ÍNDICE

Práctica 1. Introducción a la programación con C++ 2
Práctica 2. Introducción al diseño software 6
Práctica 3. Identificación de clases en especificación de requerimientos 15
Práctica 4. Implementación del sistema 22
Práctica 5. Creación de clase Coleccion con operadores sobrecargados 26
Práctica 6. Uso de plantilla set de STL para almacenar objetos Usuario 34
Práctica 7. Excepciones, depuración y documentación 36
Práctica 8. Uso de ficheros 39
Práctica 9. Uso de patrones de diseño 41

Práctica 1. Introducción a la programación con C++

©2024 **Julio Vega Pérez**
Algunos derechos reservados.
Este trabajo se entrega bajo licencia CC-BY-SA 4.0.

Realiza los siguientes ejercicios y entrégalos, sin crear ninguna carpeta, bajo el nombre que corresponda, siguiendo la sintaxis `ejercicioX.cpp`; por ejemplo, si es el Ejercicio 1, se debería llamar `ejercicio1.cpp`. En aquellos ejercicios que tengan varios apartados, añade la letra correspondiente al apartado; por ejemplo, en el Ejercicio 2, tendríamos `ejercicio2a.cpp`, entre otros.

Ejercicio 1

Escribe un programa que pida al usuario que escriba dos números, que obtenga los números del usuario e imprima la suma, producto, diferencia y cociente de los números.

Ejercicio 2

Escribe un programa que imprima los números del 1 al 4 en la misma línea, con cada par de números adyacentes separado por un espacio; esto es: `1 2 3 4`. Haz esto de varias formas:

a. Utilizando una instrucción con un operador de inserción de flujo (`<<`).

b. Utilizando una instrucción con cuatro operadores de inserción de flujo.

c. Utilizando cuatro instrucciones.

Ejercicio 3

Escribe un programa que pida al usuario que escriba dos enteros, que obtenga los números del usuario e imprima el número más grande, seguido de la expresión *es más grande*. Y, si los números son iguales, que imprima el mensaje *Estos números son iguales*.

Ejercicio 4

Escribe un programa que reciba tres enteros del teclado e imprima la suma, promedio, producto, menor y mayor de esos números. El diálogo de la pantalla debe aparecer de la siguiente manera:

```
Introduzca tres enteros distintos: 13 27 14
La suma es 54
El promedio es 18
El producto es 4914
El menor es 13
El mayor es 27
```

Ejercicio 5

Escribe un programa que lea el radio de un círculo como un número entero y que imprima su diámetro, circunferencia y área. Usa el valor constante 3,14159 para π. Realiza todos los cálculos en instrucciones de salida.

Ejercicio 6

Escribe una aplicación que muestre un patrón de tablero de ajedrez como el que se muestra a continuación. Hazlo primero empleando ocho instrucciones de salida (`cout`), y después hazlo intentando utilizar el menor número de instrucciones posible.

```
* * * * * * * *
 * * * * * * * *
* * * * * * * *
 * * * * * * * *
* * * * * * * *
 * * * * * * * *
* * * * * * * *
 * * * * * * * *
```

Ejercicio 7

Escribe un programa que reciba como entrada un número entero de cinco dígitos, que separe ese número en sus dígitos individuales y los imprima, cada uno separado de los demás por tres espacios. Por ejemplo, si el usuario escribe el número 31250, el programa debe imprimir:`3   1   2   5   0`.

Ejercicio 8

El factorial de un entero n no negativo se escribe como $n!$ (n factorial) y se define de la siguiente manera: $n! = n(n–1)(n–2)...1$ (para valores de n mayores o iguales a 1) y $n! = 1$ (para $n = 0$ o $n = 1$).

Por ejemplo, $5! = 54321$, que es 120. Usa instrucciones `while` en cada uno de los siguientes casos:

a. Escribe una aplicación que lea un entero no negativo, que calcule e imprima su factorial.

b. Escribe un programa que estime el valor de la constante matemática e, utilizando la fórmula: $e = 1 + \frac{1}{1!} + \frac{1}{2!} + \frac{1}{3!} + ...$ Pide al usuario la precisión deseada de e (es decir, el número de términos en la suma).

c. Escribe una aplicación que calcule el valor de e^x, utilizando la fórmula: $e^x = 1 + \frac{x}{1!} + \frac{x^2}{2!} + \frac{x^3}{3!} + ...$ Pide al usuario la precisión deseada de e (es decir, el número de términos en la suma).

Ejercicio 9

Calcula el valor de π a partir de la serie infinita: $\pi = 4 - \frac{4}{3} + \frac{4}{5} - \frac{4}{7} + \frac{4}{9} - \frac{4}{11} + ...$ Imprime una tabla que muestre el valor aproximado de π, después de cada uno de los primeros 1000 términos de esta serie.

Ejercicio 10

Escribe un programa que imprima la siguiente figura de rombo. Puedes utilizar instrucciones de salida que impriman un solo asterisco (*) o un solo espacio en blanco. Maximiza el uso de la repetición (con instrucciones `for` anidadas), y minimiza el número de instrucciones de salida.

```
 *
***
```

```
  *****
 *******
*********
 *******
  *****
   ***
    *
```

Bonus: haz una mejora del programa (`ejercicio10pro.cpp`) para que lea un número impar en el rango de 1 a 19, correspondiente al número de filas que configuran el rombo, y después muestre un rombo del tamaño apropiado.

Práctica 2. Introducción al diseño software

©2024 **Julio Vega Pérez**
Algunos derechos reservados.
Este trabajo se entrega bajo licencia CC-BY-SA 4.0.

1. Introducción

Comenzamos la segunda práctica de la asignatura con el análisis, desde el punto de vista del diseño software, de un problema práctico. Este problema lo iremos resolviendo poco a poco en sucesivas prácticas.

El problema es el siguiente. Supongamos que, desde una empresa, se te solicita que desarrolles un sistema software para dar soporte a una completa infraestructura de sensores. Para ello, tendrás que diseñar el sistema que da solución al problema y, posteriormente, implementarlo siguiendo el diseño que hayas elegido y empleando las técnicas de la programación orientada a objetos (POO) en C++ que veremos durante el curso.

El desarrollo de este diseño te ayudará a acostumbrarte a los tipos de problemas que se dan en el mundo laboral, así como saber dar respuesta a ellos. Empezaremos el proceso de diseño especificando los requerimientos del sistema; esto es, cuál es el propósito general del sistema multisensorial y qué se supone que debe hacer este.

2. Especificación de requerimientos

Una empresa de Fuenlabrada, denominada *Julio Veganos e Hijos*, dedicada al desarrollo autosostenible, pretende implantar un sistema multisensorial en un invernadero, de forma que pueda tener controladas todas las variables climatológicas que se dan en el mismo, tanto

presencial como remotamente. Además, este sistema debe incorporar los mecanismos necesarios para controlar la seguridad de la empresa: alarma, detección de puertas/ventanas abiertas, etc.

Cualquier empleado de la empresa podrá acceder —mediante autenticación— al interfaz del sistema para acceder a todos los datos de los componentes descritos. Esto supone que cada empleado solo puede tener una cuenta en el sistema.

La interfaz de usuario contiene los siguientes componentes hardware:

- Una pantalla que muestre todos los datos.
- Un teclado que permite la entrada de datos al usuario (lo primero de todo, su datos de *login*).
- Un micrófono que permite dar instrucciones y que sean escuchadas desde las distintas estaciones remotas del sistema.
- Un interruptor o similar que permita activar/desactivar cómodamente la alarma de la empresa.

Esta empresa te pide que desarrolles el software que permita leer los datos de los distintos sensores y cámaras instalados en distintas ubicaciones de su plantación, y los vierta de la forma más *amigable* posible mediante un interfaz gráfico. Este software debe encapsular la funcionalidad de los distintos componentes hardware dentro de los correspondientes componentes software, sin preocuparse por cómo estos dispositivos realizan sus tareas.

Esta plataforma no está desarrollada aún, por lo que en lugar de desarrollar el software para que sea ejecutado en el sistema final multisensorial, deberás desarrollar una primera versión del mismo para que —simplemente— se ejecute en el ordenador. Piensa que la versión definitiva deberá poder ejecutarse en la placa (e.g. Arduino, Raspberry, etc.) que se use en las estaciones remotas, pero esa no va a ser nuestra tarea. Esta versión *beta* utilizará el monitor del PC para simular el interfaz del sistema, el teclado como entrada de datos, y distintos algoritmos para simular los datos que verterían los distintos dispositivos sensoriales.

Una sesión con el interfaz del sistema consiste en la autenticación de un usuario (un empleado de la empresa); esto es, proporcionar la identidad del usuario con base en un número de identificación de empleado, seguida del *dashboard* o parrilla en la que se muestran de forma gráfica los datos que vierten los sensores conectados al sistema. Para autenticar un

usuario y que este pueda acceder al sistema, el programa debe interactuar con la base de datos de información sobre los empleados registrados en la empresa.

Nota: una base de datos es una colección organizada de datos almacenados en un ordenador/servidor.

Para cada empleado, la base de datos almacena un número de usuario, un NIF y un *timestamp* que indica la fecha del último acceso de este usuario al sistema.

