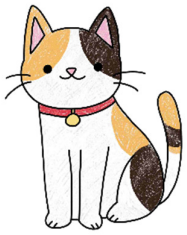
AnimalCare



La startup AnimalCare S.L. nos ha contratado para llevar a cabo el desarrollo de su plataforma de venta online de artículos para animales llamada AnimalCare, a la cual nos referiremos como "el programa".

Dicha plataforma es muy parecida en su funcionamiento a otras plataformas existentes de venta online de artículos para animales, como pueden ser, Kiwoco o Tiendanimal. Nuestra tarea será implementar dicha plataforma desde cero, usando el lenguaje de programación C++.

Para ello, se adjunta una descripción de los requisitos y características del programa, así como de las distintas clases que lo forman. Es posible añadir miembros y atributos adicionales al proyecto que faciliten la tarea o que se consideren necesario si así se desea.

La empresa nos ha pedido una entrega escalonada en varios hitos que se detallan al final de este documento. El trabajo se llevará a cabo en grupos de 2 (salvo que el total de alumnos no sea un número par, en cuyo caso habrá algún grupo de 1 alumno). AnimalCare es una empresa de proyección internacional, por lo que el idioma usado para el nombre de las clases, métodos y atributos será el inglés.

Requisitos del programa

Es imprescindible respetar la nomenclatura propuesta para clases, miembros y atributos que aparecen en este documento.

Las partes sombreadas en verde corresponden con elementos que, pese a pertenecer a una clase requerida en la primera entrega, no es necesario implementarlos hasta la segunda entrega.

Las partes sombreadas en azul corresponden con elementos que, pese a pertenecer a una clase requerida en la primera entrega, no es necesario implementarlos hasta la tercera entrega.

Descripción del funcionamiento general

El programa está compuesto por tres componentes principales: usuarios, productos y un Manager que lo gestiona todo. Todo el programa está construido alrededor del Manager, el cual nos permite realizar todas las acciones disponibles, como añadir nuevos productos a la plataforma, añadir nuevos usuarios y gestionar la información personal y los pedidos de los usuarios. Los usuarios podrán registrarse y realizar pedidos, también podrán tener varias direcciones y métodos de pago para realizar los pedidos. Además, cada producto podrá tener valoraciones de los usuarios que hayan adquirido dichos productos.

Usuarios

Cada usuario del sistema estará representado por una instancia de la clase `User`. Cada uno de ellos tendrá cierta información que debe ser pública (como el nombre de usuario) y otra que debe ser privada (como su email o contraseña), para lo cual existen dos clases que contienen esos datos: `PrivateUserData` y `PublicUserData`. Los usuarios además podrán realizar pedidos, para lo cual

tendrán una o varias direcciones de envío, así como uno o varios métodos de pago. Podrán también valorar productos que hayan adquirido y tendrán un historial de pedidos. Existe además otro tipo de usuario llamado administrador que podrá añadir nuevos productos a la plataforma.

User

- El nombre de la clase es `User`. Se crearán los archivos User.hpp y User.cpp.
- Posee un constructor paramétrico que toma como argumentos: *username*, *email* y *password*. Todos ellos son de tipo `std::string`.
- El constructor vacío debe ser protegido puesto que no vamos a querer que se pueda crear un usuario sin especificar sus datos desde fuera de la clase o subclases.
- Tiene un método público virtual llamado `isAdmin()`, que devuelve `true` en caso de que el usuario sea un administrador y `false` en caso contrario.
- El resto de los métodos y atributos de la clase se heredarán por herencia pública de `PrivateUserData` y `PublicUserData`.

