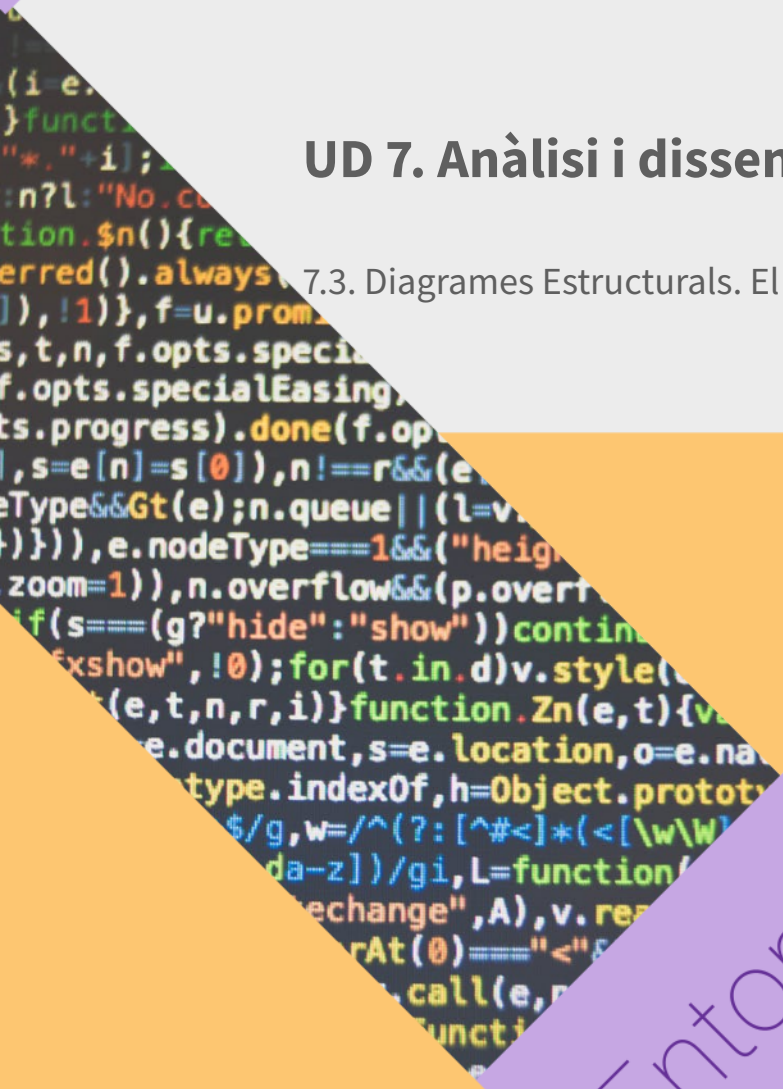
Curs 20-21

UD 7. Anàlisi i disseny orientat a objectes

7.3. Diagrames Estructurals. El diagrama de classes.



Entorns de desenvolupament

IES Jaume II El Just
Tavernes de la Valldigna

Continguts

1. Diagrames estructurals i el diagrama de classes

Els diagrames estructurals o d'estructura són aquells que **representen l'estructura estàtica dels objectes del sistema**. Entre ells, el diagrama de classes és el més conegut i utilitzat.

El diagrama de classes és un diagrama purament orientat al model de programació orientat a objectes, ja que defineix les classes que s'utilitzaran quan en la fase de construcció i la manera en què es relacionen aquestes. Com veurem, s'assembla molt al diagrama Entitat-Relació (E/R), amb la diferència que el diagrama de classes també representa la funcionalitat que ofereixen les classes. Ambdós són models de dades lògics d'un sistema.

2. Elements d'un diagrama de classes

El diagrama està format principalment per *classes* i les *relacions* entre elles, així com per *interfícies*. Veiem cadascuna d'elles.

2.1. Classes

Les classes són l'element principal del diagrama i representa, com el seu nom indica, una classe dins del paradigma d'orientació a objectes. Normalment s'utilitzen per a **representar conceptes** o **entitats del *negoci***.

