Práctica 1
Programación orientada a objetos
Gestión de una clínica
(versión 0.1)

Pedro J. Ponce de León
David Rizo
Departamento de Lenguajes y Sistemas Informáticos
Universidad de Alicante

1. Introducción

Esta práctica consiste en implementar un programa en C++ que permita gestionar los ingresos y altas de pacientes en una clínica, obtener listados de pacientes según diferentes criterios. En esta práctica además añadiremos un sencillo proceso de gestión de diagnóstico y tratamiento del paciente. Implementaremos lo que se denomina el *modelo del dominio* del sistema y la llamada *lógica de negocio*, donde se implementa la funcionalidad específica de nuestra aplicación.

2. Modelo del dominio

El modelo del dominio incluye las clases que representan los diferentes tipos de entidades de las cuales interesa guardar información. La funcionalidad que éstas añaden a través de sus métodos debe tener como finalidad el proceso simple de los datos. En esta práctica se ha incluido parte de la lógica de negocio en el modelo del dominio. Como veremos más adelante, esta lógica de negocio corresponde a los procesos de ingreso y tratamiento del paciente. El diagrama de clases del modelo del dominio se muestra en el documento adjunto a este enunciado.

Una clínica tiene varias especialidades (Otorrinolaringología, Pediatría, Traumatología, etc.). De las especialidades conocemos su nombre, las habitaciones de las que disponemos en planta y los médicos adscritos a la especialidad. No todas la habitaciones tienen el mismo número de camas, éste se especifica en su creación. La habitaciones obtienen un número de habitación de forma correlativa en su creación. Además sabemos en todo momento qué paciente está en cada cama, las cuales se indexan desde 0. En cuanto a los médicos, almacenamos su nombre y apellidos, así como los pacientes que tienen asignados.

De cada paciente conocemos su número SIP, nombre y apellidos, sexo, su fecha de nacimiento a partir de la cuál podremos calcular su edad, el médico que tiene asignado, la especialidad en la

que se le ha ingresado, y los diagnósticos que se le han hecho. Cuando se crea el paciente no tiene ni especialidad ni médico asignado, por tanto no ha sido aún diagnosticado.

Los diagnósticos se componen de un texto explicativo, un tipo de diagnóstico (alta, leve, grave, muy grave), y un tratamiento posiblemente vacío. De los tratamientos sólo almacenaremos un texto descriptivo.

3. Interfaz de las clases

Fecha

La información sobre fechas se guarda como objetos de esta clase. Aparte de los métodos de la forma canónica y los accesores tenemos los siguientes:

isBisiesto Devuelve cierto si el año de la fecha es bisiesto. Devuelve falso en caso contrario.

imprimir y toString el modo *imprimir* emite una representación textual de la fecha por el flujo de salida. El formato debe ser `dia/mes/año`, usando el caracter `'/'` como separador. Si algún atributo contiene un valor no válido, imprimirá la cadena `<Fecha no valida>`. El método **toString()** devolverá una cadena en este mismo formato.

getDiferenciaAnyos Obtiene la diferencia en años entre dos fechas. Se puede usar para calcular la edad de un paciente. Por ejemplo, una persona nacida el 3 de Junio de 1986, el 10 de Marzo de 2008 tendrá 21 años, que es la diferencia en años completos entre ambas fechas. Véase la librería `<ctime>`[1] para más información sobre el tratamiento de fechas.

operator<< Operador de salida. Realiza la misma función que el método *Fecha::imprimir*.

Clinica

Una clínica se crea inicialmente sin ninguna *Especialidad*.

addEspecialidad Añade una nueva especialidad a la *Clínica*. No comprueba si la especialidad ya existe para esta clínica.

buscarEspecialidadPaciente Devuelve la especialidad en la cual está ingresado un paciente dado o NULL si no se encuentra al paciente.

obtenerEspecialidad Devuelve la especialidad con el nombre dado, o NULL si ésta no existe.

Especialidad

Una especialidad se crea con un nombre de especialidad y un jefe de especialidad (un médico) dados. El jefe de especialidad es el primer médico que se adscribe a esta especialidad. El resto de médicos se adscribirán mediante el método *addMedico*.

[1] `http://www.cplusplus.com/reference/clibrary/ctime/time.html`

asignarHabitacion Para asignar una habitación a un paciente recorreremos las habitaciones, intentando asignar una cama de la habitación al paciente, parando cuando una habitación tiene hueco y se consigue realizar esta asignación. Las camas de cada habitación se adjudican de forma secuencial, primero la cama 0, luego la cama 1, y así sucesivamente. Sólo en el caso de conseguir encontrar una habitación para el paciente el método devolverá el valor cierto. Puede que un paciente se quede sin habitación, en cuyo caso el método devolverá falso.