PrivateUserData

- El nombre de la clase es `PrivateUserData` y representa los datos privados del usuario. Se crearán los archivos PrivateUserData.hpp y PrivateUserData.cpp.
- Consta de los siguientes atributos, todos ellos protegidos:
    - *email* y *password*, ambos de tipo `std::string`. Ambos dispondrán de sus correspondientes getters y setters.
    - *addresses*, que es un `std::vector` de punteros a `Address`. Almacena las direcciones de envío que ha registrado el usuario. Dispondrá de getter, pero no de setter.
    - *payment_options*, que es un `std::vector` de punteros a `PaymentOption`. Almacena los métodos de pago que ha registrado el usuario. Dispondrá de getter, pero no de setter.
    - *orders,* que es un `std::vector` de punteros a `Order`. Almacena el historial de los pedidos anteriores realizados por el usuario. Dispondrá de getter, pero no de setter.
- El constructor vacío debe ser protegido, puesto que no se desea crear un objeto de la clase sin especificar sus datos.
- Posee un constructor paramétrico que recibe como parámetros *email* y *password*, ambos de tipo `std::string`.

La clase `Manager` será la encargada de añadir, crear y procesar de cualquier forma posible las direcciones, métodos de pago y pedidos. Para añadir una nueva dirección tenemos el método `addAddress()`, que toma un puntero a un objeto de la clase `Address`. Para añadir una nueva forma de pago tenemos el método `addPaymentOption()`, que toma un puntero a un objeto de la clase `PaymentOption`. Finalmente, para añadir un nuevo pedido al historial tenemos el método `addOrder()`, que toma un puntero a un objeto de la clase `Order`.

PublicUserData

- El nombre de la clase es `PublicUserData`. Representa los datos públicos del usuario. Para esta clase se crearán los archivos PublicUserData.hpp y PublicUserData.cpp.
- Consta de los siguientes atributos, todos ellos protegidos:
    - *username*, de tipo `std::string`, con su correspondiente getter y setter.

    - *reputation*, de tipo entero. Cada usuario podrá hacer una reseña de los productos que haya adquirido, de tal forma que otros usuarios podrán ver y puntuar dicha reseña, acción que se realizará desde la clase `Manager`. Después, otros usuarios podrán puntuar dicha reseña, lo que aumentará o disminuirá la reputación del autor de la reseña. Este atributo tiene un getter `getReputation()`, pero en lugar de su correspondiente setter, dispone de dos métodos llamados `increaseReputation()` y `decreaseReputation()` que incrementan y decrementan en 1 el valor de la reputación. La reputación de un usuario nunca puede ser menor que cero.
- El constructor vacío debe ser protegido, puesto que no se desea crear un objeto de la clase sin especificar sus datos.
- Posee un constructor paramétrico que recibe como parámetro *username*, de tipo `std::string`. El valor por defecto de la reputación es 0.

Admin

Un administrador es un tipo de usuario, por tanto, hereda los datos públicos y privados de `User`. Lo que caracteriza a los administradores es que son a la vez usuarios y trabajadores de la empresa, por lo que son los únicos que pueden añadir productos a la plataforma para que estén accesibles y a la venta para el resto de los usuarios. Un administrador también puede ser cliente, con la diferencia de que estos **obtienen un descuento del 7.5% sobre el total en cada pedido que realizan en la plataforma.**

La clase tiene un nuevo atributo *worker_id*, de tipo `unsigned long`, **único para cada empleado**. El constructor vacío de la clase debe ser privado, puesto que no queremos crear ningún administrador sin especificar sus datos iniciales. El constructor paramétrico debe tener los siguientes parámetros: un nombre de usuario (`std::string`), un email (`std::string`), una contraseña (`std::string`) y un código de empleado de tipo `unsigned long`. Se crearán los ficheros Admin.hpp/.cpp.

Address

- El nombre de la clase es `Address` y representa una dirección de envío registrada por el usuario. Se crearán los ficheros Address.hpp/.cpp.
- Posee los siguientes atributos, todos ellos privados:
    - *id*, de tipo entero. Su valor depende del número de direcciones que haya creado el usuario al que pertenece. Es decir, si la dirección que estamos creando es la primera para un determinado usuario, id valdrá 0, si es la segunda valdrá 1, etc.
    - *address*, de tipo `std::string`
    - *city*, de tipo `std::string`
    - *province*, de tipo `std::string`
    - *postal_code*, de tipo `unsigned int`
- Todos los atributos cuentan con sus correspondientes getters y setters.
- El constructor vacío debe ser privado, puesto que no se desea crear un objeto de la clase sin especificar sus datos.
- Posee un constructor paramétrico que recibe como parámetros *id*, *address*, *city*, *province* y *postal*_code.
- Posee además un método llamado `show()` cuya función es la de devolver en forma de texto (`std::string`) el contenido de la dirección. El formato del `string` de salida es el siguiente:

{address}, {city}, {province}, {postal_code}

Donde los elementos entre corchetes representan variables de la clase (los corchetes no se imprimirán).

