Universidad de Jaén
Departamento de Informática

Programación Orientada a Objetos

http://bit.ly/poouja

Curso 2019/2020

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

Práctica 5. Relaciones entre clases (I)

Objetivos

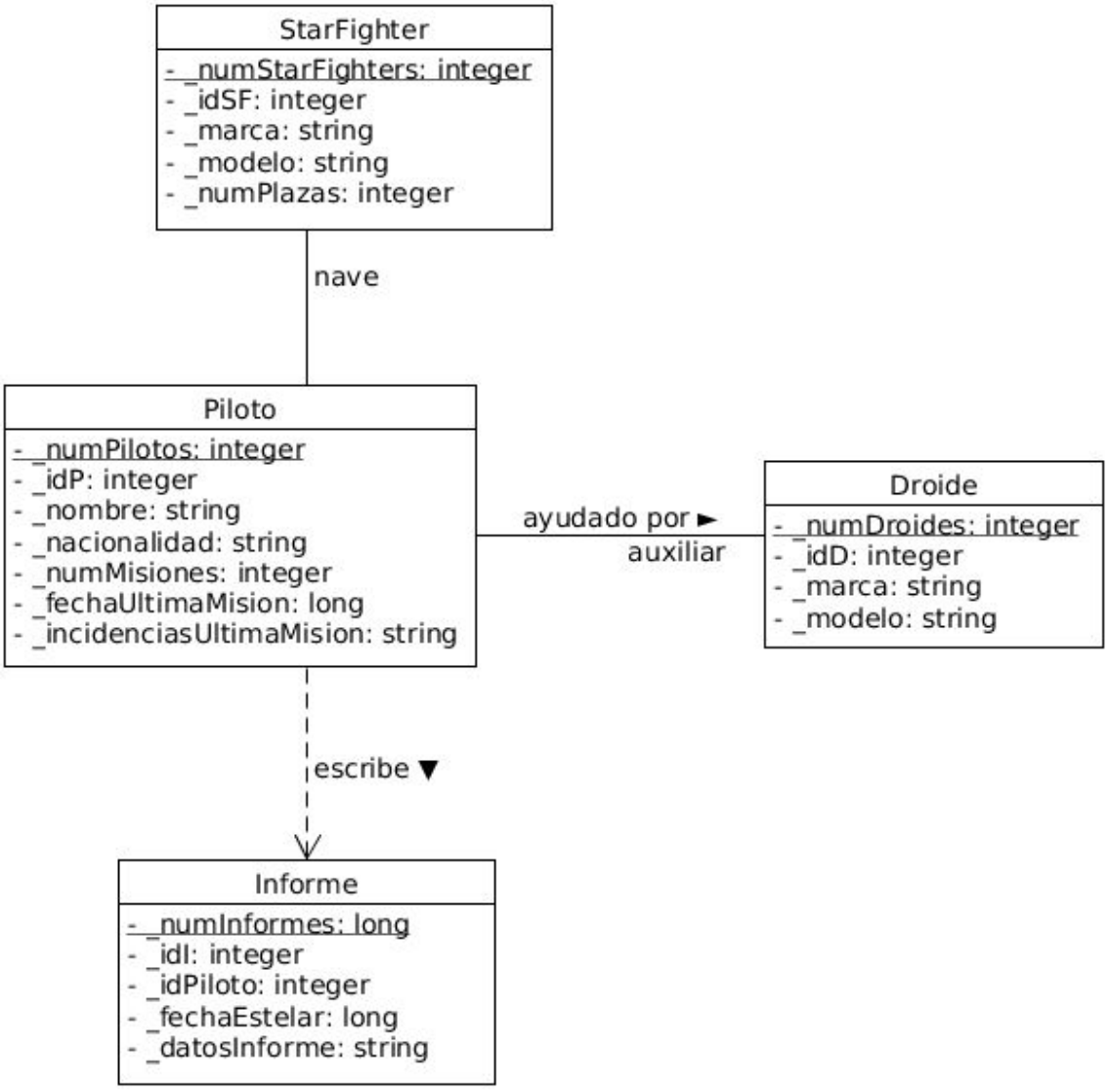
- Trabajar con diagramas UML con el diseño de clases y relaciones entre las mismas
- Implementar relaciones de dependencia y asociación
- Utilizar objetos std::stringstream para recuperar datos de cadenas de caracteres
- Saber utilizar clases de excepciones de la biblioteca estándar de C++

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 NetBeans (sí, el Clan todavía usa NetBeans ;-)) 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 devuelva un *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. Añade a las clases *Piloto*, *Droide* y *StarFighter* un método *fromCSV*, que dada una cadena de texto en formato CSV con los valores de los atributos de un objeto de la clase, extraiga dichos valores y los asigne a los atributos del objeto.

5. 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).

6. 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.
- Utilizando el método *Piloto::fromCSV* se carguen en el array de pilotos los datos proporcionados en el array *datosPilotos[]*.
- Utilizando el método *StarFighter::fromCSV* se carguen en el array de naves los datos proporcionados en el array *datosNaves[]*.
- Utilizando el método *Droide::fromCSV* se carguen en el array de droides los datos proporcionados en el array *datosDroides[]*.
- 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.

7. (**Opcional**) En algunos puntos en el 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).

Recuperación de datos en formato CSV

En la práctica anterior vimos cómo podíamos representar de una forma estructurada la información de un objeto gracias al formato CSV. De igual forma, podemos utilizar un objeto de la clase *stringstream* para descomponer una cadena de caracteres en formato CSV. Así, obtendremos los diversos elementos que constituyen la cadena, aprovechando la funcionalidad del operador >>, que realiza las transformaciones adecuadas de cada valor al tipo utilizado en la representación. Mira el siguiente ejemplo:

```
#include <sstream>
#include <string>
int main ()
{  string entrada ( "José Manuel;Alcántara Rodríguez;27;78.4" );
   std::stringstream ss;
   string nombre, apellidos;
   int edad;
   float peso;

   // Inicializamos el contenidos de ss
   // Otra forma de hacerlo es: ss << entrada;
   ss.str ( entrada );

   // Leemos el nombre y los apellidos. Usamos getline para incluir los
   // espacios en blanco
    getline( ss, nombre, ';'); //El carácter ';' se lee y se elimina de
ss
   getline ( ss, apellidos, ';' );

   // Los operadores >> van transformando cada cadena en su tipo
   // correspondiente
   ss >> edad;

   ss.ignore ( 1 );      // Sacamos el siguiente carácter ';' de ss
                         // También puede hacerse con ss.get();
   // Toma el último valor del stringstream
   ss >> peso;
   return 0;
}
```

Hay que tener en cuenta que el ejemplo anterior es una versión muy simplificada de la operación

de transformación de formato CSV a la representación interna de los valores constituyentes, en la cual no se han tenido en consideración situaciones en las cuales la cadena de entrada a procesar no tuviera el formato adecuado, p.e. que faltara o sobrara una columna, o que no se pudiera convertir el siguiente dato al formato esperado.

Para más detalles sobre la manipulación de flujos de texto se recomienda repasar los apuntes de C++ de la asignatura Fundamentos de Programación.

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;
}
```