asignarMedico En este caso siempre se asignará algún médico al paciente. Se asignará siempre el médico que tenga menor carga de pacientes. Sabemos que al menos hay un médico por especialidad, que es el jefe de especialidad. Cuando haya dos o más médicos que tengan la misma carga, se seleccionará el médico con menor nombre y apellidos en orden lexicográfico.

addHabitacion Añade una habitación a una especialidad.

ingresarPaciente Consiste en asignarle primero una habitación de la especialidad (*asignarHabitacion*). Si hay habitaciones libres entonces se le asigna un médico de la especialidad (*asignarMedico*) y el método devuelve cierto (el paciente ha ingresado). Si no hay habitaciones el paciente no ingresa y el método devuelve falso.

addMedico Añade un médico a la especialidad.

getPacientesListosParaAlta Devuelve todos los pacientes que están curados (*Paciente::estaCurado*). El orden de los pacientes en el vector resultado es irrelevante.

darAlta Para dar el alta a un paciente se debe comprobar que está curado, y en ese caso se desvincula de su médico y de la cama de la habitación que ocupaba, en cuyo caso el método devuelve cierto. Si el paciente no existe en esta especialidad o no está curado, el método no hace nada y devuelve falso.

buscarHabitacion Localiza y devuelve la habitación donde está ingresado el paciente dado o NULL si el paciente no tiene asignada ninguna habitación en esta especialidad (bien porque ha sido dado de alta o porque no ingresó en esta especialidad).

buscarMedico Localiza y devuelve el médico que trata al paciente o NULL si el paciente no tiene asignado ningún médico en esta especialidad (bien porque ha sido dado de alta o porque no ingresó en esta especialidad).

estaIngresado Devuelve cierto si el paciente dado está ingresado en esta especialidad. o falso en caso contrario.

Habitacion

Esta clase contiene un campo de clase denominado *siguienteNumero* que se establece inicialmente a 1. Su cometido es mantener un contador que indica el siguiente número de habitación a asignar. Es privado porque sólo desde esta clase es necesario acceder a dicho contador.

Constructor por defecto Se obtiene el *siguienteNumero* de habitación y se incrementa éste en uno.

getNumCamas Devuelve el número de camas que tiene la habitación.

addCama Añade 'n' camas nuevas a la habitación.

getCama Devuelve la i-ésima cama de la habitación o NULL si esta cama no existe.

ingresaPaciente Se busca la siguiente cama libre donde ingresar al paciente, devolviendo falso cuando no hay ninguna cama libre.

getNumeroCamaPaciente Devuelve el número de cama que tiene asignado al paciente dado, o -1 si el paciente no se encuentra. El número de cama se corresponde con su índice en el vector de camas, comenzando desde 0.

quitarPaciente Si el paciente está en esta habitación, se deja libre la cama que ocupaba y se devuelve cierto. Si el paciente no se encuentra, se retorna falso.

Cama

getPaciente Devuelve el paciente que ocupa esta cama, o NULL si la cama está libre.

setPaciente Asigna un paciente a la cama si esta está libre, en cuyo caso devuelve cierto. Si la cama está ocupada por otro paciente, no hace nada y devuelve falso.

isOcupada Devuelve cierto si la cama está ocupada por algún paciente.

Médico

Un médico se crea indicando su nombre y apellidos.

addPaciente Añade el paciente dado al conjunto de pacientes asignados al médico.

getCargaPacientes Devuelve el número de pacientes que este médico tiene asignados.

quitarPaciente Quita el paciente dado del conjunto de pacientes y devuelve cierto. Si el paciente no se encuentra retorna falso.

tieneAlPaciente Devuelve cierto si el paciente dado está entre el conjunto de pacientes que este médico tiene asignado.

Paciente

Un *Paciente* se crea indicando su número SIP, su nombre y apellidos, sexo ('M' para mujeres, 'H' para hombres) y fecha de nacimiento.

getEdad Devuelve el número entero de años del paciente a fecha de hoy.

operator< Devuelve cierto cuando el NSIP de éste paciente es menor lexicográficamente que el del paciente dado, o falso en caso contrario.

operator== Devuelve cierto cuando el NSIP de éste paciente es igual al del paciente dado, o falso en caso contrario.

getUltimoDiagnostico Devuelve el último diagnóstico asignado al paciente o NULL si no tiene aún asignado ningún diagnóstico.

diagnosticar Asigna un nuevo diagnóstico al paciente.

estaCurado Devuelve cierto si el último diagnóstico es un alta (*Diagnostico::esAlta()*), falso si no hay diagnósticos o si el último diagnóstico no es un alta.

operator<< El operador de salida debe mostrar en el flujo de salida la concatenación, separados por puntos y coma, del número SIP, el nombre y apellidos, sexo, fecha de nacimiento, edad del paciente y el tipo del último diagnóstico, seguido de un salto de línea. P.ej, sea el paciente Luis Martinez Perez, con SIP nº 1234567, nacido el 3 de Junio de 1986, que ha sido diagnosticado leve. Se deberá mostrar lo siguiente

```
1234567;Luis Martinez Perez;M;3/6/1986;21;Leve
```

Diagnóstico

Un diagnóstico se compone de una descripción textual y un tipo de diagnóstico (alta, leve, grave, muy grave). Inicialmente no tiene ningún tratamiento asociado.

esAlta Devuelve cierto si el tipo de tratamiento asociado es *tdAlta*.

tratar Asigna un nuevo tratamiento para este diagnóstico. Si existiere un tratamiento anterior, será sustituído por el nuevo.