PaymentOption

La aplicación permite dos tipos de métodos de pago, por tarjeta y por Bizum. Esta clase es una clase abstracta que contiene el método virtual puro `show()`.

- El nombre de la clase es `PaymentOption` y representa un método de pago registrado por el usuario para realizar los pedidos. Se crearán los ficheros PaymentOption.hpp/.cpp.
- Posee los siguientes atributos protegidos, con sus correspondientes getters y setters:
  - *id*, de tipo entero. Su valor depende del número de métodos de pago que haya creado el usuario al que está asociado. Es decir, si es el primer método de pago para un determinado usuario, id valdrá 0, si es el segundo valdrá 1, etc.
  - *billing_address*, de tipo `Address*`. Representa la dirección de facturación del método de pago y será un puntero a una de las direcciones previamente registradas por el usuario.
- El constructor vacío debe ser protegido, puesto que no se desea crear un objeto de la clase sin especificar sus datos desde fuera de la cadena de herencia.
- Posee un constructor paramétrico que recibe como parámetros *id*, de tipo entero, y *billing_address*, de tipo Address*.
- Cuenta con el método virtual puro `show()`, cuya función es la de devolver en forma de texto (`std::string`) el contenido de un método de pago. El formato varía en función del método de pago, el cual se explica más adelante en cada uno de ellos.

CreditCard

Es un tipo de método de pago, a través de una tarjeta bancaria.

- El nombre de la clase es `CreditCard` y representa una tarjeta bancaria asociada al usuario. Se crearán los ficheros CreditCard.hpp/.cpp.
- Posee los siguientes atributos privados, junto con sus setters y getters:
  - *number*, de tipo `unsigned long`, y que representa el número de la tarjeta de crédito.
  - *cardholder*, de tipo `std::string`, que representa el titular de la tarjeta.
- El constructor vacío debe ser privado, puesto que no se desea crear un objeto de la clase sin especificar sus datos.
- Posee un constructor paramétrico que recibe como parámetros *id* (`int`), *billing_address* (`Address*`), *number* (`unsigned long`) y *cardholder* (`std::string`).
- El método `show()` devuelve un `std::string` formateado con la información de la tarjeta. El formato de salida esperado para este método es el siguiente:

```
\tid {id} - Credit Card:
\tBilling address: {billing_address}
\t{number} - {cardholder}
```

Donde los elementos entre corchetes representan variables de la clase (los corchetes no se imprimirán). La secuencia de escape '\t' corresponde a un tabulador. Para la variable *billing_address* lo que se ha de imprimir es el resultado de su método `show()`, es decir, la dirección en forma de texto.

Bizum

Es otro tipo de método de pago, esta vez a través de una cuenta de Bizum.

- El nombre de la clase es `Bizum` y representa una cuenta de Bizum asociada al usuario. Se crearán los ficheros Bizum.hpp/.cpp.
- Posee un único atributo *number*, de tipo `unsigned int`, que será el número de teléfono asociado a la cuenta de Bizum. Será privado y tendrá su correspondiente setter y getter.
- El constructor vacío debe ser privado, puesto que no se desea crear un objeto de la clase sin especificar sus datos.
- Posee un constructor paramétrico que recibe como parámetros *id* (`int`), *billing_address* (`Address*`) y *number* (`unsigned int`).
- El método `show()` devuelve un `std::string` formateado con la información de la cuenta. El formato de salida esperado es el siguiente:

```
\tid {id} - Bizum Account:
\tBilling address: {billing_address}
\t{number}
```

Donde los elementos entre corchetes representan variables de la clase (los corchetes no se imprimirán). La secuencia de escape '\t' corresponde a un tabulador. Para la variable *billing_address* lo que se ha de imprimir es el resultado de su método `show()`, es decir, la dirección en forma de texto.

