Universidad de Jaén
Departamento de Informática

Programación Orientada a Objetos

https://tiny.cc/poouja

v 1.0

José Ramón Balsas Almagro
Víctor Manuel Rivas Santos
Ángel Luis García Fernández
Juan Ruiz de Miras

Práctica 4. Relaciones entre clases (I)

Objetivos

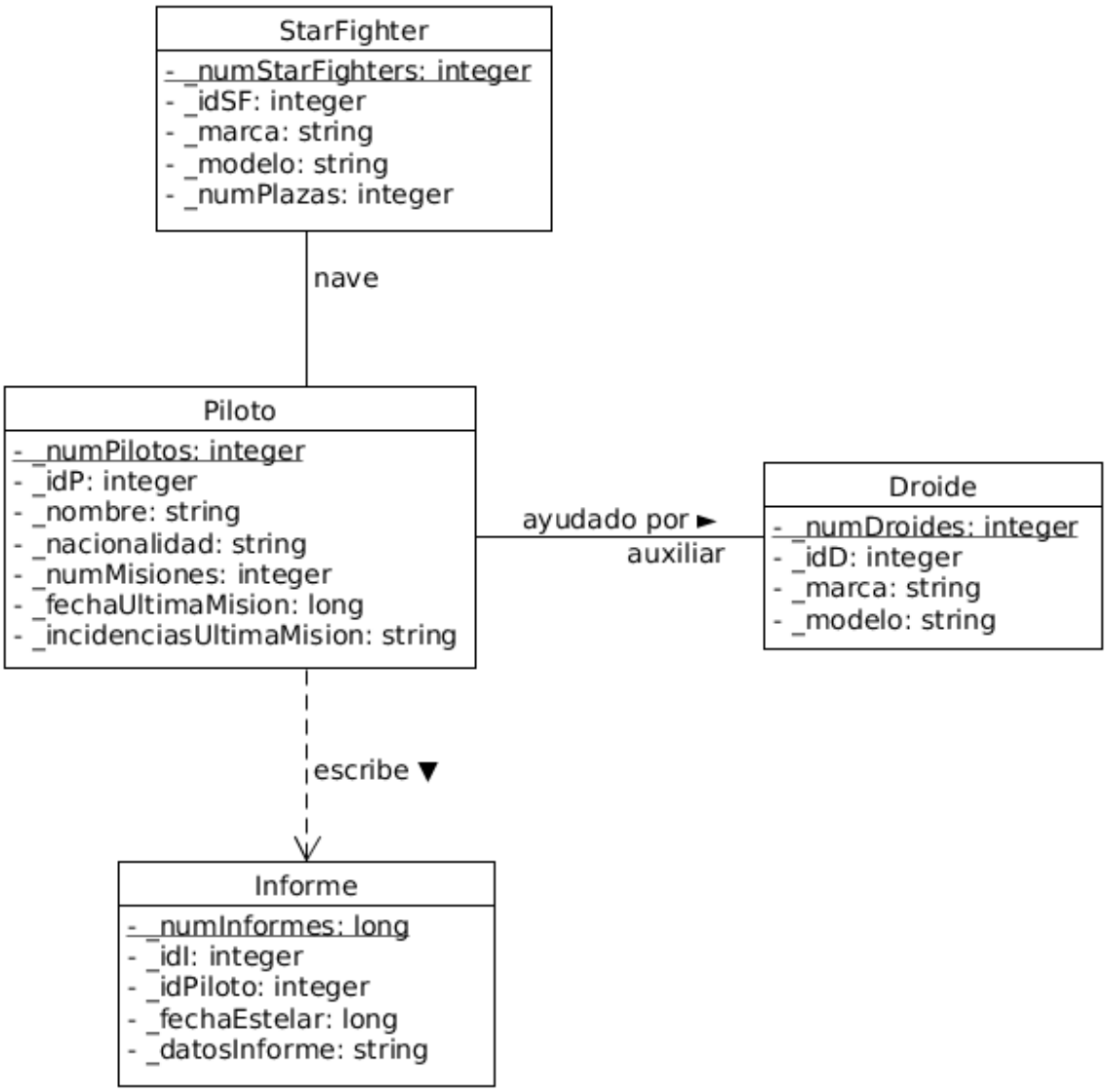
- Trabajar con diagramas UML con el diseño de clases y relaciones entre las mismas
- Implementar relaciones de dependencia y asociación
- Saber declarar, inicializar y usar variables de clase.
- Conocer el formato CSV como método de representación de información estructurada
- Saber utilizar clases de excepciones de la biblioteca estándar de C++

