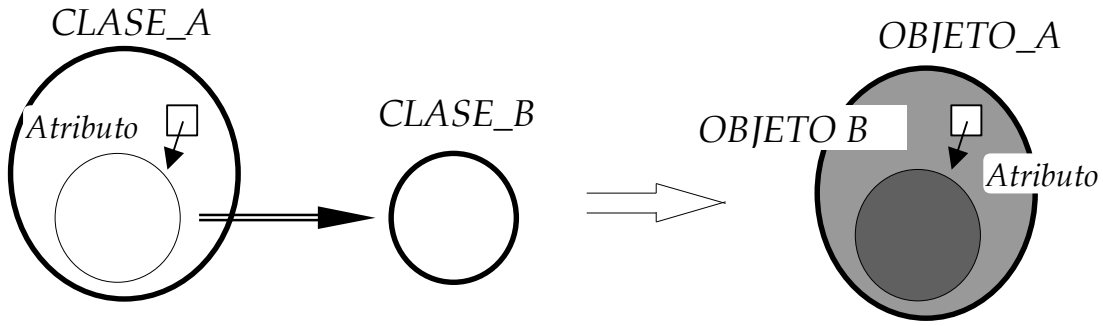


Figura 3. 3. Relación de Composición.

- una relación de asociación debe traducirse en que la clase "cliente" contemple como atributo una referencia pero sin creación de un objeto de la clase "servidor". Posteriormente, el objeto cliente "recibirá" (a través de un constructor o un mensaje) la dirección del objeto servidor (público) que "viene" del exterior del objeto cliente.

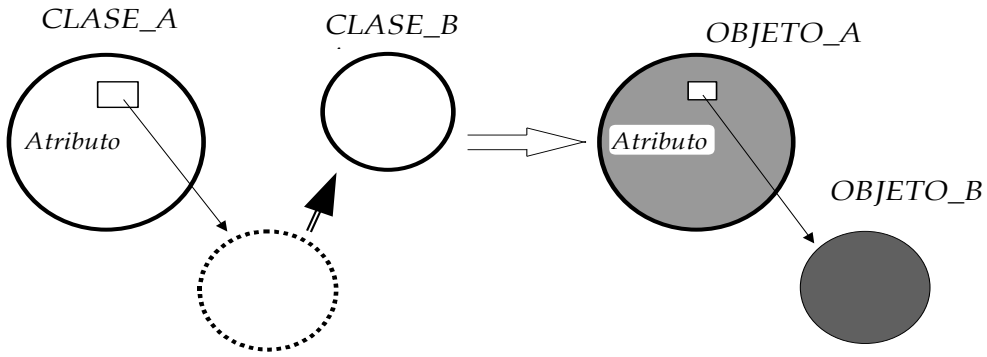


Figura 3.4. Relación de Asociación.

- una relación de uso debe traducirse en que la clase "cliente" contemple como parámetro o valor devuelto de un método un objeto de la clase "servidor" si la colaboración es pública; mientras que debe traducirse como objeto local de un método si la colaboración es privada.

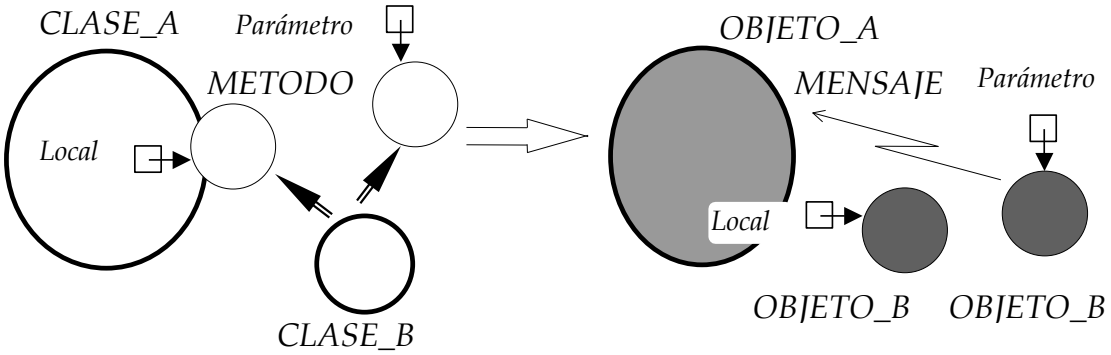


Figura 3.5. Relación de Uso.