Nota: para simplificar, vamos a asumir que la empresa planea construir sólo un sistema multisensorial, en su invernadero de Fuenlabrada, por lo que no necesitamos preocuparnos por que varias delegaciones accedan a esta base de datos al mismo tiempo. Lo que es más, vamos a suponer que la empresa no va a realizar modificaciones en la información que hay en la base de datos mientras un usuario accede al sistema. Además, cualquier sistema comercial como este se topa con cuestiones de seguridad con una complejidad razonable, las cuales van más allá del alcance de los objetivos de esta asignatura. No obstante, para simplificar nuestro ejemplo vamos a suponer que la empresa confía en este sistema para que acceda a la información de la base de datos y la manipule sin necesidad de medidas de seguridad considerables.

Al acercarse al puesto de acceso al sistema, el usuario deberá experimentar la siguiente secuencia de eventos:

1. La pantalla muestra un mensaje de bienvenida y pide al usuario que introduzca un número de empleado.

2. El usuario introduce un número de empleado de cinco dígitos, mediante el uso del teclado numérico.

3. En la pantalla aparece un mensaje, en el que se pide al usuario que introduzca su NIF (número de identificación fiscal) asociado con el número de empleado especificado.

4. El usuario introduce un NIF de ocho dígitos mediante el teclado numérico.

5. Si el usuario introduce un número de empleado válido y el NIF correcto para ese usuario, la pantalla muestra el *dashboard*. Si el usuario introduce un número de empleado inválido o un NIF incorrecto, la pantalla muestra un mensaje apropiado y después el sistema regresa al paso 1 para reiniciar el proceso de autenticación.

Una vez que el sistema autentica al usuario, el *dashboard* debe contener información actualizada de los valores vertidos por los sensores conectados al sistema, así como las imágenes

en tiempo real vertidas por las distintas cámaras de seguridad distribuidas en la finca de la empresa. También debe haber una opción para poder salir del sistema en cualquier momento.

Las opciones que muestra la pantalla principal dependerá del número de sensores que estén conectados. Se va a considerar que debe haber un mínimo de cuatro sensores conectados y dos cámaras. Pero hay que tener en cuenta que, en cualquier momento de la vida del sistema, este número puede aumentar, y habría que dar cabida de forma dinámica a los nuevos componentes. Así, las opciones que habrá por defecto en el *dashboard* son:

1. Temperatura.
2. Humedad.
3. Calidad del aire.
4. Nivel de iluminación.
5. Cámara RGB.
6. Cámara térmica.
7. Salir.

Las opciones deben estar numeradas para permitir al usuario acceder a la información de cada componente y visualizarla de forma más detallada en pantalla completa. Si el usuario introduce una opción inválida, la pantalla muestra un mensaje de error y vuelve a mostrar el menú principal. Si introduce la opción de *Salir*, se cierra el sistema y se vuelve a la pantalla de bienvenida. Por último, si el usuario introduce cualquier opción válida, se debe cambiar a la pantalla de información detallada, donde se muestre el gráfico con más detalle así como los datos registrados por ese sensor durante la última hora; además de una opción para *Volver al menú*.

3. Análisis del sistema

En la declaración anterior se presentó un ejemplo simplificado de una especificación de requerimientos. Por lo general, dicho documento es el resultado de un proceso detallado de recopilación de requerimientos, el cual podría incluir entrevistas con usuarios potenciales del sistema y especialistas en campos relacionados con el mismo. Por ejemplo, un analista de sistemas que se contrate para preparar una especificación de requerimientos para software de sistemas multisensoriales o videovigilancia (como el sistema que describimos aquí) podría

entrevistar a expertos en seguridad, domótica o robótica para obtener una mejor comprensión de qué es lo que debe hacer el software. El analista utilizaría la información recopilada para compilar una lista de requerimientos del sistema y, con ello, guiar a los diseñadores de sistemas en el proceso de diseño del mismo.

El proceso de recopilación de requerimientos es una tarea clave de la primera etapa del ciclo de vida del software. El ciclo de vida del software especifica las etapas a través de las cuales el software evoluciona, desde el tiempo en que fue concebido hasta el tiempo en que se retira de su uso. Por lo general, estas etapas incluyen: análisis, diseño, implementación, prueba y depuración, despliegue, mantenimiento y retirada. Existen varios modelos de ciclo de vida del software, cada uno con sus propias preferencias y especificaciones respecto a cuándo y con qué frecuencia deben llevar a cabo los ingenieros de software las diversas etapas. Los modelos de cascada realizan cada etapa de forma sucesiva, mientras que los modelos iterativos pueden repetir una o más etapas varias veces a lo largo del ciclo de vida de un producto.

La etapa de análisis del ciclo de vida del software se enfoca en definir el problema a resolver. Al diseñar cualquier sistema, uno debe resolver el problema de la manera correcta, pero de igual manera uno debe resolver el problema correcto. Los analistas de sistemas recolectan los requerimientos que indican el problema específico a resolver. Nuestra especificación de requerimientos describe nuestro sistema con el suficiente detalle como para tú no necesites pasar por una etapa de análisis exhaustiva; esto ya se hizo previamente.

Para capturar lo que debe hacer un sistema propuesto, los desarrolladores emplean a menudo una técnica conocida como *modelado de casos de uso*. Este proceso identifica los casos de uso del sistema, cada uno de los cuales representa una capacidad distinta que el sistema provee a sus clientes. Por ejemplo, es común que un sistema multisensorial tenga varios casos de uso, como *Ver temperatura*, *Ver humedad*, *Ver calidad del aire*, etc. El sistema simplificado que construiremos requiere seis casos de uso (Figura 1).

Cada uno de los casos de uso describe un escenario común en el cual el usuario utiliza el sistema. Ya vimos las descripciones de los casos de uso del sistema en la especificación de requerimientos; las listas de pasos requeridos para realizar cada acción describen en realidad los seis casos de uso de nuestro sistema.

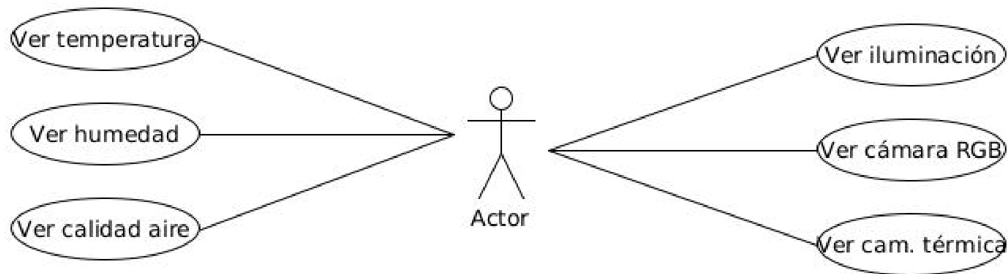


Figura 1: Diagrama de casos de uso para el sistema multisensorial, desde la perspectiva del usuario

4. Diagramas de casos de uso

Ahora presentaremos el primero de varios diagramas de UML para nuestro ejemplo práctico. Vamos a crear un diagrama de casos de uso para modelar las interacciones entre los clientes de un sistema (en este ejemplo práctico, los empleados de la empresa) y el sistema. El objetivo es mostrar los tipos de interacciones que tienen los usuarios con un sistema sin proveer los detalles; éstos se mostrarán en otros diagramas de UML (los cuales presentaremos a lo largo del ejemplo práctico). A menudo, los diagramas de casos de uso se acompañan de texto informal que describe los casos de uso con más detalle, como el texto que aparece en la especificación de requerimientos. Los diagramas de casos de uso se producen durante la etapa de análisis del ciclo de vida del software. En sistemas más grandes, los diagramas de casos de uso son herramientas simples pero indispensables que ayudan a los diseñadores de sistemas a enfocarse en satisfacer las necesidades de los usuarios.

La Figura 1 muestra el diagrama de casos de uso para nuestro sistema. La figura humana representa a un actor, el cual define los roles que desempeña una entidad externa (como una persona u otro sistema) cuando interactúa con el sistema. Para nuestro programa, el actor es un usuario que puede ver la temperatura, la humedad, la calidad del aire, etc. El usuario no es una persona real, sino que constituye los roles que puede desempeñar una persona real (al desempeñar el papel de un usuario) mientras interactúa con el sistema. Hay que tener en cuenta que un diagrama de casos de uso puede incluir varios actores. Por ejemplo, el diagrama de casos de uso para un sistema como el planteado para una empresa real podría incluir también un actor llamado *Administrador*, que pueda —entre otras cosas— modificar los datos del perfil de cada *Usuario*.

Para identificar al actor en nuestro sistema, debemos examinar la especificación de requerimientos, la cual dice: *los usuarios del sistema deben poder ver la temperatura, la humedad,*

etc.. Por lo tanto, el actor en cada uno de estos seis casos de uso es el *Usuario* que interactúa con el sistema. Una entidad externa (una persona real) desempeña el papel del usuario para realizar tales acciones. La Figura 1 muestra un actor, cuyo nombre (*Usuario*) aparece debajo del actor en el diagrama. UML modela cada caso de uso como un óvalo conectado a un actor con una línea sólida.