Una classe **defineix un grup d'objectes que comparteixen característiques comunes**. La manera més ràpida per a trobar classes sobre un enunciat o en general, sobre un tema concret és *buscar els substantius que apareixen en aquest*. Per exemple: *Persona*, *Missatge*, *Producte*, etc.

Una classe es compon de tres elements o seccions principals:

- El **nom de la classe**,
- Els **atributs**, que són variables associades a la classe i representen la part estàtica d'aquesta,
- Els **mètodes**, que són funcions associades a la classe, i representen la seua part dinàmica.

La representació d'una classe és una caixa dividida en tres zones, de la següent manera:

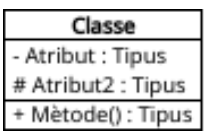


Figura 1: Representació d'una classe

- A la part superior s'indica **el nom de la classe**. Si es tracta d'una classe *abstracta* (és a dir, no es pot instanciar directament, sinò que serveix per definir altres classes), aquest nom s'indica *en cursiva*.
- La part central conté els atributs de la classe, un per línia i amb els següents formats:

```
visibilitat nom_atribut : tipus = valorInicial { propietats }
```

```
visibilitat nom_atribut : tipus
```

- La zona inferior, finalment, inclou els mètodes de la classe, i s'expressen de forma semblant:

```
visibilitat nom_mètode { *paràmetres } : TipusRetorn { propietats }
```

```
visibilitat nom_mètode: TipusRetorn
```

Quan parlem de la visibilitat d'un atribut o mètode, ens referim a si aquest és *públic*, *privat* o *protegit*. Això ho expressem amb:

- **(+) Públic.** L'atribut o mètode és accessible des de qualsevol lloc de l'aplicació.
- **(-) Privat.** L'atribut o mètode només és accessible dins la mateixa classe.
- **(#) Protegit.** L'atribut o mètode només és accessible dins la mateixa classe o les seues *classes derivades*.

Habitualment, per tal d'aprofitar els beneficis de l'encapsulament i aïllament de codi que proporciona la programació orientada a objectes, els atributs es solen declarar com a *privats*, i només en cas que es permeta l'accés a ells des de fora, es farà a través de *mètodes públics*. Aquests mètodes es coneixen com *setters* (estableixen el valor d'un atribut) i *getters* (consulten el valor d'un atribut) o mètodes *accessors*.

Quan un atribut o mètode és **estàtic** es representa al diagrama subratllant el seu nom. Recordem que els atributs o mètodes estàtics són aquells que *resideixen en la classe* i no en els objectes que s'instancien d'ella.

Exemple: Veiem un xicotet exemple d'una nau en un videojoc:

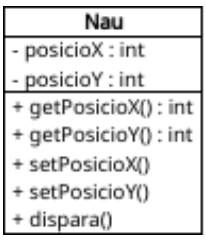


Figura 2: Exemple de classe

Com veiem, es tracta d'una classe `Nau`, amb dos atributs privats: `posicioX` i `posicioY`. Per tal d'accedir a ells, es disposa dels mètodes `getPosicioX()` i `getPosicioY()`. A més, la classe defineix el mètode `dispara()`.

La traducció a Java d'aquesta classe seria literal, incloent els atributs i mètodes corresponents, a més del mètode *constructor*:

```
1  public class Nau {
2
3    /* Atributs */
4    private int posicioX;
5    private int posicioY;
6
7    /* Constructor */
8    public Nau () { };
9
10   /* Métodes d'accés */
11
12   public void setPosicioX (int newVar) {
13     posicioX = newVar;
14   }
15
16   public int getPosicioX () {
17     return posicioX;
18   }
19
20   public void setPosicioY (int newVar) {
21     posicioY = newVar;
22   }
23
24   public int getPosicioY () {
25     return posicioY;
26   }
27
28   /* Altres métodes */
29   public void dispara()
30   {
31   }
32 }
```

Cal tindre en compte, que quan utilitzem eines CASE que ens ajuden a crear el codi a partir dels diagrames, és possible que ens generen automàticament els mètodes *accessors* als atributs, de manera que no siga necessari incloure'ls al diagrama.

2.2. Relacions

Les diferents classes que conformen una aplicació interaccionen entre elles, a través de diferents tipus de *relacions*. Generalment, les relacions seran entre dos o més classes, tot i que de vegades, una classe pot establir relacions amb ella mateixa (dependència *reflexiva*).