Índice

Contexto de la práctica

Hace mucho tiempo, en una galaxia muy, muy lejana… la flota interestelar de la Resistencia (apoyada por la República) que lucha contra la Primera Orden necesita organizar la

información relativa a sus pilotos, naves y demás equipamientos. El objetivo es gestionar de forma más eficiente sus recursos ante el ímpetu que la Primera Orden está consiguiendo en los últimos tiempos. Aunque parezca extraño, las computadoras de las que dispone la Resistencia son un tanto arcaicas, y sólo se pueden programar con un lenguaje muy antiguo, que sólo los miembros del Clan de Programadores todavía utilizan: C++. Yotepro Gramo, Maestro del Clan, haciendo gala de su buen juicio, ha decidido que inicialmente solo se va a desarrollar el software para organizar la información de las naves, los pilotos, los droides que les acompañan en sus misiones y los informes que cada piloto entrega al final de cada misión. Utilizando un sistema de representación sólo entendible para los miembros del Clan, y cuyos textos originales se perdieron durante la primera Guerra Clon, ha preparado el siguiente plano del software a desarrollar:



A partir de este plano, Casipro Gramo y Yoyapro Gramo, aprendices del Clan de Programadores, iniciaron el desarrollo de la aplicación. Sin embargo, sus conocimientos como aprendices se han mostrado insuficientes para terminar la misión encomendada por su maestro Yotepro. Desgraciadamente, en el último ataque de la Primera Orden a la base de la Resistencia murieron la mayor parte de los miembros del Clan de Programadores, así que Yoyapro y Casipro necesitan de tu ayuda para terminar la aplicación que han iniciado.

Punto de partida

El punto de partida de esta práctica son las clases ya escritas por Yoyapro y Casipro Gramo al alimón. El proyecto CLion (sí, el Clan todavía usa CLion ;-)) está disponible en el siguiente enlace: http://bit.ly/poouja1617pr5

Ejercicios

1. Añade a la clase *Piloto* los atributos y métodos necesarios para representar la relación entre un piloto y su nave. Si es necesario modificar métodos, haz también los cambios correspondientes.

2. Modifica las clases que consideres necesarias para representar la *relación entre pilotos y droides*. Si es necesario modificar métodos, haz también los cambios correspondientes.

3. Añade a la clase *Piloto* el método *generaInforme*, que devuelve un objeto de clase *Informe*, cuyos atributos replicarán la información almacenada en los atributos del piloto (en los datos del informe se incluirán también los identificadores de la nave y el droide que acompañaban al piloto en la misión, además de las incidencias de la última misión).

4. Realiza los cambios oportunos en la clase *StartFighter* para que los IDs de cada nave comiencen a partir del valor 1000. A pesar de ello, el valor del contador de naves debe seguir manteniendo su sentido original, es decir, el número de naves creadas hasta el momento.

5. Los StarFighters tienen que pasar revisiones cada 5000 parsecs[1]. Añade a la clase *StarFighter* un atributo *parsecs* de tipo decimal, que almacenará el número de parsecs que ha recorrido el StarFighter, así como un método *getParsecs* y un método *incrementarParsecs*, que solo admita un valor positivo como parámetro (y lance una excepción si no es así), y que añada ese valor a los parsecs recorridos por la nave. Modifica el método *StarFighter::toCSV* para que incluya también el nuevo atributo.

[1] Un parsec equivale a 19 billones de millas (3 años luz, milla arriba o milla abajo) https://starwars.fandom.com/wiki/Parsec

6. Modifica los métodos *set* de las clases *Droide*, *Piloto* y *StarFighter*, para que permitan encadenamiento de las llamadas a los mismos (en la clase *Informe* ya está hecho; puedes tomarla como referencia).