Los ingenieros de software (más específicamente, los diseñadores de sistemas) deben analizar la especificación de requerimientos o un conjunto de casos de uso, y diseñar el sistema antes de que los programadores lo implementen. Durante la etapa de análisis, los analistas de sistemas se enfocan en comprender la especificación de requerimientos para producir una especificación de alto nivel que describa qué es lo que el sistema debe hacer. El resultado de la etapa de diseño (una especificación de diseño) debe especificar claramente cómo debe construirse el sistema para satisfacer estos requerimientos. En las siguientes prácticas llevaremos a cabo los pasos de un proceso simple de diseño orientado a objetos (DOO) con el sistema multisensorial planteado, para producir una especificación de diseño que contenga una colección de diagramas de UML y texto de apoyo. Recuerda que UML está diseñado para utilizarse con cualquier proceso de DOO. Existen muchos de estos procesos de desarrollo software, de los cuales el más conocido es *Rational Unified Process (RUP)*, desarrollado por Rational Software Corporation (ahora una división de IBM). RUP es un proceso robusto guiado por los casos de uso para diseñar aplicaciones a nivel industrial. Para este ejemplo práctico, presentaremos nuestro propio proceso de diseño simplificado.

5. Diseño del sistema multisensorial

Ahora comenzaremos la etapa de diseño de nuestro sistema multisensorial. Un sistema es un conjunto de componentes que interactúan para resolver un problema. Por ejemplo, para realizar sus tareas designadas, nuestro sistema tiene una interfaz de usuario, contiene software para ejecutar diversas acciones e interactúa con una base de datos de información de los empleados de la empresa. La estructura del sistema describe los objetos del sistema y sus interrelaciones. El comportamiento del sistema describe la manera en que cambia el sistema a medida que sus objetos interactúan entre sí. Todo sistema tiene tanto estructura como comportamiento; los diseñadores deben especificar ambos. Existen varios tipos distintos de estructuras y comportamientos de un sistema. Por ejemplo, las interacciones entre los objetos en el sistema son distintas a las interacciones entre el usuario y el sistema, pero aun así ambas constituyen una porción del comportamiento del sistema.

El estándar UML 2 especifica 13 tipos de diagramas para documentar los modelos de

sistemas. Cada tipo de diagrama modela una característica distinta de la estructura o del comportamiento de un sistema: seis diagramas se relacionan con la estructura del sistema; los siete restantes se relacionan con su comportamiento. A continuación se describen sólo los seis tipos de diagramas que utilizaremos en nuestro ejemplo práctico, uno de los cuales (el diagrama de clases) modela la estructura del sistema, mientras que los otros cinco modelan el comportamiento:

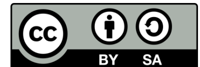
- Los **diagramas de casos de uso**, como el de la Figura 1, modelan las interacciones entre un sistema y sus entidades externas (actores) en términos de casos de uso (capacidades del sistema, como *Ver la temperatura*, *Ver la humedad*, etc.).
- Los **diagramas de clases** modelan las clases —o bloques de construcción— que se utilizan en un sistema. Cada sustantivo u objeto que se describe en la especificación de requerimientos es candidato para ser una clase en el sistema (por ejemplo, *CuentaDeEmpleado*, *Teclado*). Los diagramas de clases nos ayudan a especificar las relaciones estructurales entre las partes del sistema. Por ejemplo, el diagrama de clases del sistema multisensorial especificará que este está compuesto físicamente de una pantalla y un teclado.
- Los **diagramas de máquina de estado**, modelan las formas en que un objeto cambia de estado. El estado de un objeto se indica mediante los valores de todos los atributos del objeto en un momento dado. Cuando un objeto cambia de estado, puede comportarse de manera distinta en el sistema. Por ejemplo, después de validar el NIF de un usuario, el sistema cambia del estado *usuario no autenticado* al estado *usuario autenticado*, punto en el cual el sistema permite al usuario realizar las operaciones de monitorización que desee.
- Los **diagramas de actividad** modelan la actividad de un objeto: el flujo de trabajo (secuencia de eventos) del objeto durante la ejecución del programa. Un diagrama de actividad modela las acciones que realiza el objeto y especifica el orden en el cual desempeña estas acciones. Por ejemplo, un diagrama de actividad muestra que el sistema debe obtener los datos del usuario (de la base de datos de información de las cuentas de los trabajadores de la empresa) antes de que la pantalla pueda mostrar las acciones disponibles de monitorización al usuario.
- Los **diagramas de comunicación** (llamados diagramas de colaboración en versiones anteriores de UML) modelan las interacciones entre los objetos en un sistema, con un énfasis acerca de qué interacciones ocurren. Por ejemplo, el sistema debe comunicarse con la base de datos de información de las cuentas de los trabajadores de la empresa para obtener los datos de estos.

- Los **diagramas de secuencia** modelan también las interacciones entre los objetos en un sistema, pero a diferencia de los diagramas de comunicación, enfatizan en cuándo ocurren las interacciones. Por ejemplo, la pantalla pide al usuario que seleccione la opción de monitorización que desea visualizar antes de efectuar tal operación.

Ejercicio

Una vez entendida la teoría de cómo funciona un desarrollo software, implementa un primer borrador del diagrama de clases que modelará la estructura del sistema de monitorización planteado. En esta primera práctica, deberás realizar este borrador a mano y, una vez lo tengas terminado, hazle una foto y súbelo al repositorio de esta práctica.

Práctica 3. Identificación de las clases en la especificación de requerimientos



©2024 **Julio Vega Pérez**
Algunos derechos reservados.
Este trabajo se entrega bajo licencia CC-BY-SA 4.0.

1. Introducción

Ahora empezaremos a diseñar el sistema de monitorización que presentamos en la Práctica 2. En esta práctica identificaremos las clases necesarias para crear el sistema planteado, analizando los sustantivos y las frases nominales que aparecen en la especificación de requerimientos. Presentaremos los diagramas de clases de UML para modelar las relaciones entre estas clases. Este primer paso es importante para definir la estructura de nuestro sistema.

2. Identificación de las clases en un sistema

Para comenzar nuestro proceso de DOO, vamos a identificar las clases requeridas para crear el sistema multisensorial. Más adelante describiremos estas clases mediante el uso de los diagramas de clases de UML y las implementaremos en C++. Primero debemos revisar la especificación de requerimientos que ya hicimos en la Práctica 2 e identificar los sustantivos y frases nominales clave que nos ayuden a identificar las clases que conformarán el sistema. Tal vez decidamos que algunos de estos sustantivos y frases nominales sean atributos de otras clases en el sistema. Tal vez también concluyamos que algunos de los sustantivos no corresponden a ciertas partes del sistema y, por ende, no deben modelarse. A medida que avancemos por el proceso de diseño podemos ir descubriendo clases adicionales.

Actividad 1: Comienza por confeccionar la lista de los sustantivos y frases nominales que se encuentran en la especificación de requerimientos (Práctica 2).

Vamos a crear clases sólo para los sustantivos y frases nominales que tengan importancia en el sistema. Por ejemplo, no modelamos la *empresa* como una clase, ya que la empresa no es una parte del sistema; la empresa sólo quiere que nosotros construyamos el sistema multisensorial. *Cliente* y *usuario* también representan entidades fuera del sistema; son importantes debido a que interactúan con nuestro sistema, pero no necesitamos modelarlos como clases en el software del sistema. Recuerda que modelamos un usuario del sistema (es decir, un trabajador de la empresa) como el actor en el diagrama de casos de uso de la figura que representamos en la Práctica 2.

En nuestro sistema, que incluye la recepción de datos de varios sensores, modelaremos las distintas acciones que pueden ser realizadas: *Ver temperatura*, *Ver humedad*, *Ver calidad del aire*, etc. como clases individuales. Estas clases poseen los atributos específicos necesarios para ejecutar las acciones que representan. Por ejemplo, para *Ver temperatura*, se necesita conocer el dato *temperatura* que el sensor de temperatura proporciona. Sin embargo, una solicitud de *Ver temperatura* no requiere datos adicionales. Lo que es más, las distintas clases correspondientes a las distintas acciones que se pueden realizar en el sistema muestran comportamientos únicos. Para *Ver temperatura* se requiere mostrar el valor *temperatura* al usuario, mientras que para *Ver humedad* se requiere mostrar el valor *humedad* al usuario.

> Nota: cuando veamos *polimorfismo*, factorizaremos las características comunes de todas las operaciones que se pueden hacer en nuestro sistema en una clase de *Ver dato* general, mediante el uso de los conceptos orientados a objetos de las clases abstractas y la herencia.

> Actividad 2: Determina las clases necesarias para implementar el sistema propuesto con base en los sustantivos y frases nominales que has confeccionado para la Actividad 1.

Con base en la lista que hemos creado en la Actividad 2, ya podemos modelar las clases en nuestro sistema. En el proceso de diseño escribimos los nombres de las clases con la primera letra en mayúscula (una convención de UML), como lo haremos cuando escribamos el código de C++ para implementar nuestro diseño. Si el nombre de una clase contiene más de una palabra, juntaremos todas las palabras y escribiremos la primera letra de cada una de ellas en mayúscula, lo que se conoce como *notación camello* (por ejemplo, `NombreConVariasPalabras`). Utilizando esta convención, vamos a crear las clases. Construiremos nuestro sistema mediante el uso de todas estas clases como bloques de construcción. Sin embargo, antes de empezar a construir el sistema, debemos comprender mejor la forma en que las clases se relacionan entre sí.