Per tal de representar una relació entre classes en un diagrama de classes fem ús d'una línia que les uneix. Aquesta línia serà diferent segons el tipus de relació.

Les relacions en el diagrama de classes tenen diverses propietats, que podrem representar més o menys segons el nivell de detall que volgam aportar. Aquestes **propietats de les relacions** són:

- **Multiplicitat**: Indica el nombre d'elements que participen en la relació. Pot ser un número, un rang.. Si es tracta d'un número qualsevol, podem utilitzar $\star$o n.
- **Nom de l'associació**, que ens ajuda a entendre la relació entre les classes.

Tipus de relacions

Les relacions, com hem dit poden ser de diferents tipus. En un diagrama de classes, podem trobar-nos els següents tipus de relacions:

- Associació.
- Agregació.
- Composició.
- Dependència.
- Herència.

Vegem cadascun d'aquests tipus de relacions.

Associació És la relació més comuna i s'utilitza per a representar *dependència semàntica*. Es representa amb una simple línia contínua que uneix les classes que estan incloses en l'associació.

Veiem un exemple:

Figura 3: Exemple d'associació

En aquest exemple, tenim definides dues classes: `Nau` i `Pilot`. La relació entre elles és que un `Pilot`, `Pilota`una `Nau`. Com veiem, la relació té una multiplicitat 1:1, és a dir, en un moment donat, un *pilot* està *pilotant* una nau, i una nau és *pilotada* per un pilot.

Tal i com l'hem expressada, es tracta d'una relació *bidireccional*. Això vol dir que des de `Nau`, podem *navegar* a `Pilot`, i a l'inrevès. En altres paraules, donada una nau podem saber qui la pilota, i donat un pilot podem saber quina nau pilota.

Aquestes relacions d'associació, també admeten **associacions unidirecconals**, que restringeixen la direcció de navegació. Per expressar això, farem ús d'una fletxa que indicarà el sentit d'aquesta associació:

Figura 4: Exemple d'associació unidireccional

Com hem comentat, les relacions són un concepte semàntic, i sovint no tenen una *equivalència* exacta quan ho passem a un llenguatge de programació. Per exemple… com expressaríeu les dues relacions anteriors amb Java? El mecanisme que tenim a Java per *relacionar* classes és utilitzar les classes com a tipus de dades, i definir objectes d'aquest tipus dins d'altres. Com veurem, totes les relacions, quan les passem a Java es basaran en aquest mecanisme. Així doncs, per tal d'establir la relació direccional anterior amb Java, inclouriem un objecte de tipus *Pilot* dins la definició de *Nau*:

```java
/* Classe Pilot */

public class pilot {

  // Atributs
  private String Nom;

  // Constructor
  public pilot () { };

  // Métodes accessors
  public void setNom (String newVar) {
    Nom = newVar;
  }

  public String getNom () {
    return Nom;
  }

}
```

```java
/* Classe Nau */
public class nau {

  // Atributs
  private int posicioX;
  private int posicioY;
  private pilot pilotActual;

  // Constructor
  public nau () { };

  // Mètodes accessors
  public void setPosicioX (int newVar) {
    posicioX = newVar;
  }

  public int getPosicioX () {
    return posicioX;
  }

  public void setPosicioY (int newVar) {
    posicioY = newVar;
  }

  public int getPosicioY () {
    return posicioY;
  }

  public void setPilot (int newPilot) {
    pilotActual = newPilot;
  }

  public pilot getPilot () {
    return pilotActual;
  }

  // Altres mètodes
  public void dispara()
  {
  }
```

En cas que volguèrem que l'associació fora bidireccional, hauriem d'incloure també un objecte de tipus *Nau* dins de *Pilot*. L'inconvenient d'això és que caldria *programàticament* estar pedent de quan caviem aquesta associació que es modifiquen els dos objectes relacionats. En estos casos, el més pràctic serà determinar quin serà el sentit *normal* de les relacions, i fer ús d'associacions direccionals incloent només un objecte d'una classe dins l'altra. A l'exemple anterior, ens serà més útil conéixer quin pilot està pilotant una nau en un moment donat, que no quina nau pilota un pilot en un moment donat.