7. Completa la función *main*, de forma que:

- Se cree un array de 5 pilotos.
- Se cree un array de 5 naves.
- Se cree un array de 5 droides.
- Utiliza los setters de la clase *Piloto* y los datos del array *datosPilotos* para cargar información en los objetos de clase *Piloto*. (No tienes que procesar las cadenas de texto; simplemente copia y pega los datos).
- Utiliza los setters de la clase *StarFighter* y los datos del array *datosNaves* para cargar información en los objetos de clase *StarFighter*. (No tienes que procesar las cadenas de texto; simplemente copia y pega los datos). (Inicialmente, cada StarFighter tiene distancia recorrida 0).
- Utiliza los setters de la clase *Droide* y los datos del array *datosDroides[]* para cargar información en los objetos de clase *Droide*. (No tienes que procesar las cadenas de texto; simplemente copia y pega los datos).
- Se asocie el primer piloto con el tercer droide y la segunda nave.
- Se asocie el segundo piloto con el primer droide y la cuarta nave.
- Se asocie el tercer piloto con el segundo droide y la primera nave.
- Se generen dos informes con los datos de las últimas misiones de los dos primeros pilotos.
- Se muestren por la terminal los datos de los informes generados en formato CSV.
- Finalmente, se liberen los recursos ocupados.

8. (**Opcional**) En algunos puntos del código, Casipro y Yoyapro Gramo dejaron comentarios Doxygen indicando tareas pendientes (con la etiqueta **@todo**), principalmente relacionadas con excepciones. Intenta terminarlas.

Contenidos

Dependencia y asociación

A lo largo del Tema 3 de teoría estamos estudiando los distintos tipos de relaciones que pueden darse entre las clases que intervienen en una aplicación. Los dos primeros tipos de relaciones son:

- **Dependencia**, que implica el uso puntual de objetos de unas clases por objetos de otras clases

- **Asociación**, en las cuales los objetos de una de las clases (al menos una) conservan la relación con los objetos de la otra por un largo periodo de tiempo.

Cada una de estas relaciones puede ser modelada utilizando el lenguaje UML, como se ha visto en teoría, de forma que podemos indicar expresamente la clases que intervienen en cada relación y el tipo de las mismas. También es interesante señalar que el nombre de la relación o el rol de las clases participantes se puede utilizar para nombrar a los atributos que establecen la relación en cada una de las clases.

La implementación de las relaciones se realiza de distintas formas en función del tipo de relación, y no existe una forma estándar de implementar cada una de ellas. No obstante, sí suele haber ciertas prácticas que suelen ser habituales:

En el caso de la dependencia:

- Los objetos de una clase utilizan objetos de la otra. Esto puede reflejarse de distintas maneras; las más habituales son el paso de parámetros a los métodos y el valor devuelto por alguno de los métodos
- Como la dependencia no implica una relación duradera en el tiempo, incluso no tiene por qué reflejarse en modo alguno en los atributos de las clases relacionadas.

En el caso de la asociación:

- La práctica común es añadir a una de las clases un atributo de tipo puntero, de tal forma que cada objeto de esa clase almacena la dirección de memoria del objeto de la otra clase con el que se relaciona.
- El nombre de este atributo será el del rol con el que la relación está etiquetada en el diagrama UML.
- Como con otros tipos de atributos, en la clase con el atributo de tipo puntero podría ser necesario implementar métodos para asignar y consultar el valor de este atributo.
- Además, habrá que tener especial cuidado a la hora de destruir los objetos, puesto que la destrucción de un objeto asociado con otro no implica la destrucción de los dos objetos relacionados, sino que el que no se destruye puede ser reutilizado (o incluso podría ya estar asociado con otros objetos).

Variables de clase

Al definir una nueva clase, definimos los atributos y métodos que tendrán los objetos que definamos posteriormente. Normalmente, cada uno de los objetos que definamos almacenará sus valores específicos para cada uno de los atributos, es por ello que normalmente **a los atributos se les llama variables de instancia**, porque sus valores variarán para cada una de las instancias declaradas.