3. Modelado de las clases

UML nos permite modelar, a través de los diagramas de clases, las clases en el sistema ATM y sus interrelaciones. La Figura 1 representa a la clase `SistemaMonitorizacion`. En UML, cada clase se modela como un rectángulo con tres compartimentos. El compartimento superior contiene el nombre de la clase, centrado horizontalmente y en negrita. El compartimento intermedio contiene los atributos (datos) de la clase. El compartimento inferior contiene los métodos (funciones) de la clase. En la Figura 1, los compartimentos intermedio e inferior están vacíos, ya que no hemos determinado los atributos y métodos de esta clase todavía.

Figura 1: Representación de una clase en UML mediante un diagrama de clases

Los diagramas de clases también muestran las relaciones entre las clases del sistema. En la Figura 2 se muestra cómo nuestras clases `SistemaMonitorizacion` y `VisualizacionTemperatura` se relacionan una con la otra. Observa que los rectángulos que representan a las clases en este diagrama no están subdivididos en compartimentos. UML permite suprimir los atributos de las clases y sus operaciones de esta forma, cuando sea apropiado, para crear diagramas más legibles. Un diagrama de este tipo se denomina diagrama con elementos omitidos (*elided diagram*): su información, tal como el contenido de los compartimentos segundo y tercero, no se modela. Más adelante colocaremos información en estos compartimentos.

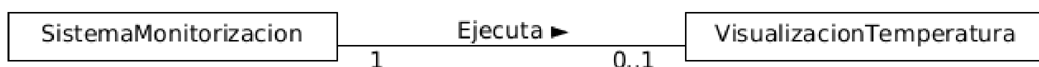


Figura 2: Diagrama de clases que muestra una asociación entre clases

Actividad 3: Modela el conjunto de las clases del sistema que habías determinado. Para ello, usaremos la herramienta *umlet* (Figura 3), ya instalada en los equipos del laboratorio y disponible para instalar desde los repositorios oficiales de cualquier distribución de Ubuntu (también en otros S.S.O.O.). Es una herramienta muy simple, básica, pero a la vez muy potente. Todos los elementos disponibles están a mano derecha; para usar cualquiera de ellos basta con hacer doble click sobre él o arrastrarlo. En la parte inferior derecha encontramos las propiedades del elemento actual, para modificarlo según las necesidades.

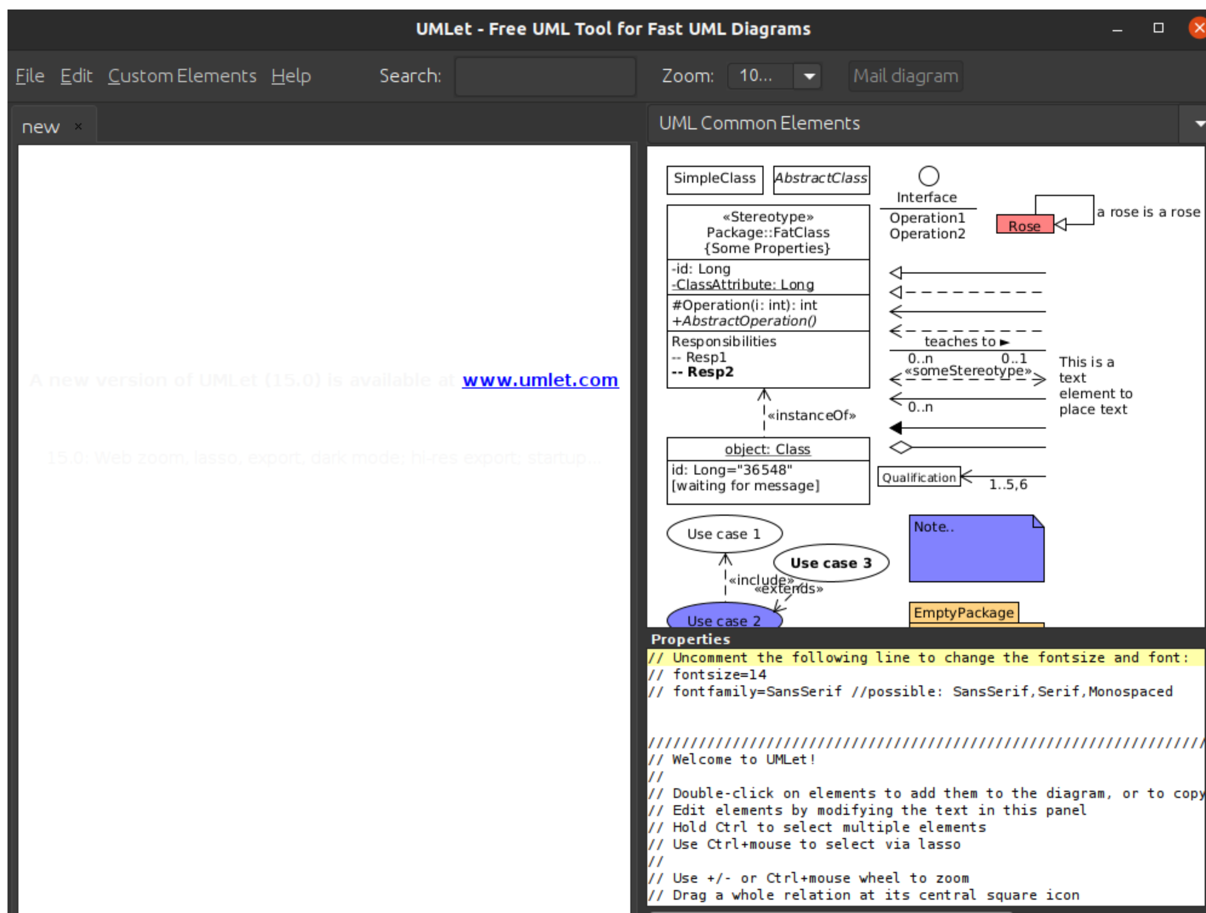


Figura 3: Interfaz de la herramienta umlet, de modelado UML

En la Figura 2, la línea sólida que conecta a las dos clases representa una **asociación**: una relación entre clases. Los números cerca de cada extremo de la línea son valores de **multiplicidad**; éstos indican cuántos objetos de cada clase participan en la asociación. En este caso, al seguir la línea de un extremo al otro se revela que, en un momento dado, un objeto `SistemaMonitorizacion` participa en una asociación con cero o con un objeto `VisualizacionTemperatura`; cero si el usuario actual no está realizando una operación de visualización de temperatura o si ha solicitado un tipo distinto de acción/visualización, y

uno si el usuario ha solicitado una operación de visualización de temperatura. UML puede modelar muchos tipos de multiplicidad. En la Tabla 1 se listan y explican los tipos de multiplicidad.

Símbolo	Significado
0	Ninguno
1	Uno
m	Un valor entero
0..1	Cero o uno
m, n	m o n
$m..n$	Cuando menos m, pero no más que n
$*$	Cualquier entero no negativo (cero o más)
$0..*$	Cero o más (idéntico a *)
$1..*$	Uno o más

Cuadro 1: Tipos de multiplicidad

Una asociación puede tener nombre. Por ejemplo, la palabra `Ejecuta` por encima de la línea que conecta a las clases `SistemaMonitorizacion` y `VisualizacionTemperatura` en la Figura 2 indica el nombre de esa asociación. Esta parte del diagrama se lee así: *un objeto de la clase SistemaMonitorizacion ejecuta cero o un objeto de la clase VisualizacionTemperatura*. Los nombres de las asociaciones son direccionales, como lo indica la punta de flecha rellena; por lo tanto, sería inapropiado, por ejemplo, leer la anterior asociación de derecha a izquierda como *cero o un objeto de la clase VisualizacionTemperatura ejecuta un objeto de la clase SistemaMonitorizacion*.

También se puede añadir una palabra en el extremo, en este caso, de `VisualizacionTemperatura` de la línea de asociación en la Figura 2. Esta palabra sería un nombre de rol, el cual identificaría el rol que desempeña el objeto `VisualizacionTemperatura` en su relación con el `SistemaMonitorizacion`. Un nombre de rol agrega significado a una asociación entre clases, ya que identifica el rol que desempeña una clase dentro del contexto de una asociación. Una clase puede desempeñar varios roles en el mismo sistema. Por ejemplo, en un sistema de personal de una universidad, una persona puede desempeñar el rol de *profesor* respecto a los estudiantes. La misma persona puede desempeñar el rol de *colega* cuando participa en una asociación con otro profesor, y de *entrenador* cuando entrena a los atletas estudiantes. Sin embargo, a menudo, los nombres de los roles se omiten en los diagramas de clases, ya que el significado de una asociación suele estar claro sin ellos.