Agregació Les relacions d'agregació indiquen quan un objecte està format per alters objectes. És a dir, un objecte és part d'un altre, encara que té existència per sí mateix.

Aquesta relació es representa amb una línia amb un rombe en la part de la classe que és una agregació d'altra classe.

Per exemple, al cas de la nau, una nau pot estar formada per un propulsor, una cabina i un canó:

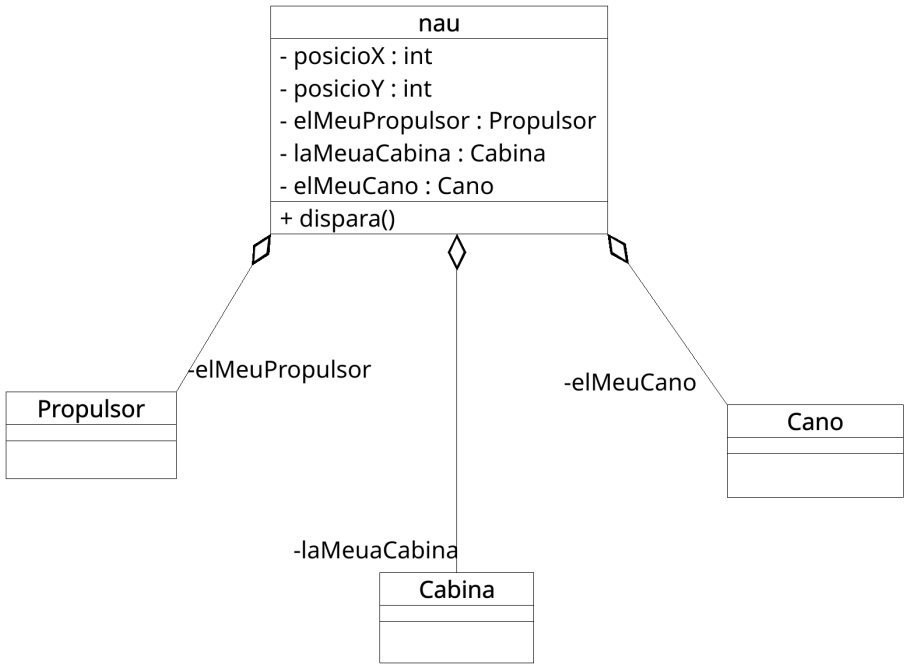


Figura 5: Exemple d'agregació

Com veiem, la forma de representar açò en Java, serà incorporar com a atributs tres objectes de tipus *Cabina*, *Propulsor* i *Canó* dins la classe *Nau*.

Composició Es tracta d'una relació semblan a l'agregació, però amb un valor semàntic més fort, i és que els elements que en formen part, no tenen sentit sense l'objecte que componen: Quan l'element compost desapareix, totes les seues parts desapareixen, ja que no tenen sentit per sí mateixos. A diferència també de l'agregació, en la composició, els elements components no es poden compartir entre diversos elements, mentre que en l'agregació el mateix objecte sí que pot estar compartit amb diversos elements.

La seua representació al diagrama de classes és amb una línia contínua amb un romble ple en la classe que és composta.

Per exemple, una partida d'un videojoc en línia serà la composició de diversos jugadors:

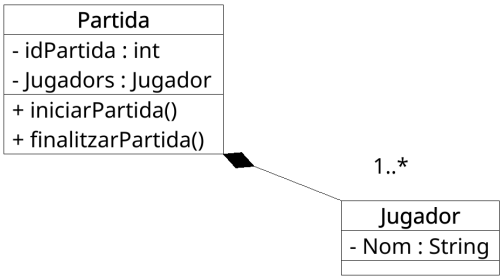


Figura 6: Exemple de composició

Aquest diagrama indica que una partida està composada per diversos jugadors. En aquest cas, una vegada es finalitza i es destrueix una partida, els jugadors de la partida deixen de tindre sentit. Fixeu-vos que la multiplicitat de *Jugadors* és *un a molts*, el que vol dir que una partida tindrà entre un i diversos jugadors.