Sin embargo, es posible definir atributos que estén compartidos por todos los objetos de una

misma clase; son las **variables de clase**, también llamadas **atributos estáticos** o **atributos de clase**. Así, para cada objeto instanciado se reservará memoria para cada una de sus variables de instancia, sin embargo sólo se reservará memoria una vez para cada variable de clase. Todos los objetos de la clase compartirán dicha variable, pudiendo acceder a ella y modificarla (en caso de que no se haya definido como constante).

La inclusión de una variable de clase se realiza poniendo la palabra reservada **static** delante de la declaración del atributo, como en el siguiente ejemplo:

```
   static int contador = 0;
```

En el siguiente ejemplo mostramos una implementación de la clase *Temazo* con una variable de clase que almacena cuántos objetos se han creado desde que la aplicación se inicia, para así poder asignar a cada nuevo objeto un identificador único.

```
/// @file Temazo.h
class Temazo {
public:
   Temazo ( std::string titulo, std::string interprete,
            int duracion );
   //...
   int getID() const;
private:
   static int _totalTemazos; ///< contador del total de temazos
   std::string _titulo="";
   std::string _interprete="";
   int _duracion=0;
   int _puntuacion=0;
   int _idTemazo = 0; ///< id único de cada temazo
};
```

La variable de clase se puede utilizar desde cualquier método de la clase. Continuando con nuestro ejemplo, vamos a mostrar cómo se asigna desde el constructor de la clase el nuevo id único a cada nuevo temazo utilizando para ello la variable de clase con el contador de temazos creados hasta el momento:

```
/// @file Temazo.cpp
#include "Temazo.h"

// Inicialización de las variables de clase: en el .cpp, y fuera de
```

```
// cualquier método
int Temazo::_totalTemazos = 0;

Temazo::Temazo ( std::string titulo, std::string interprete,
                 int duracion ) : _titulo(titulo)
                                , _interprete(interprete)
                                , _duracion(duracion)
{
     // Asigna el id del nuevo temazo utilizando el atributo de clase
     _idTemazo = _totalTemazos + 1;

     // Modifica la variable de clase, para que guarde el número de
     // temazos creados hasta el momento desde que inició la aplicación
     _totalTemazos++;
}

int Temazo::getID () const
{  return _idTemazo;
}

// No se incluye la implementación del resto de los métodos de
// la clase
```

En el ejemplo anterior, cada vez que se ejecute el constructor de Temazo utilizamos el contador actual de temazos para asignar el nuevo ID al temazo que estamos creando. Además, aprovechamos para actualizar este contador, indicando que hay un nuevo objeto más. Este ejemplo también muestra una característica fundamental de las variables de clase: deben ser inicializadas antes de poder ser usadas. Como se puede observar, hemos inicializado la variable dentro del fichero .cpp, en una instrucción que está fuera de cualquier método de la clase.

Observa en el código de partida de la práctica cómo se han incluido variables de clase en las diferentes clases para disponer en todas ellas de identificadores únicos para los objetos creados de cada una.

El formato CSV

CSV es el acrónimo de "Comma Separated Values" y hace referencia a un tipo de fichero de formato texto utilizado para almacenar información simple de una hoja de cálculo. A grandes rasgos, podemos decir que cada fila de la hoja de cálculo se almacenará en una línea diferente del archivo (delimitada por el carácter de fin de línea). En cada línea, los valores de cada columna se almacenarán separados por un carácter específico, que normalmente es una coma "," o un punto y coma ";", aunque podrían ser otros. Utilizar uno u otro dependerá de la herramienta con la que queremos exportar/importar la información del fichero (Excel utiliza por defecto ";" mientras que otras hojas de cálculo utilizan ","), y de la convención que se

utilice para los decimales (en unos países se usa un punto para separar los decimales, y en otros la coma). Ejemplos de datos en formato CSV son:

```
Juego de tronos ; HBO ; 2011 ; English

Isabel ; RTVE ; 2011 ; Español
```

Lo importante de este tipo de archivos es que son una primera aproximación al almacenamiento de información estructurada y, además, permiten el intercambio de información entre diferentes aplicaciones. El motivo de esto último es que, al ser ficheros en formato texto, su contenido puede ser observado fácilmente con cualquier herramienta y, al estar estructurado (en filas y columnas), puede construirse fácilmente una aplicación que sea capaz de recuperar la información almacenada. De hecho, como hemos comentado, varias aplicaciones comerciales o libres existentes permiten actualmente exportar o importar contenidos utilizando ficheros en este formato.