Order

Esta clase representa un pedido realizado por un usuario. Para esta clase se crearán los archivos Order.hpp y Order.cpp. Para lograr su funcionalidad cuenta con los siguientes atributos privados:

- *reference*, de tipo `unsigned long`. Es un identificador único para cada pedido y debe ser único entre todos los pedidos de todos los usuarios de todo el programa.
- *products*, que es un `std::vector` de `unsigned long`, es decir, que almacena las referencias a los productos del pedido. Tiene su correspondiente getter, pero no setter. En su lugar, la clase dispone del método `addProduct()`, que recibe como parámetro la referencia del nuevo producto del pedido, de tipo `unsigned long,` y que la añade al vector.
- *date*, de tipo `std::time_t`. Para ello vamos a utilizar la función `std::time()` de la librería `<ctime>`. Pasando como único argumento un 0 a la función obtenemos una variable del tipo `std::time_t` que contiene la fecha actual. Concretamente lo que se obtiene es el llamado Tiempo Unix o Tiempo Posix, que se define como la cantidad de segundos transcurridos desde la medianoche UTC del 1 de enero de 1970. Esta manera de representar las fechas es muy utilizada en sistemas tipo Unix.
- *delivery_address*, de tipo entero, que representa la dirección de envío del pedido. Es una de las direcciones registradas por el usuario que ha realizado el pedido, en concreto, el ID de la dirección.

- *payment_option*, de tipo entero, que representa el método de pago elegido para realizar el pedido. Es uno de los métodos de pago registrados por el usuario que ha realizado el pedido, en concreto, el ID de dicho método de pago.
- *total*, de tipo float, que representa el coste total del pedido. Será la suma del precio de todos los productos que forman el pedido, excepto para los administradores, que tienen un descuento.

Todos los atributos tienen su correspondiente getter y setter, a menos que se indique lo contrario. El constructor vacío debe ser privado, puesto que no se desea crear un objeto de la clase sin especificar sus datos. La clase posee dos constructores paramétricos:

- El primero recibe como parámetros la referencia del pedido (`unsigned long`), los productos del pedido (`std::vector<unsigned long>`), la dirección de envío (`int`), el método de pago (`int`) y el importe total del pedido (`float`).
- El segundo recibe como parámetros la referencia del pedido (`unsigned long`), la dirección de envío (`int`) y el método de pago (`int`).

En todos los constructores el atributo *date* se rellena dentro del propio constructor, no es un parámetro, para así asegurar que la fecha del pedido corresponde con el momento exacto de su creación.

Product

Esta clase representa un producto de los que se venden en la plataforma en desarrollo. Se crearán los ficheros Product.hpp/.cpp. Para lograr su funcionalidad esta clase cuenta con los siguientes atributos, todos ellos privados, teniendo los 4 primeros sus correspondientes setters y getters:

- *name*, de tipo `std::string`, que representa el nombre o descripción corta del producto.
- *description*, de tipo `std::string`, que representa la descripción larga del producto.
- *reference*, de tipo `unsigned long`, que representa una referencia única para cada producto.
- *price*, de tipo `float`, que representa el precio del producto.
- *reviews*, que es un `std::vector` de punteros a `Review`, es decir, contiene todas las reseñas hechas por los usuarios para este producto en concreto. Este último atributo posee su correspondiente getter `getReviews()`, pero no tiene setter. En su lugar la clase dispone de un método `addReview()` que acepta como único parámetro un puntero a Review, con la reseña que ha de ser añadida al producto.

De nuevo, el constructor vacío deberá ser privado, pues no es deseable poder crear un producto sin datos. El constructor parametrizado deberá tener como parámetros un nombre (`std::string`), una descripción (`std::string`), una referencia (`unsigned long`) y un precio (`float`).

La clase producto además tendrá implementada la sobrecarga del operador `<<` con el fin de poder imprimir por pantalla fácilmente un producto junto con sus reseñas utilizando simplemente `cout`. El formato de salida por pantalla deberá ser el siguiente:

```
{reference} - {name}
{description}
{price}
\t-- User reviews --
\t{review}
\t----
\t{review}
\t----
```

Donde los elementos entre corchetes representan variables de la clase (los corchetes no se imprimirán). La secuencia de escape '\t' corresponde a un tabulador. El formato para imprimir una `Review` se detalla a continuación. Si el producto no tiene ninguna reseña, solo se imprimirá por pantalla hasta *price*.