Además de indicar relaciones simples, las asociaciones pueden especificar relaciones más

complejas, como cuando los objetos de una clase están compuestos de objetos de otras clases. Considera, por ejemplo, un sistema de monitorización real. ¿Qué piezas reúne un fabricante para construirlo? Nuestra especificación de requerimientos nos indica que el sistema está compuesto de una pantalla, un teclado, un micrófono y un interruptor.

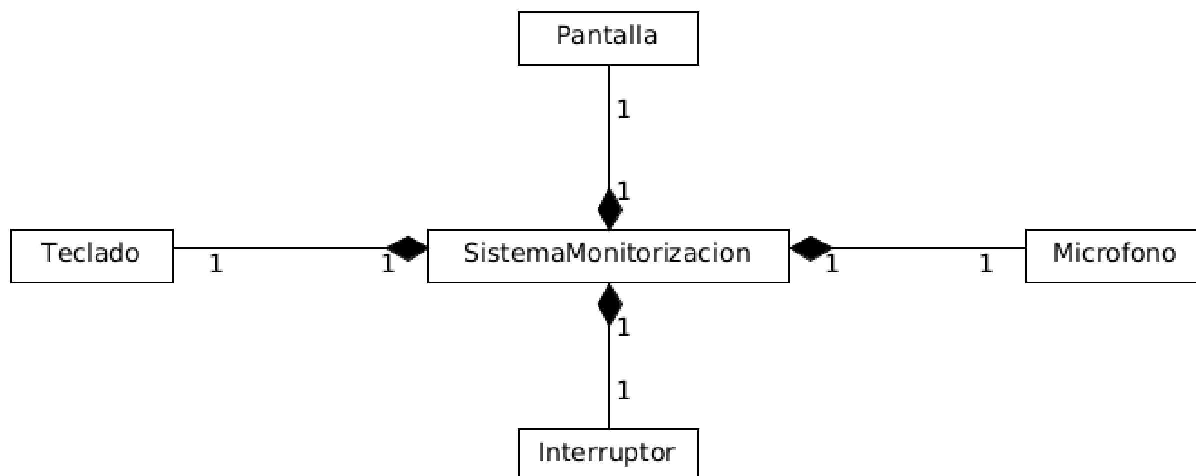


Figura 4: Diagrama de clases que muestra las relaciones de composición

En la Figura 4, los rombos sólidos que se adjuntan a las líneas de asociación de la clase `SistemaMonitorizacion` indican que esta clase tiene una **relación de composición** con las clases `Pantalla`, `Teclado`, `Microfono` e `Interruptor`. La composición implica una relación en todo/en parte. La clase que tiene el símbolo de composición (el rombo sólido) en su extremo de la línea de asociación es el todo (en este caso, `SistemaMonitorizacion`), y las clases en el otro extremo de las líneas de asociación son las partes; en este caso, las clases `Pantalla`, `Teclado`, `Microfono` e `Interruptor`. Las composiciones en la Figura 4 indican que un objeto de la clase `SistemaMonitorizacion` está formado por un objeto de la clase `Pantalla`, un objeto de la clase `Teclado`, un objeto de la clase `Microfono` y un objeto de la clase `Interruptor`. El sistema *tiene una* pantalla, un teclado, un micrófono y un interruptor. La relación *tiene un* define la composición (cuando veamos el tema de herencia y polimorfismo, veremos que la relación *es un* define la herencia).

De acuerdo con la especifi cación de UML, las relaciones de composición tienen las siguientes propiedades:

1. Solo una clase en la relación puede representar el todo; es decir, el rombo puede colocarse solo en un extremo de la línea de asociación. Por ejemplo, la pantalla es parte

del sistema o el sistema es parte de la pantalla, pero la pantalla y el sistema no pueden representar ambos el *todo* dentro de la relación.

2. Las partes en la relación de composición existen solo mientras exista el todo, y el todo es responsable de la creación y destrucción de sus partes. Por ejemplo, el acto de construir un sistema incluye la manufactura de sus partes. Lo que es mas, si el sistema se destruye, también se destruyen su pantalla, teclado, micrófono e interruptor.

3. Una parte puede pertenecer sólo a un todo a la vez, aunque esa parte puede quitarse y unirse a otro todo, el cual entonces asumirá la responsabilidad de esa parte.

Los rombos sólidos en nuestros diagramas de clases indican las relaciones de composición que cumplen con estas tres propiedades. Si una relación *tiene un* no satisface uno o más de estos criterios, UML especifica que se deben adjuntar rombos sin relleno a los extremos de las líneas de asociación para indicar una **agregación**: una forma más débil de la composición. Por ejemplo, una PC y un monitor de ordenador participan en una relación de agregación: la computadora *tiene un* monitor, pero las dos partes pueden existir en forma independiente, y el mismo monitor puede conectarse a varios PC a la vez, con lo cual se violan las propiedades segunda y tercera de la composición.

Ejercicio

Llegados a este punto, ya hemos identificado las clases en nuestro sistema de monitorización, aunque tal vez descubramos otras, a medida que avancemos con el diseño y la implementación. Como ejercicio final, completa el diagrama de clases para el sistema planteado. Este diagrama debe modelar las clases que ya habías modelado en la Actividad 3 así como las asociaciones entre ellas que podemos inferir de la especificación de requerimientos.

Exporta el resultado final a formato pdf. En el repositorio de esta práctica deberían resultar dos ficheros: `practica2.uxf` (que contiene el fuente de *umlet*), y `practica2.pdf` (que contiene el resultado final impreso en pdf).

Más adelante determinaremos los atributos para cada una de estas clases y, utilizando estos, analizaremos la forma en que cambia el sistema con el tiempo. Y cuando ya tengamos los atributos establecidos, determinaremos las operaciones de las clases en nuestro sistema.

Práctica 4. Implementación del sistema

©2024 **Julio Vega Pérez**
Algunos derechos reservados.
Este trabajo se entrega bajo licencia CC-BY-SA 4.0.

1. Introducción

En las prácticas anteriores se han introducido los fundamentos de la orientación a objetos para, a continuación, aplicar estos al diseño orientado a objetos de un sistema ficticio de monitorización. Ahora empezaremos a implementar este diseño orientado a objetos en C++.

Previamente, hemos de repasar concienzudamente el diagrama de clases de la práctica anterior, porque la implementación ha de ser un reflejo/conversión de este en código.

2. Visibilidad

Lo primero a tener en cuenta a la hora de implementar es aplicar convenientemente los especificadores de acceso a los miembros de las clases. Ya hemos visto los especificadores `public` y `private`. Los especificadores de acceso determinan la visibilidad, o accesibilidad, de los atributos y operaciones de un objeto para otros objetos. Antes de empezar a implementar nuestro diseño, debemos considerar qué atributos y operaciones de nuestras clases deben ser `public` y cuáles deben ser `private`.

Ya hemos visto que, por lo general, los miembros de datos deben ser `private`, y que los métodos invocados por los clientes de una clase dada deben ser `public`. Sin embargo, los métodos que se llaman sólo por otros métodos de la clase como *funciones utilitarias* deben ser generalmente `private`. UML emplea marcadores de visibilidad para modelar la visibilidad de los atributos y las operaciones. La visibilidad pública se indica mediante la colocación

de un signo más (+) antes de una operación o atributo; un signo menos (-) indica visibilidad privada.

Actividad 1: Comienza por actualizar el diagrama de clases que habías confeccionado para la Práctica 3 agregando los marcadores de visibilidad.

Ejercicio: Implementación del sistema

Ahora ya sí, estamos listos para empezar a implementar el sistema. Primero convertiremos las clases del diagrama de clases en archivos de cabecera de C++. Este código representará el *esqueleto* del sistema. (Más adelante modificaremos estos archivos para incorporar el concepto orientado a objetos de la herencia.)

Sigue estas pautas para cada clase:

1. Usa el nombre que se localiza en el primer compartimento de una clase en el diagrama de clases para definir la clase en un archivo de encabezado. Recuerda usar las directivas del preprocesador `#ifndef`, `#define` y `#endif` para evitar que el archivo de encabezado se incluya más de una vez en el programa.

2. Usa los atributos que se localizan en el segundo compartimento de la clase para declarar los atributos.

3. Usa las asociaciones descritas en el diagrama de clases para declarar las referencias (o apuntadores, según sea apropiado) a otros objetos. Por ejemplo, observa en la Figura *Diagrama de clases que muestra las relaciones de composición* de la Práctica 3 que un objeto de la clase `SistemaMonitorizacion` está formado por un objeto de la clase `Pantalla`, un objeto de la clase `Teclado`, un objeto de la clase `Microfono` y un objeto de la clase `Interruptor`.

   Así, en la implementación de la clase `SistemaMonitorizacion` habrá un constructor que inicializa estos atributos con referencias a objetos actuales. Además, en esta clase, se deberán incluir (mediante `#include`) los archivos de encabezado que contienen las definiciones de las clases `Pantalla`, `Teclado`, `Microfono` e `Interruptor`, de manera que podamos declarar referencias a objetos de estas clases.