La diferència entre agregació i composició és semàntica, i molts llenguatges de programació, com és el cas de Java no diferencien entre elles.

Anem a veure com fariem l'exemple de la partida en Java. Com representaríeu que *una partida pot tindre més d'un jugador*? Anem a veure alguna forma de representar-ho amb estructures de dades que ens proporciona Java.

En primer lloc, definiriem la classe `Jugador`:

```
1  public class Jugador {
2
3    private String Nom;
4
5    public Jugador () { };
6
7    public void setNom (String newVar) {
8      Nom = newVar;
9    }
10
11   public String getNom () {
12     return Nom;
13   }
14 }
```

Com veiem, és una classe senzilla, on definim l'atribut *Nom* i els mètodes accessors necessaris. Anem a veure ara com fer la classe *Partida*:

```
1
2  import java.Util.ArrayList;
3
4  public class Partida {
```

```
5
6     private int idPartida;
7
8     // Definim Jugadors com a una llista
9     // implementada com un vector (ArrayList)
10    // d'objectes de tipus *Jugador*
11    private ArrayList<Jugador> Jugadors;
12
13    public Partida () {
14        // En el constructor de la partida, creem
15        // aquest ArrayList.
16        this.Jugadors = new ArrayList<Jugador>();
17    };
18
19    public void setIdPartida (int newVar) {
20      idPartida = newVar;
21    }
22
23    public int getIdPartida () {
24      return idPartida;
25    }
26
27    public void setJugadors (String Nom) {
28      // Per afegir un nou jugador donat el nom
29      // el crearem i l'afegirem a l'ArrayList:
30
31      Jugador nouJugador=new Jugador();
32      nouJugador.setNom(Nom);
33      // Per afegir-lo, utilitzem el métode add
34      // de l'arraylist
35      this.Jugadors.add(nouJugador);
36    }
37
38    public Jugador getJugadors () {
39      return this.Jugadors;
40    }
41
42    public void iniciarPartida()
43    {
44    }
45    public void finalitzarPartida()
46    {
47    }
48
49  }
```

Analitzem un poc l'exemple. Hem introduït el tipus de dada `ArrayList`, definit a la llibreria `java.Utils`, i hem definit dins la classe `Partida` un `ArrayList` d'objectes de tipus `Jugador`:

```
1  public class Partida {
2  ...
```

```
3    private ArrayList<Jugador> Jugadors;
4    ...
```

La forma de declarar un `ArrayList` en general té la següent sintaxi:

```
1  ArrayList<TipusBase> NomDelArrayList;
```

Recordem que un `ArrayList` és una llista, que internament s'implementa com un vector, però la qual ens ofereix mètodes com `add`, `remove`, `set` o `get` per crear, eliminar, modificar o consultar elements, sense preocupar-nos de la gestió dels índex interna del vector.

Quan definim un `ArrayList`, hem d'indicar el tipus de dades que aquest contindrà (el seu *TipusBase*). Aquest podrà ser enter, cadenes de caràcters... o altra classe, com és el nostre cas, que es tracta d'objectes de tipus `Jugador`.

Amb això, hem definit l'`ArrayList`, però aquest no té encara espai reservat en memòria per a ell. Per a això, haurem de crear-lo dins el constructor de la classe partida. És a dir, quan creem una partida, haurem de crear l'`ArrayList` que contindrà la llista (buïda de moment) de jugadors.

Per tal de fer això, al constructor, farem:

```
1    public Partida () {
2        this.Jugadors = new ArrayList<Jugador>();
3    };
```

Simplement, hem hagut de crear l'`ArrayList` de `Jugadors` i assignar-lo a **`this.`**`Jugadors`.

Ara ens queda per vore com afegim jugadors a la partida. Per a això, anem a veure el mètode `setJugadors`:

```
1    public void setJugadors (String Nom) {
2      // Per afegir un nou jugador donat el nom
3      // el crearem i l'afegirem a l'ArrayList:
4
5      Jugador nouJugador=new Jugador();
6      nouJugador.setNom(Nom);
7      // Per afegir-lo, utilitzem el métode add
8      // de l'arraylist
9      this.Jugadors.add(nouJugador);
10   }
```

Com veiem, al mètode ens passen el nom del jugador, i és el propi mètode `setJugadors`, qui crea un nou objecte (`nouJugador`) de tipus `Jugador`, i l'afig a l'`ArrayList` mitjançant el mètode `add` que aquest ens proporciona.