Review

Esta clase representa las reseñas dejadas por los usuarios para un determinado producto. Se crearán los ficheros Review.hpp/.cpp. Cuenta con los siguientes atributos privados:

- *id*, de tipo `unsigned long`, representa un ID único para cada reseña en toda la plataforma. Tiene getter, pero no setter.
- *date*, de tipo `std::time_t`, representa la fecha en la que se creó la reseña. Para más detalles del tipo `std::time_t`, consultar el atributo *date* de la clase `Order`. Tiene getter y setter.
- *rating*, de tipo entero, será una calificación de 0 a 5 estrellas. Tiene sus correspondientes setters y getters.
- *text*, de tipo `std::string`, el texto de la reseña. Posee sus correspondientes setters y getters.
- *votes*, de tipo entero, representa los votos de otros usuarios a la reseña. Tiene getter, pero no posee setter. Un su lugar cuenta con el método `incrementVotes()`, que incrementa en uno los votos y `decrementVotes()` que los decrementa en uno.
- *author*, es un puntero a `PublicUserData` y representa la información pública del usuario que ha creado la reseña. Tiene getter, pero no setter.

Esta clase tiene además un constructor vacío privado para evitar que se puedan crear objetos sin los datos iniciales y un constructor parametrizado que tomará como parámetros un ID (`unsigned long`), una puntuación (`int`), el texto de la reseña (`std::string`) y el autor que la creó (un puntero a `PublicUserData`). El atributo *date* se rellena dentro de los constructores y el valor por defecto de los votos es 0.

Finalmente, la clase `Review` cuenta con un método `show()` que devuelve un `std::string` con la reseña en forma de texto formateada con el siguiente formato:

```
\t{rating} starts on {date} by {username}
\t{text}
\t{votes} votes
```

Donde los elementos entre corchetes representan variables de la clase (los corchetes no se imprimirán). La secuencia de escape '\t' corresponde a un tabulador.

Manager

La clase `Manager` es la encargada del funcionamiento de la aplicación y de proporcionar toda la funcionalidad de esta.

Cuenta con los siguientes atributos privados:

- *users*, un `std::vector` de punteros a User, el cual almacena los usuarios actualmente registrados en el sistema. Se dispone del getter `getUsers()`.
- *products*, un `std::vector` de punteros a `Product`, que contiene todos los productos que se venden en la plataforma. Se dispone del getter `getProducts()`.
- *current_user*, un entero que indica que usuario es el que se encuentra actualmente logueado en el sistema. El valor de este atributo es la posición en el vector *users* del usuario. Cuando no hay ningún usuario logueado, debe tener un valor de -1.

Cuenta con los siguientes métodos generales:

- Un constructor vacío que inicializa los atributos de la clase a valores por defecto.
- Un destructor encargado de eliminar todos los objetos creados dinámicamente por el `Manager`.

Además, cuenta con los siguientes métodos básicos:

- **`login()`**. Para poder realizar cualquier tipo de acción sobre la plataforma, un usuario debe estar registrado y logueado. Para ello, esta función toma como parámetros el *email* y *password*, ambos de tipo `std::string` y los valida. Un usuario no se puede loguear con unas credenciales incorrectas ni tampoco se puede loguear si ya está logueado algún otro usuario. Devuelve `true` en caso de éxito, `false` en caso contrario.
- **`logout()`**. Este método se utiliza para cerrar la sesión del usuario actualmente logueado. No se puede cerrar sesión si no hay nadie logueado. Devuelve `true` en caso de éxito y `false` en caso contrario.
- **`isLogged()`**. Con esta función podremos conocer si hay algún usuario logueado en el sistema. Devuelve `true` en el caso de que haya algún usuario logueado y `false` en caso contrario.
- **`getCurrentUser()`**. Con este método se obtiene toda la información (pública y privada) del usuario actualmente logueado. Devuelve un puntero a `User` si hay un usuario logueado y `nullptr` en caso contrario.
- **`addUser()`**. Este método permite crear un nuevo usuario. Dentro del método se realizarán las comprobaciones pertinentes antes de crear un nuevo usuario, como por ejemplo, que no puedan existir dos usuarios con el mismo nombre de usuario o email. Toma como parámetros *username*, *email* y *password*. Todos ellos de tipo `std::string`. Devuelve `true` en caso de éxito y `false` en caso contrario.
- **`addAdmin ()`**. Idéntico al método anterior, con la diferencia de que se crea un usuario de tipo `Admin`, por lo que además toma como parámetro el ID de empleado.
- **`eraseCurrentUser().`** Con este método se puede dar de baja (eliminar) al miembro logueado actualmente. Devuelve `true` en caso de éxito y `false` en caso contrario.