Por último, comentaremos que aunque el formato CSV es el más extendido por su sencillez, tiene limitaciones a la hora de almacenar información con una estructura compleja. Por este motivo se han desarrollado soluciones mucho más potentes para almacenar información estructurada en ficheros de texto, como son XML o JSON.

Traducción de objetos a formato CSV

Un enfoque habitual para poder almacenar objetos en algún sistema de almacenamiento externo consiste en transformarlos en un formato que sea sencillo de almacenar y posteriormente recuperar, reduciendo la complejidad inherente de la propia estructura del objeto. Una primera aproximación simplificada a este problema podría consistir en utilizar el formato CSV para almacenar un objeto simple, organizando la información de sus atributos en una línea y separados por un carácter específico, como puede ser el punto y coma (**;**). Para ello, podríamos dotar a nuestras clases de métodos que permitan obtener la representación en formato CSV de un objeto, o de un constructor y/o método que permita inicializar un objeto a partir de una representación del mismo en formato CSV. A modo de comentario, este proceso de transformar un objeto en una representación que pueda almacenarse o transmitirse se le conoce con el nombre de serialización (*serialization* en inglés).

Ejemplo de uso de la conversión a CSV:

```
int main () {
   Plato primeros[5];   // Vector de platos
   /*
      Trabajo con los objetos: asignación de valores…
```

```
   */
   // Finalmente, se muestran en formato CSV en pantalla
   for ( int i = 0; i < 5; i++ )
   {
      std::cout << primeros[i].toCSV () << std::endl;
   }
   return 0 ;
}
```

El siguiente paso consistirá en encontrar una forma de transformar los atributos de un objeto en una secuencia de caracteres donde los diferentes valores están separados por un carácter específico, como por ejemplo el punto y coma (";"). Simplificando mucho el problema, nos quedaremos únicamente con atributos de tipo simple, como pueden ser cadenas de caracteres y números (enteros y reales). En el caso de los atributos de tipo cadena de caracteres no hay problema, puesto que ya tienen una representación de texto. Sin embargo, en el caso de los atributos de tipo entero o real, necesitamos transformarlos de su representación binaria a su representación en cadena de caracteres. Para ello utilizaremos la clase std::stringstream de la biblioteca estándar <sstream> de C++.

Podemos considerar que un objeto de la clase stringstream tiene la misma funcionalidad que los objetos cin y cout; es decir, sus mismos métodos y operadores, con la diferencia que las operaciones realizadas no tienen repercusión sobre la pantalla o el teclado. Por el contrario, las operaciones realizadas sobre un stringstream se realizarán sobre una estructura interna que podrá ser recuperada posteriormente como una cadena de caracteres. Veamos un ejemplo:

```
#include <sstream>
#include <string>
#include <iostream>

int main () {
   std::stringstream ss;
   std::string nombre("Francisco Manuel"),
               apellidos("Gómez Pérez"), resultado;
   int edad=34;
   float peso=81.5;

   // Enviamos información al objeto stringstream
   // Los operadores << van transformando en texto cada tipo
   // de dato
   ss << nombre << ";" << apellidos << ";" << edad << ";"
      << peso;

   // Recuperamos el contenido de ss en una cadena
   // Podría usarse: ss>>resultado, pero usamos str
```

```
    // para que también se incluyan los espacios en blanco
    resultado = ss.str ();

    // Mostramos la cadena:
    // Francisco Manuel;Gómez Pérez;34;81.5
    std::cout << resultado <<std::endl;
    return 0;
}
```

En la siguiente sesión estudiaremos cómo utilizar la clase stringstream para realizar la lectura de objetos almacenados de esta forma. Para más detalles sobre la manipulación de flujos de texto, es recomendable consultar el capítulo correspondiente del libro de Joyanes (2014) propuesto en la bibliografía, o cualquier otro que trate sobre C++.

