Programación 2
Curso 2015/2016

Práctica 3
Imperial Commander (orientado a objetos)

Esta práctica consiste en implementar, utilizando programación orientada a objetos, una parte de la práctica 2, eliminando todo lo relacionado con ficheros.

1. Normas generales

1.1. Entrega

1. El último día para entregar esta práctica es el **viernes 20 de mayo, hasta las 23:59**. No se admitirán entregas fuera de plazo.

2. La práctica debe tener varios ficheros llamados "`Fighter.cc`" y "`Fighter.h`", "`Fleet.cc`" y "`Fleet.h`", "`Ship.cc`" y "`Ship.h`", "`Util.cc`" y "`Util.h`", y "`Makefile`". Para intentar evitar confusiones, se debe entregar todo el código (incluyendo el `Makefile` correspondiente) en un archivo comprimido `prog2p3.tgz`, que se debe crear con la orden:

   ```
   tar cvfz prog2p3.tgz *.cc *.h Makefile
   ```

   Para poder generar un ejecutable y realizar pruebas, en la página web de la asignatura se publicará un fichero "`imperialCommander.cc`", que no debería ser modificado y que no es necesario entregar. Para corregir la práctica se utilizará ese fichero, esté o no presente en el `prog2p3.tgz`.

1.2. Detección de plagios/copias

En el Grado en Ingeniería Informática, la Programación es una materia fundamental, que se aprende programando y haciendo las prácticas de las diferentes asignaturas (y otros programas, por supuesto). Si alguien pretende aprobar las asignaturas de programación sin programar (copiando), obviamente tendrá serios problemas en otras asignaturas o cuando intente trabajar. Concretamente, en Programación 2 es muy difícil que alguien que no ha hecho las prácticas supere el examen de teoría, por lo que copiar una práctica es una de las peores decisiones que se puede tomar.

La práctica debe ser un trabajo original de la persona que la entrega; en caso de detectarse indicios de copia de una o más prácticas se suspenderá la asignatura a **todos los alumnos implicados** (tanto al que copia como al que se *deja* copiar), y **se enviará un informe a la dirección de la EPS**, para que se tomen las medidas disciplinarias oportunas.

1.3. Otras normas

1. La práctica se debe entregar exclusivamente a través del servidor de prácticas del departamento de Lenguajes y Sistemas Informáticos, al que se puede acceder desde la página principal del departamento (`www.dlsi.ua.es`, "Entrega de prácticas") o directamente en la url `http://pracdlsi.dlsi.ua.es`.

   - No se admitirán entregas por otros medios (correo electrónico, Campus Virtual, etc.).
   - El usuario y contraseña para entregar prácticas es el mismo que se utiliza en el Campus Virtual.
   - La práctica se puede entregar varias veces, pero sólo se corregirá la última entrega (las anteriores entregas no se borran).

2. El programa debe poder ser compilado sin errores con el compilador de C++ existente en la distribución de Linux de los laboratorios de prácticas; si la práctica no se puede compilar su calificación será 0. Se recomienda compilar y probar la práctica con el autocorrector inmediatamente antes de entregarla.

3. La corrección de la práctica se hará de forma automática, por lo que es imprescindible respetar estrictamente los textos y los formatos de salida que se indican en este enunciado.

4. Al principio de todos los ficheros fuente entregados se debe incluir un comentario con el nombre y el NIF (o equivalente) de la persona que entrega la práctica, como en el siguiente ejemplo:

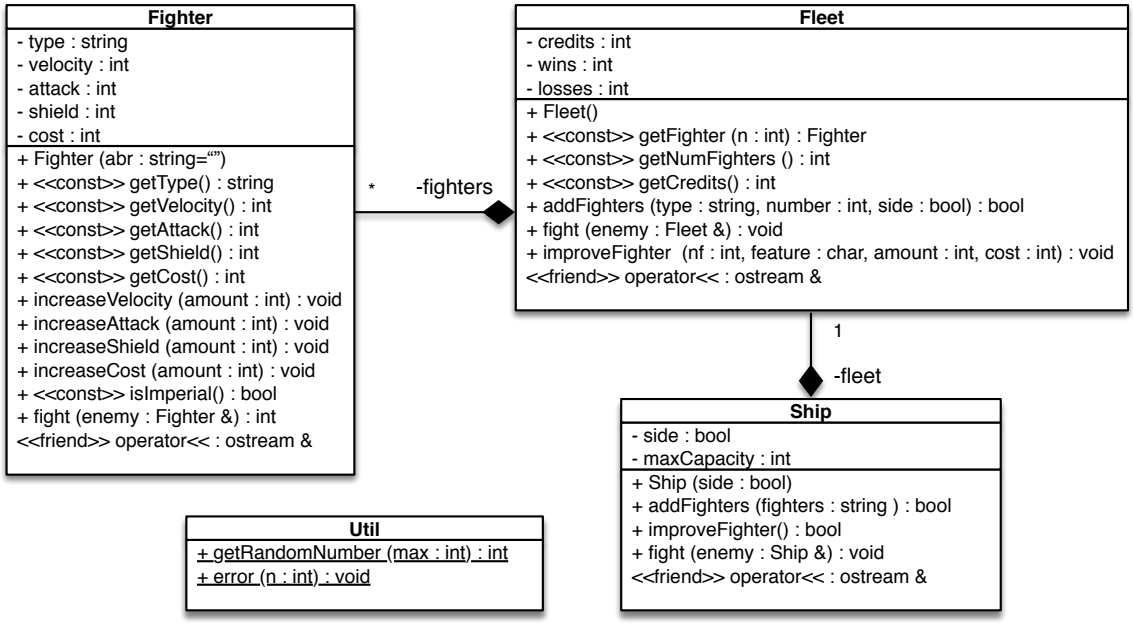
   ```
   // NIF   12345678X  GARCIA GARCIA, JUAN MANUEL
   ```

5. El cálculo de la nota de la práctica y su influencia en la nota final de la asignatura se detallan en las transparencias de la presentación de la asignatura.

2. Introducción

En esta práctica se implementarán con programación orientada a objetos las opciones del menú relacionadas con la gestión de los cazas del destructor imperial y con la lucha entre escuadrones de cazas. Las opciones relacionadas con ficheros binarios y de texto no se implementarán en esta práctica.

En la siguiente página tienes un diagrama UML con las clases que es necesario implementar. Es aconsejable que imprimas en papel el enunciado de la práctica para estudiar bien el diagrama.



3. Clases y métodos

En el diagrama UML se muestran las clases que se deben implementar, con sus atributos y métodos. Además, tendrás que utilizar más métodos (privados siempre) y quizá atributos (también privados), según sea necesario para implementar la práctica.

A continuación se describen los métodos más importantes de cada clase; los constructores y *getters/setters* no se explican, salvo que lo que tengan que hacer no sea evidente. Algunos de ellos no es necesario utilizarlos en la práctica, pero se utilizarán en las pruebas unitarias para la corrección de la práctica.

Importante: para facilitar la corrección con pruebas unitarias, los atributos y métodos privados deben declararse con la palabra reservada `protected` en vez de con `private`.

3.1. Fighter

Esta clase se utiliza para almacenar los datos de un caza. Los métodos más relevantes son:

`Fighter` : constructor para crear un caza del tipo indicado en el argumento, que será una abreviatura de las utilizadas en las prácticas anteriores. Si el argumento es una cadena vacía (de longitud 0), o bien si se invoca al constructor sin parámetros, se creará un caza con todos los datos a 0 y con el atributo "`type`" inicializado a una cadena vacía.

Si la abreviatura que se pasa como parámetro no coincide con ninguna de las abreviaturas conocidas, se debe lanzar la excepción `invalid_argument`, pasando como parámetro de la excepción la cadena `"wrong fighter type"`. Para poder usar la excepción `invalid_argument` es necesario incluir el

fichero `stdexcept`, tanto en este fichero como en el que se captura la excepción:

```
#include <stdexcept>
```

`increaseVelocity`, etc : los métodos `increaseVelocity`, `increaseAttack`, `increaseShield` e `increaseCost` suman el valor del parámetro al valor del atributo correspondiente.

`isImperial` : devuelve `true` si es un caza imperial, o `false` en caso contrario. Para ello se debe comparar el atributo `type` con los datos de la tabla de datos de cazas.