Fixeu-vos que quan destruïm la partida, l'ArrayList també es destruïrà, i amb ell, tots els jugadors que s'han creat (a través del recol·lector de fem). Aquest és el comportament corresponent a una **composició**.

Si en aquest cas haverem volgut representar una **associació** en lloc de la composició, podriem haver definit el mètode `setJugador` de la següent forma:

```
1 public void setJugadors (Jugador jugador) {
2     this.Jugadors.add(jugador);
3   }
```

Fixeu-vos que en aquest cas, en lloc de rebre com a argument el nom del jugador, rebem ja una instància de jugador, el que vol dir que aquest objecte s'ha creat fora, i el que rebem, realment és una referència a aquest. En aqest cas, quan s'elimine la `Partida`, s'eliminarà l'ArrayList de jugadors, junt amb les instàncies a aquest, però no s'eliminaran els objectes de tipus `Jugador`, ja que segueixen existint fora d'aquesta classe (estan referenciats en algun lloc, i el recol·lector de fem no els elimina).

Dependència La relació de dependència es dóna quan una classe requereix d'alguna funcionalitat d'altra classe. La seua representació és una fletxa discontínua, des de la classe que requereix la untilitat de l'altra fins la classe que la ofereix.

Un exemple de dependència que us resultarà familiar és el següent:

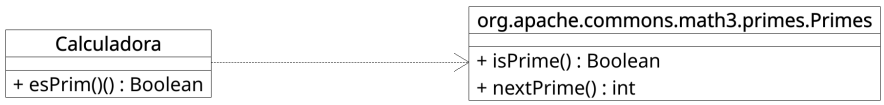


Figura 7: Representació d'una dependència

Com veiem, així expressem que la nostra classe *Calculadora* depenia de la classe *Primes*, de la llibreria `org.apache.commons.math3.primes.Primes`.

Herència L'herència és un dels tipus de relació més interessants i utilitzats en programació orientada a objectes, i ens permet que una classe *filla*, *herete* atributs i mètodes d'una altra classe (classe pare o superclasse). La *subclasse*, a més, podrà definir nous mètodes i atributs. S'utilitza en relacions *és un*.

Als diagrames de classes, per tal d'indicar una relació d'herència, es fa ús d'una línia amb un triangle a l'extrem on es troba la classe de la que s'hereta. Veiem-ho a següent exemple:

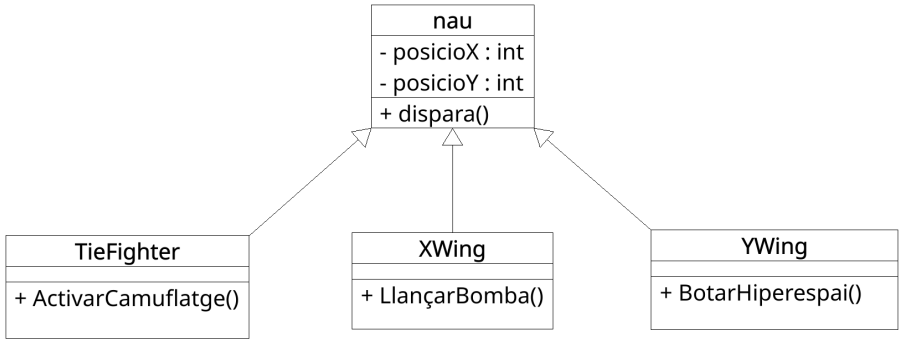


Figura 8: Exemple d'herència

Com veiem, tant les classes `TieFighter` com `XWing` i `YWing` són una especialització de la classe `Nau`, pel que *hereten* les seues propietats (`posicioX` i `posicioY`) així com els seus mètodes (`dispara()`). A més, cadascuna de les classes, definelx altres mètodes propis de la classe, com `ActivarCamuflatge()` en `TieFighter`, `LlançarBomba()` en `XWing` i `BotarHiperespai` en `YWing`.