Encadenamiento de llamadas a métodos en C++

Una práctica habitual en C++, así como en otros lenguajes de programación orientada a objetos es el denominado *encadenamiento de métodos* (*method chaining* en inglés). De hecho, es algo que hasta ahora hemos estado haciendo continuamente sin darnos cuenta cada vez que hemos utilizado el operador "<<" con el objeto *cout* de la biblioteca estándar de C++:

```
    std::cout << "esto "
              << "son llamadas encadenadas "
              << "al método <<"
              << std::endl;
```

La clave está en que los métodos devuelvan una referencia al propio objeto sobre el que se llama al método. Así, se puede aprovechar esta referencia para llamar a otro método de la clase sobre el mismo objeto. Por ejemplo:

```
/// @file Persona.h
/*...resto del código...*/
class Persona
{  private:
       std::string _nombre = "";
       std::string _apellidos = "";
       int _edad = 0;
       /*...resto del código...*/
    public:
       /*...resto del código...*/
       // MÉTODOS SET QUE NOS PERMITEN ENCADENAMIENTO:
       Persona& setNombre ( const std::string& nNombre );
       Persona& setApellidos ( const std::string& nApellido );
       Persona& setEdad ( const int nEdad );
};
```

```
/// @file Persona.cpp

/*...resto del código...*/
Persona& Persona::setNombre ( const std::string& nNombre )
{  this->_nombre = nNombre;
   return ( *this );            // Necesario devolver la referencia
}

Persona& Persona::setApellidos ( const std::string& nApellido )
{  this->_apellidos = nApellido;
   return ( *this );            // Necesario devolver la referencia
}

Persona& Persona::setEdad ( const int nEdad )
{  this->_edad = nEdad;
   return ( *this );           // Necesario devolver la referencia
}
```

```
/// @file main.cpp

#include "Persona.h"
/*...resto del código...*/
int main ()
{  Persona p;
   // LLAMADAS ENCADENADAS A MÉTODOS SET SOBRE EL MISMO OBJETO:
   p.setNombre ( "Pepe" )
    .setApellidos ( "Martínez" )
    .setEdad ( 33 );
}
```

El encadenamiento de llamadas a métodos nos permite hacer código más conciso y fácil de mantener.

Excepciones en la biblioteca estándar de C++

En el tema 2 estudiamos la utilidad que podía suponer utilizar objetos para lanzar excepciones. Como esto es algo habitual, la biblioteca estándar de C++ incorpora el módulo *stdexcept* que define algunas clases para lanzar excepciones asociadas a errores habituales. De esta forma, en muchos casos evitaremos tener que crear nuevas clases para lanzar excepciones y podremos reutilizar las propias de la biblioteca estándar de C++. El uso de estas excepciones desde nuestras clases supone una relación de dependencia con ellas pero, al ser

consideradas estándar, tenemos la certeza de que estarán disponibles en cualquier compilador de C++ que podamos utilizar.

Algunas de estas excepciones son: *std::invalid_argument*, para cuando el valor de algún parámetro en una función o método no son adecuados; *std::out_of_range*, para valores numéricos fuera de un rango permitido; *std::length_error*, etc. El resto de excepciones disponibles pueden encontrarse en la documentación del módulo *exception*.

En cuanto a la funcionalidad que nos aportan, lo interesante es que todas ellas permiten añadir un mensaje descriptivo en su constructor y disponen de un método *what()* para obtener dicho mensaje. Por ejemplo:

```
#include <stdexcept>

void StarFighter::setNumPlazas ( int numPlazas )
{  if ( numPlazas <= 0 )
   {  throw std::invalid_argument ( "[StarFighter::setNumPlazas]: el "
                                    "número de plazas no puede ser negativo"
);
   }
   _numPlazas = numPlazas;
}

int main ( )
{  StartFighter nave;

   try
   {  nave.setNumPlazas(-1);
   }
   catch (std::invalid_argument &e)
   {  std::cerr << "Ocurrió un error: " << e.what();
   }

   return 0;
}
```