`fight` : simula la lucha entre dos cazas, el que recibe la invocación del mensaje y el parámetro, de forma idéntica a la descrita en la práctica 1. Por ejemplo, si este método se invoca así:

```
f1.fight(f2);
```

se simulará el combate entre los cazas `f1` y `f2`.

`operator<<` : operador de salida, imprimirá los datos de un caza como la función `listFighter` de las prácticas anteriores.

Nota sobre implementación: Las constantes `MAXFIGHTERS` y `FIGHTERABR` deberían aparecer en el fichero `Fighter.h` antes de la clase. Además, la tabla con los datos de los cazas solamente se utiliza en esta clase, y por tanto debe ir dentro del fichero `Fighter.cc`, aunque es necesario utilizar un registro diferente:

```
// Fighter data table
struct FighterData {
  string type;
  int velocity,attack,shield,cost;
};

const FighterData FIGHTERTABLE[] = {
    { "TIE-Fighter",      150,  75, 30,  45 },
    { "TIE-Bomber",        80, 150, 45,  75 },
    { "TIE-Interceptor",  180,  65, 30,  55 },
    { "TIE-Advanced",     160,  80, 90,  95 },
    { "X-Wing",           175,  90, 75,  65 },
    { "Y-Wing",            90, 150, 90,  90 },
    { "A-Wing",           200,  60, 50,  45 },
    { "B-Wing",           120, 200, 90, 100 }
};
```

3.2. Fleet

Esta clase implementa las flotas de cazas de las naves imperiales y rebeldes. Como en la práctica anterior, debe almacenar los cazas utilizando la clase `vector`. Los métodos más importantes son:

`Fleet` : constructor que inicializa créditos, victorias y derrotas.[1]

[1] Las constantes `IMPERIAL` y `REBEL` deben aparecer en el fichero `Fleet.h` antes de la clase, ya que se utilizan en el método `addFighters` y en la clase `Ship`.

`addFighters` : añade cazas a la nave, siempre que sea posible. Debe mostrar el error `WRONG_FIGHTER_TYPE` si el tipo de caza es incorrecto, que se produce cuando:

1. La abreviatura utilizada (el parámetro `type`) no es correcta, en cuyo caso el constructor de Fighter lanzará la excepción `invalid_argument` que es necesario capturar en este método con un bloque `try-catch`.
2. Es un tipo de caza del bando contrario, es decir, la flota es de una nave imperial y el caza es rebelde, o bien la flota es rebelde y el caza imperial.

Por último, si el tipo de caza es correcto debe mostrar el error `NO_FUNDS` si no hay suficientes créditos para añadir todos esos cazas a la flota; es posible que se pueda añadir alguno, pero no todos, en cuyo caso se muestra el error y no se añade ninguno.

Debe devolver `true` si ha conseguido añadir los cazas, y false en caso contrario.

`fight` : este método simulará el combate entre escuadrones de dos flotas, la que recibe la llamada (normalmente será la flota del destructor imperial) y el parámetro (normalmente la de la nave rebelde); el combate se simula como en la práctica anterior. Se debe utilizar varios métodos privados para implementar la lucha entre escuadrones.

Para simular la lucha se debe pedir los cazas imperiales que formarán parte del escuadrón (después de haber comprobado que hay suficientes cazas en ambas flotas para un combate), crear el escuadrón imperial, crear el escuadrón rebelde (como en la práctica anterior, de forma aleatoria), simular el combate entre los escuadrones, actualizar los datos de victorias, derrotas y créditos de ambas flotas, y finalmente se debe devolver los cazas supervivientes a cada flota.

`getFighter` : devuelve el caza de la flota indicado por el parámetro, que se debe suponer que tendrá un valor entre 0 y el número total de cazas de la flota menos 1 (no se debe comprobar). El caza no se debe eliminar de la flota.

`getNumFighters` : devuelve el número total de cazas de la flota.

`getCredits` : devuelve el total de créditos disponible en la flota.