4. Mejora del punto anterior. Al incluir los archivos de encabezado para las clases `Pantalla`, `Teclado`, `Microfono` e `Interruptor`, se hace más de lo necesario. La clase `SistemaMonitorizacion` contiene referencias a objetos de estas clases, no verdaderos objetos; y

la cantidad de información requerida por el compilador para crear una referencia difiere de la que se requiere para crear un objeto. Recuerda que para crear un objeto, el programador debe proporcionar al compilador una definición de la clase que introduzca el nombre de la clase como un nuevo tipo definido por el usuario, y que indique los miembros de datos que determinen cuánta memoria se requiere para almacenar el objeto. Sin embargo, al declarar una referencia (o apuntador) a un objeto, sólo se requiere que el compilador sepa que la clase del objeto existe; esto es, no necesita conocer el tamaño del objeto.

Cualquier referencia (o apuntador), sin importar la clase del objeto al que hace referencia, sólo contiene la dirección de memoria de dicho objeto. La cantidad de memoria requerida para almacenar una dirección es una característica física del hardware de la computadora. Por ende, el compilador conoce el tamaño de cualquier referencia (o apuntador). Como resultado, es innecesario incluir el archivo de encabezado completo de una clase cuando se declara sólo una referencia a un objeto de esa clase; simplemente necesitamos introducir el nombre de la clase, pero no necesitamos proporcionar la distribución de los datos del objeto, ya que el compilador conoce de antemano el tamaño de todas las referencias.

C++ proporciona una instrucción conocida como **declaración anticipada**, la cual indica que un archivo de encabezado contiene referencias o apuntadores a una clase, pero ésta se encuentra fuera del archivo de encabezado. Así, podemos reemplazar las instrucciones `#include` mencionadas de la definición de la clase `SistemaMonitorizacion`, por declaraciones anticipadas de tales clases:

```
class Pantalla; // declaración anticipada de la clase Pantalla
class Teclado; // declaración anticipada de la clase Teclado
class Microfono; // declaración anticipada de la clase Microfono
class Interruptor; // declaración anticipada de la clase Interruptor
```

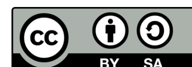
En vez de incluir (mediante `#include`) el archivo de encabezado completo para cada una de estas clases, sólo colocamos una declaración anticipada de cada clase en el archivo de encabezado para la clase `SistemaMonitorizacion`. Ten en cuenta que si la clase `SistemaMonitorizacion` contara con objetos actuales en vez de referencias, entonces tendríamos que incluir los archivos de encabezado completos.

Usar una declaración anticipada (en donde sea posible) en vez de incluir un archivo de encabezado completo nos ayuda a evitar un problema del preprocesador, conocido como inclusión circular. Este problema ocurre cuando el archivo de encabezado para

la clase A incluye el archivo de encabezado para la clase B, y viceversa. Algunos preprocesadores no pueden resolver tales directivas `#include`, lo cual produce un error de compilación. Por ejemplo, si la clase A sólo utiliza una referencia a un objeto de la clase B, entonces la instrucción `#include` en el archivo de encabezado de la clase A se puede reemplazar por una declaración anticipada de la clase B, para evitar la inclusión circular.

5. Usa las operaciones que se localizan en el tercer compartimento de la clase para escribir los prototipos de los métodos. Si todavía no se ha especificado un tipo de valor de retorno para una operación, declaramos el método con el tipo de valor de retorno `void`.

Práctica 5. Creación de la clase `Coleccion` con operadores sobrecargados



©2024 **Julio Vega Pérez**
Algunos derechos reservados.
Este trabajo se entrega bajo licencia CC-BY-SA 4.0.

1. Introducción

Las colecciones o arrays de elementos basados en punteros tienen varios problemas. Por ejemplo, un programa puede *salirse* fácilmente de cualquier extremo de un array, ya que C++ no comprueba si los subíndices están fuera del rango de este (aunque sí se puede hacer esto explícitamente). Los arrays de tamaño n deben enumerar sus elementos así: $0, ..., n-1$; no se permiten —por ejemplo— rangos de subíndices alternados. Un array no puede recibirse ni enviarse todo a la vez; se debe leer o escribir cada elemento de este de manera individual.

Dos arrays no pueden compararse significativamente con los operadores de igualdad o relacionales debido a que los nombres de los arrays son simplemente punteros a la dirección en la que empiezan estos en memoria y, desde luego, dos arrays siempre estarán en distintas ubicaciones de memoria. Cuando se pasa un array a una función de propósito especial diseñada para manejar arrays de cualquier tamaño, el tamaño del array debe pasarse como argumento adicional. Un array no puede asignarse a otro con el(los) operador(es) de asignación debido a que los nombres de los arrays son punteros `const`, y un puntero constante no se puede utilizar del lado izquierdo de un operador de asignación.

Como cabría pensar, éstas herramientas que describimos —entre otras— deberían ser, sin duda, la opción *natural* para tratar con los arrays, pero los arrays basados en punteros no proporcionan dichas herramientas. Sin embargo, C++ proporciona los medios para implementar dichas herramientas de los arrays a través del uso de las clases y la sobrecarga de

operadores.

2. Descripción

En esta práctica, vamos a crear una clase tipo array de enteros que ofrezca todas las funcionalidades que se han descrito. Esta clase realizará la comprobación de rangos para asegurar que los subíndices permanecen dentro de los límites del objeto `Coleccion`. Tamb ién debe permitir asignar un objeto `Coleccion` a otro mediante el operador de asignación. Además, los objetos de la clase `Coleccion` deben conocer su tamaño, por lo que éste no se necesitará pasar —en ningún momento— como argumento al pasar un objeto `Coleccion` a una función. Por otro lado, deberán poderse enviar o recibir objetos `Coleccion` completos mediante los operadores de flujo. Y, por último, también se podrán realizar comparaciones entre objetos `Coleccion` mediante los operadores de igualdad: $==$ y $!=$.

Para cumplir con todo lo anterior, cada objeto `Coleccion` contendrá dos atributos: `size`, que indicará el número de elementos en el objeto `Coleccion`, y un puntero `int (ptr)` que apunta a la colección de enteros. Además de esta consideración, veremos a continuación algunos otros detalles de implementación.

2.1. Sobrecarga de operadores de flujo como funciones `friend`

El operador de inserción de flujo sobrecargado y el operador de extracción de flujo sobrecargado se deben declarar como funciones `friend` de la clase `Coleccion`. Así, cuando el compilador vea una expresión del tipo `cout << objetoColeccion`, invocará a la función `operator<<` con la llamada `operator<< (cout, objetoColeccion)`. Del mismo modo, cuando el compilador vea una expresión del tipo `cin >> objetoColeccion`, este invocará a la función `operator>>` con la llamada `operator>> (cin, objetoColeccion)`.

Recuerda que estas funciones de operador de inserción de flujo y operador de extracción de flujo no pueden ser miembros de la clase `Coleccion`, ya que el objeto `Coleccion` siempre se menciona del lado derecho del operador de inserción de flujo y del operador de extracción de flujo. Si estas funciones de operador fuesen miembros de la clase `Coleccion`, tendrían que utilizarse las siguientes instrucciones para enviar y recibir un objeto Array, cuya sintaxis resultaría muy confusa para cualquier programador de C++: `objetoColeccion << cout;` y `objetoColeccion >> cin;`.

La función `operator<<` deberá imprimir el número de elementos indicados por `size` a

partir de la colección de enteros a la que apunta `ptr`. La función `operator>>` introduce los datos directamente en el array al que apunta `ptr`. Cada una de estas funciones de operador devuelve una referencia apropiada para permitir instrucciones de salida o entrada en cascada, respectivamente.

Estas funciones, al ser declaradas como funciones `friend`, tienen acceso a los datos private de un objeto `Coleccion`. Y también pueden acceder a las funciones `getSize()` y `operator[]`, aunque para esto no haría falta que fuesen funciones `friend`.

2.2. Constructor predeterminado de `Coleccion`

Una buena costumbre es implementar un constructor predeterminado. En este caso, se implementará un constructor predeterminado para la clase, en el que se especificará un tamaño predeterminado de 10 elementos. Así, cuando el compilador vea una declaración del tipo `Coleccion integers1 (7);`, este invocará al constructor predeterminado de `Coleccion`, ya que el constructor predeterminado en este caso recibe un solo argumento tipo `int`, que tiene un valor predeterminado de 10.

Por simplicidad, se ha considerado que el valor pasado por parámetro será mayor que 0, pero podría ser que no, en cuyo caso habría que implementar un mecanismo de excepciones (que ya veremos más adelante) que maneje tal situación. Para la reserva de memoria de la colección (que recordemos parte de un simple puntero) se usará `new`; una vez hecha la reserva, se asigna el puntero devuelto por esta reserva al atributo `ptr`. Después, el constructor empleará una instrucción `for` para establecer todos los elementos de la colección en 0.

Sería posible tener una clase `Coleccion` que no inicializara sus atributos si, por ejemplo, estos atributos se van a leer más adelante en algún momento; pero esto se considera, por razones obvias, una mala práctica de programación. Los objetos `Coleccion` (y los objetos en general) deben inicializarse de manera apropiada y mantenerse en un estado consistente.