Observando detenidamente las tres reglas dadas, se observa que ajustan la naturaleza de las características de temporalidad, versatilidad y visibilidad de cada relación al caso más próximo de los citados en la ley estricta de Demeter, (excepto *this* entendida como una autocolaboración obvia): un atributo es privado; un objeto local es eminentemente momentáneo; etc.

Es lógico que estas reglas no abarquen otras posibilidades que no se contemplan en la Ley Estricta de Demeter porque, en tal caso, no podrían emitirse mensajes de un objeto a otro y, por tanto, no podrían colaborar.

- o -

La lectura más simplista que se puede realizar de todo este camino emprendido (desde las colaboraciones entre objetos a relaciones entre clases y, finalmente, a reglas de traducción a código) sería recorrerlo en el sentido inverso.

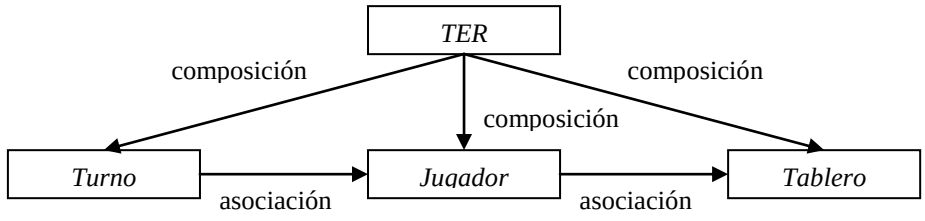
Por ejemplo, como una clase *A* tiene un método con un parámetro que es una referencia a un objeto de la clase *B*, se deduce que la clase *A* "usa" a la clase *B*. Nada más lejos de la realidad.

- o O o -

A continuación se mostrará la primera solución del problema de las "Tres en Raya" (clases *TER, Jugador, Tablero, Coordenada* y *Turno*) contemplando los 3 tipos de relaciones entre clases. El momento de establecer las relaciones a través del método de desarrollo será, cada vez que surja una nueva clase, bien como atributo, parámetro o local auxiliar, frente a todas las existentes ya existentes.

1.1) Tomar una clase: *TER*.
2.1) Determinar el comportamiento: Idem
3.1) Determinar los atributos con sus respectivas clases; Idem pero añadiendo el diseño de las relaciones entre las clases:

- *TER-Tablero*. Se propone composición: una partida de "Tres en Raya" tiene un tablero; es una relación privada (el tablero lo gestiona únicamente la partida), es duradera (colaboran a lo largo de toda la vida de la partida) y es poco versatil (una vez determinado un tablero no es indiferente cambiarlo por otro).
- *TER-Jugador*. Se propone composición: una partida de "Tres en Raya" tiene dos jugadores; es una relación privada (los jugadores los gestiona únicamente la partida), es duradera (colaboran a lo largo de toda la vida de la partida) y es poco versatil (una vez determinados unos jugadores no es indiferente cambiarlo por otro).
- *TER-Turno*. Se propone composición: una partida de "Tres en Raya" tiene un turno; es una relación privada (el tablero lo gestiona únicamente la partida), es duradera (colaboran a lo largo de toda la vida de la partida) y es poco versatil (una vez determinado un turno no es indiferente cambiarlo por otro).
- *Jugador-Tablero*. Se propone asociación: cada jugador pone y mueve fichas sobre un tablero compartido con otro jugador; es una relación pública (el tablero lo gestiona un jugador, el otro y la partida para ver si se termina con "Tres en Raya"), es duradera (colaboran a lo largo de la vida del jugador) y es poco versatil (una vez determinado un tablero no es indiferente cambiarlo por otro).
- *Turno-Jugador*. Se propone asociación: el turno controla qué jugador pone o mueve según la historia; es una relación pública (los jugadores los gestiona el turno y la partida para que "canten" victoria), es duradera (colaboran a lo largo de la vida del turno) y es poco versatil (una vez determinados unos jugadores no es indiferente cambiarlo por otros).
- *Turno-Tablero*. Se considera inexistente de forma directa pero sí de forma transitiva a través del jugador.




```
class TER
{
private final Tablero tablero=new Tablero();
private final Jugador jugadores[]=new Jugador[2];
private final Turno turno=new Turno(jugadores); // asociacion

public TER ()
{
     Jugador.en(tablero); // asociacion
     jugadores[0]=new Jugador('o');
     jugadores[0]=new Jugador('x');
}
```