`improveFighter` : según el valor del parámetro `feature`, que será "`v`", "`a`" o "`s`", incrementa la velocidad, el ataque o el escudo del caza indicado como primer parámetro (que será un número entre 0 y el número total menos 1, como en `getFighter`), aumenta el coste del caza, y disminuye los créditos de la nave. No se debe comprobar que los parámetros tienen valores correctos, se debe asumir que lo serán.

`operator<<` : operador de salida, muestra los créditos, victorias y derrotas de la flota, y los cazas de la flota con el formato de la práctica anterior. Una salida de ejemplo sería:

```
credits=425, wins=0, losses=0
[1] TIE-Fighter (v=150, a=75, s=30, c=45)
[2] TIE-Fighter (v=150, a=75, s=30, c=45)
...
[25] TIE-Advanced (v=160, a=80, s=90, c=95)
```

Aunque realmente este operador se va a invocar desde el operador de salida de `Ship`, que sacaría también la capacidad y el bando de la nave, que se imprimirían antes, con lo que quedaría:

```
Ship info: max. capacity=30, side=IMPERIAL, credits=425, wins=0, losses=0
[1] TIE-Fighter (v=150, a=75, s=30, c=45)
[2] TIE-Fighter (v=150, a=75, s=30, c=45)
...
[25] TIE-Advanced (v=160, a=80, s=90, c=95)
```

3.3. Ship

En esta clase se implementan los métodos relacionados con el destructor imperial y en general cualquier nave, imperial o rebelde.

`Ship` : crea una nave, imperial o rebelde según el valor del parámetro, inicializando la capacidad y la dotación inicial de cazas (llamando al método `addFighters`, como en la práctica anterior). Las constantes para la capacidad de las naves y la dotación inicial de cazas deben aparecer en el fichero `Ship.cc`.

`addFighters` : añade cazas al destructor imperial, siempre que sea posible, llamando al método con el mismo nombre en la clase `Fleet`. Si los nuevos cazas no caben en la nave se debe emitir el error `CAPACITY_EXCEEDED`; esta comprobación debe realizarse antes de llamar al método de la clase `Fleet`. Esto supone un cambio con respecto a la práctica 2: por ejemplo, si el destructor imperial tiene 29 cazas y le quedan 100 créditos, y el usuario introduce en la opción `Add fighters` la cadena `"3taf"`, el error que debe salir es el de `CAPACITY_EXCEEDED`; en la práctica 2, saldría primero el error `WRONG_FIGHTER_TYPE` y, en el caso de que estuviera bien la abreviatura (`"3ta"`), saldría el error `NO_FUNDS`.

Debe devolver `true` si ha conseguido añadir todos los cazas, y `false` en caso contrario.

`improveFighter` : pide al usuario por pantalla el número de caza a mejorar, la característica que se desea mejorar, y la cantidad. Después calcula el coste en créditos, pide confirmación y realiza la mejora llamando al método `improveFighter` de la clase `Fleet`. Se puede reutilizar el código de la práctica anterior con pocas modificaciones.

Debe devolver `true` si se ha modificado el caza, `false` en caso de que se haya producido algún error.

`fight` : simula el combate entre dos escuadrones, uno imperial y otro rebelde (como en la práctica anterior).

`operator<<` : muestra los datos de la nave, como en las prácticas anteriores (similar a la función `listShip`).

3.4. Util

Esta clase implementa básicamente dos métodos que se utilizan en las otras clases:

`getRandomNumber` : se utiliza para obtener un número aleatorio.

`error` : se usa para emitir mensajes de error. Las constantes con los números de los errores deben estar en el fichero `Util.h`

4. Programa principal

El programa principal estará en el fichero "`imperialCommander.cc`" que se publicará en la web de la asignatura. Este fichero no debería modificarse y puede no entregarse con la práctica (pero si se entrega no habrá ningún problema, se sobreescribirá con la versión publicada en la web antes de corregir).

5. Implementación

Para implementar la práctica debes tener en cuenta, además de todo lo comentado anteriormente:

1. Es probable que puedas aprovechar código de las prácticas 1 y 2, y es buena idea que lo hagas, pero tampoco intentes aprovecharlo todo; a veces se tarda menos en escribir el código de nuevo que en intentar adaptar un código ya existente.

2. Como en las prácticas anteriores, el resto de decisiones en la implementación de la práctica quedan a tu criterio.