En Java, fem ús de la paraula reservada **`extends`**per indicar que una classe deriva d'una altra. Veiem com quedaria l'exemple de dalt:

```
1  public class Nau {
2    private int posicioX;
3    private int posicioY;
4
5    /* Constructor */
6    public Nau () { };
7
8    /* Métodes d'accés */
9    public void setPosicioX (int newVar) {
10     posicioX = newVar;
11   }
12
13   public int getPosicioX () {
14     return posicioX;
15   }
16
17   public void setPosicioY (int newVar) {
18     posicioY = newVar;
19   }
20
21   public int getPosicioY () {
22     return posicioY;
23   }
24
25   /* Altres métodes */
26   public void dispara()
```

```
  {
  }
}
```

```
class TieFigter extends Nau {

    // Constructor
    TieFigher(){
        // Invoca al constructor del pare
        super();
    }

    // Métodes nous
    void ActivarCamuflatge(){
        ...
    }
}
```

```
class XWing extends Nau {

    // Constructor
    TieFigher(){
        // Invoca al constructor del pare
        super();
    }

    // Métodes nous
    void LlançarBomba(){
        ...
    }
}
```

```
class YWing extends Nau {

    // Constructor
    TieFigher(){
        // Invoca al constructor del pare
        super();
    }

    // Métodes nous
    void BotarHiperespai(){
        ...
    }
}
```

Açò ens permet, per exemple, definir un objecte de tipus `XWing`:

```
XWing nauLuke=new XWing();
```

I utilitzar tant els mètodes de `Nau` com de `XWing`:

```
1 nauLuke.dispara();
2 nauLuke.LlançarBomba();
```

En principi, podrem instanciar objectes de les quatre classes. En cas que només tinguera sentit definir naus dels tipus de les subclasses, podriem definir la classe `Nau` com a *classe abstracta*, indicant que aquesta classe només serveix per definiar altres classes, però no per instanciar objectes.

Un altre aspecte interessant que introdueix la programació orientada a objectes és el **polimorfisme**, que ens permet enviar missatges sintàcticament iguals a classes de diferent tipus. Per exemple, el mètode `dispara()` es pot utilitzar en qualsevol objecte de la classe `nau` o de qualsevol de les classes derivades. Açò ens permet, per exemple, en una partida, tindre una estructura de dades d'objectes genèrics, de tipus nau, que emmagatzeme naus de qualsevol tipus.

Per exemple, podriem definir un `ArrayList` d'objectes de tipus `Nau`:

```
1 ArrayList<Nau> flota=new ArratList<Nau>();
```

I afegir elements de qualsevol tipus derivat a dins:

```
1 XWing nauLuke=new XWing();
2 YWing nauSnap=new YWing();
3 TieFighter nauVader=new TieFighter();
4
5 flota.add(nauLuke);
6 flota.add(nauSnap);
7 flota.add(nauVader);
```

Sobre l'herencia múltiple

En un diagrama de classes ens podem trobar el cas que una classe siga una especialització de dues classes ascendents. Per exemple:

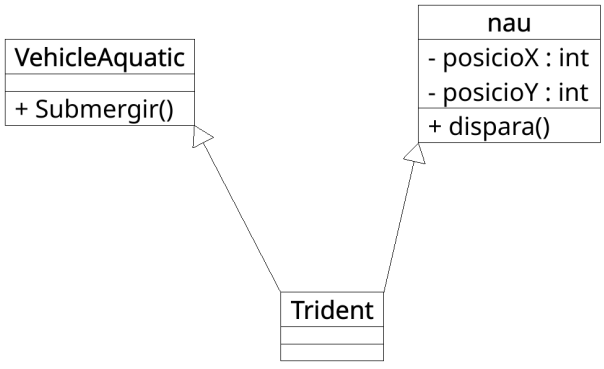


Figura 9: Exemple d'herència múltiple

En aquest exemple, la classe `Trident` és una especialització tant de `Nau` com de `VehicleAquatic`, pel que hereta les propietats i mètodes de les dos.