4.1) Codificar métodos:

```
public void jugar ()
{
      tablero.mostrar();
      for (int i=0; i<5 ; i++)
      {
            turno.toca().poner();
            turno.cambiar();
            tablero.mostrar();
      }
      if (tablero.hayTER())
            turno.noToca().victoria();
      else
      {
            turno.toca().poner();
            tablero.mostrar();
            while (!tablero.hayTER())
            {
                  turno.cambiar();
                  turno.toca().mover();
                  tablero.mostrar();
            }
            turno.toca().victoria();
      }
}

public static void main (String arg[])
{
   TER partida=new TER();
   partida.jugar();
}

}
```

5.1) Recapitular:

```
class Tablero
{
public void mostrar () {...}
public boolean hayTER () {...}
}
```

```
class Jugador
{
public static void en (Tablero tablero) {...}
public Jugador (char color) {...}
public void poner () {...}
public void mover () {...}
public void victoria () {...}
}
```

```
class Turno
{
public Turno (Jugador jugadores[]) {...}
public void cambiar() {...}
public Jugador toca() {...}
public Jugador noToca() {...}
}
```

1.2) Tomar una clase: *Jugador* de la fase 5.1)
2.2) Determinar el comportamiento: de la fase 5.1)
3.2) Determinar los atributos con sus respectivas clases; para *poner*, *mover* y cantar *victoria* un *Jugador* solo precisa saber el color de sus fichas y, común a todo jugador, sobre qué tablero.

```
class Jugador
{

private char color;
private static Tablero tablero; // asociacion

public static void en (Tablero tablero) {...}
public Jugador (char color) {...}
public void poner () {...}
public void mover () {...}
public void victoria () {...}
}
```

4.2) Codificar métodos:

```
public static void en (Tablero tablero)
{
     Jugador.tablero = tablero;
}

public Jugador (char color)
{
  this.color=color;
}
public void poner ()
{
     (new GestorIO()).out("juega: "+color);
     Coordenada destino=new Coordenada();
     do
     {
          destino.recoger("Coordenada destino de puesta");
     } while(!destino.valida() || tablero.ocupado(destino));
     tablero.poner(destino,color);
}
```

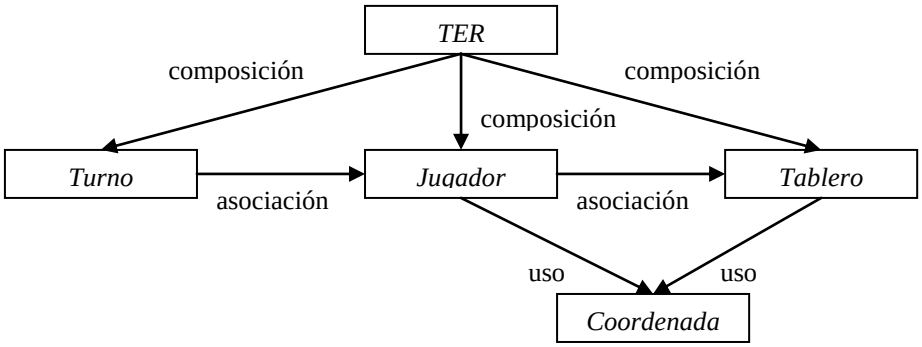
```
public void mover ()
{
      Coordenada origen=new Coordenada();
      do
      {
            origen.recoger("Coordenada origen de movimiento");
      } while (!origen.valida() || tablero.vacio(origen));
      tablero.sacar(origen);
      this.poner(tablero);
}

public void victoria ()
{
      GestorIO gestorIO=new GestorIO();
      gestorIO.out ("las "+color+" han ganda....");
}

}
```

5.2) Recapitular. Surge una nueva clase a relacionar.