- Desde el `Manager` se podrá editar la información básica del usuario actualmente logueado. Para ello se dispone de los métodos **`editUsername()`**, **`editEmail()`** y **`editPassword()`**. Todos ellos toman como único parámetro un `std::string` con la nueva información. El nuevo nombre de usuario y/o email deben ser únicos. Todos ellos devuelven `true` en caso de éxito y `false` en caso contrario.
- Además, el `Manager` también permite añadir una nueva dirección de envío o un nuevo método de pago para el usuario actualmente logueado. Para ello existen los métodos **`addAddress()`**, **`addCreditcard()`** y **`addBizum()`**. El primero toma como parámetros una dirección (`std::string`), una ciudad (`std::string`), una provincia (`std::string`) y un código postal (`unsigned int`). El método `addCreditCard()` toma como parámetros una dirección de facturación (`Address*`), un número de tarjeta (`unsigned long`) y un titular (`std::string`). El método `addBizum()` recibe como parámetros una dirección de facturación (`Address*`) y un número de teléfono (`unsigned int`). Los tres métodos devuelven `true` en caso de éxito y `false` en caso contrario.

De cara a la segunda entrega, será necesario implementar los siguientes métodos:

- **`addProduct()`**. Este método añade un nuevo producto al `Manager`. Solo los usuarios logueados que sean administradores pueden hacerlo. Recibe como parámetros *name*, de tipo `std::string`, *description*, de tipo `std::string`, *reference*, de tipo `unsigned long` y *price* de tipo `float`. No puede haber dos productos con la misma referencia. Devuelve `true` en caso de éxito y `false` en caso contrario.

- **`showUsers()`**. Con este método obtenemos todos los usuarios registrados. En concreto, este método devuelve un vector de punteros a `PublicUserData`, exponiendo solo la parte pública de los usuarios. Este método solo puede ser llamado por un administrador, de tal forma que, si un usuario normal intentase acceder a él, este devolvería un vector vacío.
- **`makeOrder()`**. Este método recibe como parámetros *products*, de tipo `std::vector` de `unsigned long`, *delivery_address*, de tipo entero, y *payment_option*, de tipo entero. El método calculará el importe total del pedido en función de los productos añadidos y del posible descuento a los administradores y le asignará una referencia única. Con estos parámetros se crea un nuevo pedido y se añade al historial de pedidos del usuario actualmente logueado. Este método devuelve `true` en caso de éxito y `false` en caso contrario.
- **`createReview()`**. Este método permite al usuario actualmente logueado hacer una reseña de un producto. Solo se permite hacer reseñas de productos adquiridos por el usuario logueado. Si el usuario nunca ha adquirido el producto, no podrá reseñarlo. El método toma como parámetros la referencia del producto a reseñar (`unsigned long`), *rating*, de tipo `int` y *text* de tipo `std::string`. Con ellos esta función crea la reseña y la añade al resto de reseñas del producto reseñado, creando un id único para ella. Este método devuelve `true` en caso de éxito y `false` en caso contrario.
- **`getReviewsByRating()`**. Este método nos permite filtrar las reseñas de un producto por puntuación. Toma como parámetros la referencia del producto del cual se quieren obtener las reseñas, de tipo `unsigned long`, y un entero con la puntuación por la que filtrar. Devuelve un `std::vector` con todas aquellas reseñas (`Review*`) que tengan la puntuación solicitada.
- Los usuarios pueden votar las reseñas creadas por otros usuarios, pero no pueden votar aquellas que hayan creado ellos mismos. Para ello la clase `Manager` cuenta con los métodos **`upvoteReview()`** y **`downvoteReview()`**. Estos métodos reciben como parámetro el *id* de la