L'herència múltiple no està suportada en tots els llenguatges de programació. Al cas de Java, per exemple, una classe només pot tindre una classe ascendent. Aquesta limitació del llenguatge es supleix mitjançant les **interfícies**.

2.3. Interfícies

Una interfície és una entitat que declara una sèrie d'atributs, mètodes i obligacions. És una espècie de *contracte* on tota instància associada a una interfície ha d'implementar els serveis que indica aquella interfície.

Com que únicament són declaracions, les interfícies no poden ser instanciades, és a dir, no podem crear objectes a partir d'una interfície.

Les interfícies s'associen a classes. Una associació entre una classe i una interfície representa que la classe compleix amb el contracte que indica la interfície, és a dir, inclou aquells mètodes i atributs que indica la interfície.

La seua representació és similar a les classes, però indicant a dalt la paraula <<`interface`>>:

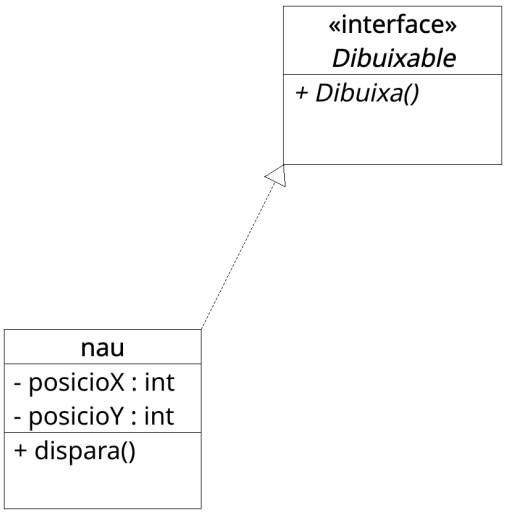


Figura 10: Exemple d'interfície

En aquest exemple, una nau diem que *implementa* la interfície `Dibuixable`, que *obliga* a les classes que la implementen a disposar d'un mètode *Dibuixa()*, per dibuixar la nau a la pantalla. En el cas de les interfícies, una **classe sí que pot implementar diverses interfícies**, suplint així la limitació de l'herència múltiple.

Com veiem, la relació entre una classe i la interfície o interfícies que implementa és amb la mateixa línia discontínua que indicàvem les relacions d'herència, amb un triangle/flexta a la *interfície*.

En Java, definiriem la interfície amb la paraula clau **`interface`**, i faríem ús de la paraula **`implements`** per indicar que una classe implementa aquesta interfície:

```
1  public interface Dibuixable {
2    public void Dibuixa();
3  }
4
5  public class Nau implements Dibuixable{
6
7    ...
8    public void Dibuixa(){
9        ...
10   }
11
12 }
```

3. Dibuixant diagrames de classes

Els diagrames de classe estan associats al disseny orientat a objectes. A mode de guía, els passos que generalment seguim a l'hora de realitzar aquest disseny seran els següents:

- **Identificar les classes,** que solen correspondre's a substantius dins el domini del problema (client, factura, jugador, etc.)
- **Identificar les relacions** entre les diferents classes, els punts en comú i les abstraccions entre elles, de manera que ajude a agupar-los en dibuixar el diagrama de classes.
- **Crear l'estructura** fent ús dels connectors apropiats entre les classes, donant especial atenció a la multiplicitat o les herències. Els atributs i mètodes es perfilaran més tard, una vegada estiga l'estructura del diagrama resolta.

Algunes **bones pràctiques** que podem tindre en compte a l'hora de generar diagrames de classes són:

- Quan els diagrames creixen poden tornar-se incoherents i poc clars. És preferible crear diagrames menuts que després pogam vincular entre ells que un gran diagrama que es preste a confussió.
- Per començar, podem fer ús de la notació de classe més simple per tindre una visió general d'alt nivell del sistema (pràcticament un entitat-relació), i perfilar-lo després amb els atributs i mètodes.

- Cal assegurar la major claredat en els diagrames, organitzant bé les classes, i evitant cosescom, per exemple que es creuen o superposen línies que representen relacions, ja que es presta a confussió.
- De vegades, també és útil fer ús de colors per agrupar mòduls comuns, i ajudar així al lector/client a diferenciar-los.