2.3. Constructor de copia de `Coleccion`

Además del constructor predeterminado, se implementará también un constructor de copia. Este iinicializará un objeto `Coleccion` a partir de un objeto `Coleccion` existente. Para realizar dicha copia, hay que tener la precaución de no dejar a ambos objetos `Coleccion` apuntando a la misma memoria asignada en forma dinámica. Este problema ocurriría si no implementásemos este constructor y, en su lugar, usásemos el constructor de copia predeterminado (el de por defecto) para esta clase.

Los constructores de copia se invocan cada vez que se necesita una copia de un objeto, como cuando se pasa un objeto por valor a una función, se devuelve un objeto por valor de una función, o se inicializa un objeto con una copia de otro objeto de la misma clase. El constructor de copia, en ese caso, se llamará en una declaración cuando se instancia un objeto de la clase `Coleccion` y se inicializa con otro objeto de la clase `Coleccion`, como por ejemplo, en la instrucción de declaración `Coleccion integers3 (integers1);`.

Otros aspectos interesantes a tener en cuenta son los siguientes. Por un lado, para permitir que se copie un objeto `const`, el argumento para un constructor de copia debe ser una referencia `const` también. Por otro lado, un constructor de copia debe recibir su argumento por referencia, no por valor; en caso contrario, la llamada al constructor de copia produce recursividad infinita (un error lógico fatal), ya que para recibir un objeto por valor, el constructor de copia tiene que realizar una copia del objeto que se usa como argumento. Piensa que, cada vez que se requiere una copia de un objeto, se hace una llamada al constructor de copia de la clase; si el constructor de copia recibe su argumento por valor, ¡se llamaría a sí mismo de manera recursiva para realizar una copia de su argumento!

Y una última consideración. Si lo que hace el constructor de copia es simplemente copiar el puntero del objeto de origen al puntero del objeto de destino, entonces ambos objetos apuntarían a la misma memoria asignada en forma dinámica. De esta forma, el primer destructor en ejecutarse eliminaría la memoria asignada en forma dinámica, y el `ptr` del otro objeto quedaría indefinido. Esto probablemente produciría un grave error en tiempo de ejecución, como la terminación anticipada del programa, a la hora de utilizar este puntero.

3. Implementación

El esqueleto que debería conformar la clase `Coleccion` (`Coleccion.h`) debería ser como se muestra a continuación:

```
/* Coleccion.h
   Practical exercise 5.
   Coleccion class definition with overloaded operators.
*/

#ifndef ARRAY_H
#define ARRAY_H

#include <iostream>
```

```cpp
class Coleccion {
  friend std::ostream &operator<< (std::ostream &, const Coleccion &);
  friend std::istream &operator>> (std::istream &, Coleccion &);

public:
  explicit Coleccion (int = 10); // default constructor
  Coleccion (const Coleccion &); // copy constructor
  ~Coleccion(); // destructor

  size_t getSize() const; // return size

  const Coleccion &operator= (const Coleccion &); // assignment operator
  bool operator== (const Coleccion &) const; // equality operator

  // inequality operator; returns opposite of == operator
  bool operator!= (const Coleccion &right) const {
    return !(*this == right); // invokes Coleccion::operator==
  } // end function operator!=

  // subscript operator for non-const objects returns modifiable lvalue
  int &operator[] (int);

  // subscript operator for const objects returns rvalue
  int operator[] (int) const;

private:
  size_t size; // pointer-based array size
  int *ptr; // pointer to first element of pointer-based array
}; // end class Coleccion

#endif
```

Por su parte, el fichero fuente `Coleccion.cpp` debería contener la implementación de todas las funciones. A continuación se muestra la implementación del constructor y destructor:

```cpp
/* Coleccion.cpp
   Practical exercise 5.
   Coleccion class member -and friend- function definitions.
*/
```

```cpp
#include <iostream>
#include <iomanip> // I/O manipulation library
#include "Coleccion.h" // Coleccion class definition

using namespace std;

// default constructor for class Coleccion (default size 10)
// considering arraySize > 0, otherwise an exception should be thrown
Coleccion::Coleccion (int arraySize)
  : size (arraySize),
    ptr (new int [size]) {
  for (size_t i = 0; i < size; ++i)
    ptr [i] = 0; // set pointer-based array element
} // end Coleccion default constructor

// copy constructor for class Coleccion; must receive a reference to a Coleccion
Coleccion::Coleccion (const Coleccion &arrayToCopy)
  : size (arrayToCopy.size),
  ptr (new int [size]) {
    for (size_t i = 0; i < size; ++i)
      ptr [i] = arrayToCopy.ptr [i]; // copy into object
} // end Coleccion copy constructor

// destructor for class Coleccion
Coleccion::~Coleccion() {
  delete [] ptr; // release pointer-based array space
} // end destructor
```

Por último, para probar todos los flecos de la clase `Coleccion`, necesitamos un programa principal que haga uso de las funciones que ofrece esta. A continuación se muestra la implementación de un programa `main.cpp`. Sírvase como ejemplo de batería de pruebas del programa, al cual, por supuesto, se puede añadir toda la funcionalidad que se considere oportuna.

```cpp
/* main.cpp
   Practical exercise 5.
   Coleccion class test program.
*/
#include <iostream>
```

```
#include "Coleccion.h"

using namespace std;

int main() {
  Coleccion integers1 (7); // seven-element Coleccion
  Coleccion integers2; // 10-element Coleccion by default

  // print integers1 size and contents
  cout << "Size of Coleccion integers1 is " << integers1.getSize()
    << "\nColeccion after initialization:\n" << integers1;

  // print integers2 size and contents
  cout << "\nSize of Coleccion integers2 is "
    << integers2.getSize()
    << "\nColeccion after initialization:\n" << integers2;

  // input and print integers1 and integers2
  cout << "\nEnter 17 integers:" << endl;
  cin >> integers1 >> integers2;

  cout << "\nAfter input, the Coleccions contain:\n"
    << "integers1:\n" << integers1
    << "integers2:\n" << integers2;

  // use overloaded inequality (!=) operator
  cout << "\nEvaluating: integers1 != integers2" << endl;

  if (integers1 != integers2)
    cout << "integers1 and integers2 are not equal" << endl;

  // create Coleccion integers3 using integers1 as an
  // initializer; print size and contents
  Coleccion integers3 (integers1); // invokes copy constructor

  cout << "\nSize of Coleccion integers3 is "
    << integers3.getSize()
    << "\nColeccion after initialization:\n" << integers3;
```

```
  // use overloaded assignment (=) operator
  cout << "\nAssigning integers2 to integers1:" << endl;
  integers1 = integers2; // note target Coleccion is smaller

  cout << "integers1:\n" << integers1
    << "integers2:\n" << integers2;

  // use overloaded equality (==) operator
  cout << "\nEvaluating: integers1 == integers2" << endl;

  if (integers1 == integers2)
    cout << "integers1 and integers2 are equal" << endl;

  // use overloaded subscript operator to create rvalue
  cout << "\nintegers1[5] is " << integers1 [5];

  // use overloaded subscript operator to create lvalue
  cout << "\n\nAssigning 1000 to integers1[5]" << endl;
  integers1 [5] = 1000;
  cout << "integers1:\n" << integers1;
} // end main
```

Práctica 6. Uso de plantilla `set` de STL para almacenar (sin duplicar) a un conjunto de objetos `Usuario`

©2024 Julio Vega Pérez
Algunos derechos reservados.
Este trabajo se entrega bajo licencia CC-BY-SA 4.0.

1. Introducción

Como es lógico, en la empresa habrá más de un trabajador/usuario del sistema que estamos desarrollando. Por ello, se hace necesario crear una estructura que nos permita almacenar la información de todos ellos.

En otra práctica anterior creamos nuestro propio array de enteros, lo cual resultó en decenas de líneas de código. Y ahora conocemos que existe el concepto de *plantilla*, que es un mecanismo de ayuda que nos proporciona C++ para poder manejar estructuras de datos y algoritmos que tienen un uso elevado, independientemente de los datos que estas alberguen. De este modo, no tenemos que reinventar la rueda una y otra vez.

Ejercicio

Aprovechando la plantilla `set` de la librería STL, añade la implementación necesaria para poder almacenar la información de varios usuarios del sistema de monitorización; esto es, una colección tipo conjunto (un `set`) de objetos tipo `Usuario`.

Si lo necesitas, haz uso de la referencia web oficial de C++[1]. En ella podrás encontrar numerosos recursos: tutoriales, referencia de bibliotecas (como STL), foros de resolución de

[1] https://cplusplus.com

dudas, artículos, etc. El menú que aparece en la parte superior izquierda te será de ayuda para ir directamente a buscar lo que necesitas; en este caso, para encontrar ayuda sobre la plantilla `<set>`, puedes seguir la ruta *Reference→Containers→<set>*[2].

Genera todos los casos de usos que se te ocurran hasta asegurarte de que el conjunto funciona como es debido. Recuerda que, para implementar debidamente los casos de uso, es altamente recomendable probar los casos extremos: qué ocurre cuando la colección está vacía e intento obtener un elemento, o qué ocurre cuando inserto más allá de los límites del vector que pueda tener establecidos, o qué ocurre cuando intento obtener un elemento que no existe, y un largo etcétera.