`Review`, de tipo `unsigned long`, e incrementan o decrementan en una unidad los votos de esa reseña. Además, incrementan o decrementan en una unidad la reputación del usuario que ha creado la reseña. Cada usuario solo puede votar una vez cada reseña. Ambos métodos devuelven `true` en caso de éxito y `false` en caso contrario.

- El usuario que ha creado una reseña, y que se encuentra actualmente logueado, puede modificar dicha reseña. Para ello tenemos los métodos **`modifyReviewRating()`** y **`modifyReviewText()`**. El primero de ellos toma como parámetros el *id* de la review, de tipo `unsigned long`, y la nueva puntuación de 0 a 5 estrellas, de tipo `int`. El segundo método toma como parámetros el *id* de la review, de tipo `unsigned long`, y un `std::string` con el nuevo texto de la `review`. Ambos métodos devuelven `true` en caso de éxito y `false` en caso contrario.
- Finalmente, el usuario actualmente logueado puede eliminar una reseña propia utilizando el método **`deleteReview()`** que toma como parámetro el *id* (`unsigned long`) de la reseña a eliminar. Un usuario normal no puede eliminar una reseña que no haya creado a menos que sea un administrador. Estos usuarios si pueden eliminar reseñas que, bajo su criterio, no sean adecuadas. Este método devuelve `true` en caso de éxito y `false` en caso contrario.

Además de las funciones mencionadas anteriormente, se deberá poder cargar y guardar el estado del sistema a través de los siguientes métodos, usando ficheros (requisito de la tercera entrega):

- **`saveToFile()`,** que toma como argumento la ruta del archivo a guardar en forma de std::string y vuelca toda la información del sistema al fichero: usuarios, con toda la información que contienen y los productos junto con sus reseñas.
- **`loadFromFile()`**, que toma como argumento la ruta del archivo a cargar en forma de std::string y restaura toda la información guardada.

El formato de los ficheros se detalla al final de este documento.

Interfaz del programa

Para poder interactuar con los usuarios, el programa debe disponer de una interfaz que muestre los datos que se manejan desde el Manager de la aplicación y que permita introducir nuevos datos a la aplicación (datos de nuevos usuarios, productos, etc.). Esta interfaz se llevará a cabo a través de la terminal usando las funciones de entrada y salida estándar de C++.

La interfaz deberá permitir llevar a cabo todas las acciones posibles que permite el Manager, tales como iniciar sesión, crear nuevos usuarios, crear productos, hacer pedidos, etc. Además, deberá controlar que acciones puede realizar cada usuario del sistema. La primera vez que se ejecute la aplicación será necesario que se pida por terminal la creación de un administrador, puesto que es necesario para crear nuevos productos, necesarios para poder utilizar el programa.

El formato de la interfaz queda a criterio del alumno, valorándose su claridad y cómo de intuitiva es la misma. La interfaz será probada a mano durante la entrega final. Teniendo en cuenta la cantidad de acciones que puede llevar a cabo el Manager, es muy conveniente **NO implementar la interfaz íntegramente dentro de la función main()**. Es altamente recomendable implementar la interfaz en una clase a parte o que al menos todas las funciones que contenga estén definidas e implementadas en un archivo .hpp/.cpp separado del resto del código.

Posibles mejoras

Además de la funcionalidad básica (necesaria para aprobar), la interfaz en terminal y el soporte para ficheros, el alumno puede implementar alguna de las siguientes mejoras (u otras propuestas por el alumno), que serán probadas a mano durante la entrega final.