- *Jugador-Coordenada*. Se propone uso: cada jugador necesita objetos de la clase *Coordenada* cada vez que pone y mueve fichas; es una relación privada (la coordenada la gestiona un jugador y nadie más), es momentanea (el jugador colabora con la coordenada cuando pone/mueve sin necesidad de memorizarla para posteriores acciones) y es muy versatil (cada vez que pone/mueve precisa distintos objetos *Coordenada* sin ligarse a ninguno).
- *Tablero-Coordenada*. Se propone uso: el tablero necesita objetos de la clase *Coordenada* cada vez que pone y mueve fichas; es una relación pública (la coordenada viene dada del exterior del tablero, en particular el jugador), es momentanea (el tablero colabora con la coordenada cuando pone/mueve sin necesidad de memorizarla para posteriores acciones) y es muy versatil (cada vez que pone/mueve precisa distintos objetos *Coordenada* sin ligarse a ninguno).




```
class Coordenada
{
public void recoger (String titulo) {...}
public boolean valida () {...}
}
```

```
class Tablero
{
// fase recapitulación 5.1
public void mostrar () {...}
public boolean hayTER () {...}

// fase recapitulación 5.2
public boolean ocupado (Coordenada coordenada) {...}
public void poner (Coordenada coordenada, char color) {...}
public boolean vacio (Coordenada coordenada) {...}
public void sacar (Coordenada coordenada, char color) {...}
}
```

1.3) Tomar una clase: *Tablero* de la fase 5.1)
2.3) Determinar el comportamiento: de las fases 5.1) y 5.2)
3.3) Determinar los atributos con sus respectivas clases; Idem.
4.3) Codificar métodos: Idem.
5.3) Recapitular. Idem

1.4) Tomar una clase: *Coordenada* de la fase 5.2)
2.4) Determinar el comportamiento: de las fases 5.2) y 5.3)
3.4) Determinar los atributos con sus respectivas clases. Idem.
4.3) Codificar métodos:
5.4) Recapitular: no surgen ni nuevas clases ni nuevos métodos de clases ya recapituladas.

1.5) Tomar una clase: *Turno* de la fase 5.1)
2.5) Determinar el comportamiento: de las fases 5.1)
3.5) Determinar los atributos con sus respectivas clases; para *cambiar* y obtener a quien *toca* y el contrario solo se precisa memorizar las referencias de los dos jugadores y un valor alternativo entre 1 y 2.

```
class Turno
{
private int valor=0;
private Jugador jugadores[];

public Turno (Jugador jugadores[]) {...}
public void cambiar() {...}
public Jugador toca() {...}
public Jugador noToca() {...}
```

4.5) Codificar métodos:

```
public Turno (Jugador jugadores[])
{
      this.jugadores = jugadores;
}

public void cambiar()
{
      this.valor=(this.valor+1)%2;
}

public Jugador toca()
{
      return jugadores[valor];
}
public Jugador noToca()
{
      return jugadores[(valor+1)%2];
}

}
```

5.5) Recapitular: Idem.

3.3. Paquetes.

Este capítulo
trabajar en paralelo
re compilar todo
encontrar código
reusabilidad

solución: fisico-lógico
colisión y nombrado
classpath

3.3.1. Creación e Importación.

3.3.1. Accesibilidad en Paquetes.

Clases publica/no publica
Atributos y métodos de paquete

(Ej: Tres en Raya)
(Ej: biblioteca)