[2] https://cplusplus.com/reference/set

Práctica 7. Excepciones, depuración y documentación

©2024 Julio Vega Pérez
Algunos derechos reservados.
Este trabajo se entrega bajo licencia CC-BY-SA 4.0.

1. Descripción

Son varios los objetivos a conseguir con esta práctica. En primer lugar, identificar situaciones críticas del sistema que estamos desarrollando y controlarlas mediante el manejo de excepciones. En segundo lugar, hacer uso de las herramientas GDB y Valgring para hacer un correcto seguimiento de la ejecución del sistema, detectar errores lógicos y fallos de segmentación. Y en tercer lugar, documentar formalmente el sistema con la herramienta Doxygen.

En esta práctica cobra especial importancia la descripción de los casos de uso. Al final de esta práctica deberías tener un sistema software bien depurado, con el manejo de las excepciones allá donde sea necesario, y analizando paso a paso la ejecución del programa. Y todo ello debe quedar reflejado mediante el registro de los casos de uso. Esto es lo que le da solidez a un desarrollo software. Recuerda siempre probar los casos extremos.

Y no menos importante es la documentación formal de un sistema software para que este adquiera un nivel profesional. Recuerda que no hace falta detallar absolutamente todo, pero sí las cabeceras de las clases y las de aquellos métodos más relevantes, así como los atributos más importantes de una clase. También es importante destacar aquellas sutilezas que consideres que incorpora tu código. Toma como ejemplo la documentación de cualquiera de las librerías software vistas en teoría para hacerte una idea del aspecto final que debería tomar la documentación de tu sistema.

Ejercicio 1. Manejo de excepciones

Recuerda todos los conceptos relativos al manejo de exceptiones que hemos visto en clase e introduce, al menos, uno de cada uno de los siguientes conceptos:

- Clase propia de excepción.
- Bloque simple `try-catch`.
- Bloque `try-catch` con varios manejadores `catch`.
- Instrucción `throw`.
- Relanzamiento de excepción.
- Excepción `bad_alloc` al usar `new`.

Ejercicio 2. Uso de GDB y Valgrind

Usando las herramientas GDB y Valgrind, y con con la ayuda de los puntos de ruptura o *breakpoints*, realiza el seguimiento de, al menos, los siguientes ítems:

- Alguna estructura de almacenamiento (e.g. un vector) que uses en tu sistema.
- Algún atributo compartido entre clases que pertenezcan a una jerarquía de herencia.
- Algún bloque de repetición (e.g. un while).
- Algún mecanismo de construcción de objeto que sea instancia de clase hija/nieta en una jerarquía de herencia.

Para documentar este seguimiento, realiza capturas de pantalla de la ventana del Terminal, mostrando el contenido de lo que vaya vertiendo el depurador durante la ejecución del sistema. Haz uso de los distintos comandos que ofrece el depurador para mostrar la información que vayas considerando de interés en cada momento.

Ejercicio 3. Documentación del sistema con Doxygen

Por último, una vez tengas bien depurado tu sistema, deberás documentarlo debidamente para que la herramienta Doxygen genere la documentación final en HTML y LaTeX.

La documentación deberá contener, al menos, lo siguiente:

- Cabeceras de las clases.
- Cabeceras de los métodos más relevantes (e.g. los métodos *getters* y *setters* no haría falta).
- Algunos atributos que consideres de interés.
- Algún bloque de código que consideres importante en el funcionamiento del sistema.

El resultado de esta documentación deberá albergarse en una carpeta denominada `doxygen-doc` en el sistema raíz del programa. Y esta contendrá, a su vez, otras dos carpetas: `html` y `latex`, que contendrán la documentación en estos dos lenguajes.

Práctica 8. Uso de ficheros

©2024 **Julio Vega Pérez**
Algunos derechos reservados.
Este trabajo se entrega bajo licencia CC-BY-SA 4.0.

1. Introducción

En esta práctica continuaremos con el desarrollo del sistema en el que venimos trabajando durante el curso. El objetivo en este caso es ofrecer la funcionalidad para que toda la información de los usuarios que hasta ahora teníamos que cargar una y otra vez en el arranque de la aplicación, se guarde permanentemente. Con esto, daremos un gran paso para que la aplicación tenga una funcionalidad muy cercana a un uso real.

Para conseguir ese objetivo, necesitamos hacer uso de ficheros (que se guardan en disco duro) que nos sirvan de soporte para almacenar toda la información. Recordemos que, sin estos, los datos cargados hasta ahora en nuestra aplicación están almacenados en memoria RAM, que es volátil; esto es, se pierden una vez cerremos la aplicación (o se interrumpa la alimentación eléctrica).

Para poder manejar estos ficheros, nos apoyaremos en las bibliotecas de gestión de gestión de entrada/salida (*input/output*) que nos ofrece C++, y que hemos visto en clase. Y, dado que vamos a leer/escribir objetos, nos apoyaremos en la modalidad de ficheros binarios (`ios::binary`).

Ejercicio

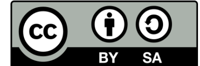
Deberás almacenar en fichero, al menos, la información de todos los usuarios. Para ello, crea un fichero `users.dat`, donde se guardará la información de estos. También puedes optar

por tener distintos ficheros para los distintos tipos de usuarios que tengas contemplados en tu sistema.

Además de los ficheros, deberás —evidentemente— implementar la funcionalidad necesaria para que toda la información que tenemos cargada en la colección (la que sea que usemos) de usuarios pueda guardarse y/o cargarse en/desde el disco duro. Recuerda, muy importante, como ya vimos en clase, la dinámica a seguir es la misma que, por ejemplo, la partida de un videojuego; esto es, primero se carga la partida y al final, antes de salir, se guarda la partida. Lo que, en nuestro contexto, se traduciría en:

1. Al inicio, **cargamos** datos del fichero a la colección de usuarios. Esto es: fichero/s $\Longrightarrow$ coleccion.
2. Ejecución de la aplicación normal.
3. Antes de salir, **guardamos** los datos de la colección al fichero, en modalidad `ios::trunc`, para conseguir tener siempre una *copia espejo* del contenido de lal colección en el fichero. Esto es: colección $\Longrightarrow$ fichero/s.

Práctica 9. Uso de patrones de diseño



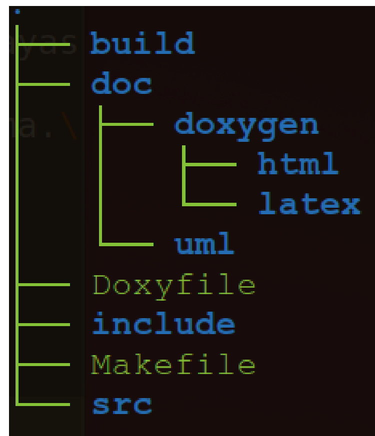
©2024 **Julio Vega Pérez**
Algunos derechos reservados.
Este trabajo se entrega bajo licencia CC-BY-SA 4.0.

1. Descripción

Esta es la última práctica del curso. Como contrapunto final del sistema en el que venimos trabajando durante el curso, el objetivo de esta práctica es implementar alguno de los patrones de diseño que hemos visto en clase. Para ello, observa el diagrama UML que tengas actualizado en la documentación de tu aplicación y extrae, al menos, una necesidad de patrón de diseño, que tendrás que implementar.

Una vez tengas claro qué patrón vas a llevar a cabo, rediseña tu diagrama UML, desarrolla en consecuencia el patrón de diseño que hayas elegido, descríbelo argumentadamente en el `README.md` de la práctica, y actualiza debidamente la documentación final del sistema.

2. Requisitos finales



En este repositorio deberás tener tu sistema al completo, con la organización reflejada en la Figura 2, con todos los detalles que has ido implementando a lo largo del curso, y que deberán ser —al menos— los siguientes:

- Archivos cabecera, en carpeta `include`; archivos fuente, en carpeta `src`; y ejecutable (`main`), en carpeta `build`.
- Uso de `Makefile` para la compilación de todos tus ficheros.
- Herencia y polimorfismo.
- Sobrecarga de operadores y/o flujo.
- Colección `set` para almacenar a los usuarios del sistema.
- Uso de ficheros para guardar permanentemente la información de los usuarios del sistema.
- Uso de algún patrón de diseño coherentemente argumentado en el `README.md`.
- Manejo de excepciones.
- Depuración con GDB.
- Documentación en carpeta `doc`, que ha de incluir, al menos, lo siguiente:
    - Diagrama UML final coherente con el sistema final desarrollado. Este ha de estar incluido en la carpeta de `uml`.
    - Documentación con Doxygen (en carpeta `doxygen`), tanto en HTML como en LaTeX(con su correspondiente archivo `refman.pdf` generado). Deberás tener el correspondiente fichero `Doxyfile` con los *features* que hayas considerado.

- Fichero `README.md` que incluya, además de lo habitual, lo siguiente:
    - Descripción final del sistema.
    - Descripción argumentada del patrón de diseño elegido e implementado.
    - Descripción de los pasos a seguir para la correcta compilación y ejecución del programa.
    - Descripción de los pasos a seguir para visualizar la documentación generada por Doxygen.

Fecha de entrega

Esta práctica se podrá entregar hasta el día antes del examen ordinario.