NOTA: El tipo enumerado *TipoDiagnostico* se debe definir en el fichero .h de la clase *Diagnostico*.

Tratamiento

Un tratamiento consta únicamente de un texto que lo describe.

4. El programa principal

En esta primera práctica, el proceso de crear la clínica con sus especialidades, médicos y habitaciones, así como los procesos de ingreso, diagnóstico, tratamiento y alta de pacientes tienen lugar en el programa principal. Adjunto al enunciado tienes un programa principal de ejemplo (fichero `main.cc`).

4.1. Diagnóstico, tratamiento y alta

Tras ingresar al paciente y asignarle un médico, y sólo tras estos pasos, se puede añadir un diagnóstico al paciente. Siempre conoceremos cuál es el último diagnóstico del paciente porque los guardamos sequencialmente. Con el último diagnóstico sabremos si el paciente está curado (el diagnóstico es *alta*), si tiene pendiente la asignación de un tratamiento, y en el caso de tener un tratamiento, acceder a él. El tratamiento de un diagnóstico se puede cambiar en cualquier momento.

5. Documentación

Se ha de incluir en los ficheros fuente todos los comentarios necesarios en formato Doxygen. Estos comentarios deben estar en su versión corta y detallada, y deben definirse para:

Ficheros debe incluir nombre y dni de los autores.

Clases propósito de la clase: 3 líneas

Operaciones 1 línea para funciones triviales, y 2 líneas + parámetros entrada, parámetros de salida y funciones dependientes para operaciones más complejas.

Atributos propósito de cada uno de ellos: 1 línea

6. Estructura de directorios

La práctica debe ir organizada en tres directorios:

`include` contiene los ficheros `Cama.h, Clinica.h, Diagnostico.h, Especialidad.h, Fecha.h, Habitacion.h, Medico.h, Paciente.h, Tratamiento.h`

`lib` contiene los ficheros `Cama.cc, Clinica.cc, Diagnostico.cc, Especialidad.cc, Fecha.cc, Habitacion.cc, Medico.cc, Paciente.cc, Tratamiento.cc`

`src` contiene el fichero `main.cc`

Además, al ejecutar la herramienta Doxygen se generará un cuarto directorio `html` que contendrá la documentación en html.

7. Normas de evaluación

Para poder evaluar la práctica con las pruebas que acompañan a este enunciado, se debe implementar en el lenguaje C++. Se recomienda desarrollarla en un sistema operativo GNU/Linux, utilizando el compilador de C++ de GNU (g++).

Las pruebas unitarias que se pueden encontrar en el autocorrector de esta práctica sólo comprueban una pequeña parte de los casos. El alumno deberá completar estas pruebas con las necesarias para evaluar exhaustivamente la práctica, diseñando casos de prueba para todos los métodos a implementar.

8. Requisitos mínimos para obtener una evaluación positiva

- La práctica debe funcionar sin errores. En particular, no se debe producir ningún tipo de error del tipo *segmentation fault*, *null pointer assignment*, etc.
- No se deben utilizar variables globales.
- Ninguna operación debe emitir ningún tipo de comentario o mensaje por salida estándar (*cout*). Las operaciones que requieran escribir mensajes en pantalla deben recibir un parámetro `ostream&` que será el que se utilice para emitir dichos mensajes. Deben evitarse también los mensajes por la salida de error.
- Los ficheros de código fuente serán compilados en el momento de la corrección para generar el ejecutable. Por ello, es de vital importancia que la práctica compile con el *makefile* proporcionado en el autocorrector que acompaña a este enunciado. Si no se puede generar el ejecutable, la práctica no puede ser evaluada con las herramientas adjuntas.
- Se debe respetar de manera estricta el formato del nombre de *todas las propiedades* (públicas, protegidas y privadas) de las clases, tanto en cuanto a ámbito de visibilidad como en cuanto a tipo y forma de escritura.
- La práctica debe estar suficientemente documentada, de manera que el contenido de la documentación que se genere mediante el comando `make doc` (y el fichero `Doxyfile` proporcionado junto con este enunciado) sea significativo.

9. Aclaraciones

- Se pueden añadir los atributos y métodos *privados* que se consideren imprescindibles para una correcta implementación. No obstante, eso no exime de implementar TODOS los métodos presentes en el enunciado, ni de asegurarse de que funcionan tal y como se espera, incluso si no se utilizan nunca en la implementación de la práctica.