- Añadir la sobrecarga del operador << para poder imprimir por pantalla más clases con `cout` (para la clase `Product` ya es obligatorio).
- Mostrar todas las valoraciones realizadas por un determinado usuario ordenadas cronológicamente.
- Añadir sobrecargas de operadores adicionales.
- Comprobar el formato de los datos utilizados por el usuario. Por ejemplo, una tarjeta de crédito debe tener 16 dígitos, el código postal 5, etc.
- Cuando se impriman por pantalla fechas, darles formato, en lugar de imprimir directamente el tipo time_t, Ej.: "16 Mar 2021 18:09:10", en lugar de "1615918150", el equivalente en tiempo Unix.
- Añadir una clase que haga las funciones de "carrito de la compra", donde se almacenen los productos que un usuario va añadiendo durante el proceso de compra.
- Utilizar excepciones.

Entregas

El trabajo se realizará en tres entregas. A continuación, se detallan los requisitos de cada entrega:

Primera entrega

Estará compuesta por:

- Diagrama UML del sistema completo (diagrama de clases).
- Definición de todas las clases que forman el sistema, con sus atributos y métodos. Solo será necesario implementar aquellos métodos que no estén subrayados de ningún color.

Fecha de entrega: 24/10/2021

Segunda entrega

Esta segunda entrega consiste en:

- Todo lo anterior.
- Implementar el código de aquellos métodos que se encuentren subrayados en verde.

Fecha de entrega: 20/11/2021

Tercera entrega

La tercera y última entrega estará compuesta por la funcionalidad básica del programa (entregas 1 y 2), mínimo necesario para poder aprobar el trabajo. **Si el programa funciona correctamente hasta este punto, la nota del trabajo será como máximo de un 5, condicionada a la revisión del código por parte del profesor encargado y a las cuestiones realizadas a cada miembro del grupo en la defensa final.**

Para obtener el resto de los puntos, se deberán entregar las siguientes funcionalidades:

- Soporte para guardar y cargar los datos de la aplicación desde ficheros de datos (implementación de los métodos resaltados en azul), **2 puntos**.
- Interfaz del programa, **3 puntos**.
- Mejoras extra que cada grupo considere oportunas o quiera realizar al programa, (Solo mejoras consideradas útiles) máximo de **2 puntos extra** en total.

Fecha de entrega: 15/12/2021

ANEXO: Formato de los archivos de salida

En primer lugar, se volcarán los `Users`, y posteriormente, los `Products`. El formato de cada `User` es el siguiente:

- La string "**User:**" al comienzo del bloque de cada usuario
- *username*
- *email*
- *password*
- *reputation*
- -1 si el usuario es "normal", *worker_id* si es administrador
- Para cada dirección de envío, la string "**Address:**"
- *id*
- *address*
- *city*
- *province*
- *postal_code*
- Para cada método de pago, la string "**CreditCard:**" o "**Bizum:**"
- *id*
- *billing_address*, en concreto el *id* asociado
- Si el método de pago es CreditCard:
    - *number*
    - *cardholder*
- Si el método de pago es Bizum:
    - *number*
- Para cada pedido realizado por el usuario, la string "**Order:**"
    - *reference* (la referencia del pedido)
    - Para cada producto:
        - La string "**order_product:**"
        - *reference* (la de cada producto del pedido)
    - *date*
    - *delivery_address*
    - *payment_option*
    - *total*

Ejemplo de salida de `User`:

```
User:
Carlos
carlos@gmail.com
1234
0
2569847
Address:
0
Calle Falsa, 123
Leganes
Madrid
```

28926
Address:
1
Calle Falsa, 5, 2º A
Getafe
Madrid
28963
Bizum:
0
1
900856060
Order:
0
Order Product:
0
12133266461
1
0
560.10

El formato de cada `Product` es el siguiente:

- La string "**Product:**"
- name
- description
- reference
- price
- Para cada reseña, la string "**Review:**"
    - id
    - date
    - rating
    - text
    - votes
    - username

Ejemplo de salida `Product:`

Product:
affordable cat food 12 kg
Inexpensive cat food low in essential nutrients necessary for proper health and growth.
12133266461
9.99
Review:
0
12133266360
0
Muy mala calidad. Tiene la apariencia y el sabor del cartón mojado.
15394
Carlos