Programación Orientada a Objetos

Práctica 2: Relaciones de asociación, y contenedores de la STL
Implementación del caso de uso 1 «Gestión de usuarios y tarjetas» y
del caso de uso 2 «Gestión del carrito de la compra» del S. G. L.

Curso 2018–19

Índice

1. Introducción

En esta práctica se va a llevar a cabo la implementación y prueba del caso de uso 1 «Gestión de usuarios y tarjetas» y del caso de uso 2 «Gestión del carrito de la compra» del Sistema de Gestión de Librería (SGL) descrito de manera general en la sección de prácticas del campus virtual de la asignatura.

Se partirá de los requisitos correspondientes a estos casos de uso y a partir de su diagrama de clases se realizará su implementación en el lenguaje de programación C++. Después se deberán realizar una serie de pruebas para asegurarse de que la aplicación, para los casos de uso 1 y 2, funciona correctamente y satisface los requisitos especificados.

2. Objetivos

Implementar en C++ el modelo de clases para los casos de uso 1 y 2 del SGL cumpliendo los principios de la programación orientada a objetos y todas las pruebas necesarias para que estemos seguros en un alto grado de que la implementación tiene el comportamiento esperado.

3. Descripción de los requisitos de los casos de uso

Comenzaremos la implementación del SGL por la gestión de los usuarios que van a realizar las compras y de las tarjetas de crédito con las que realizarán sus pagos. El SGL tendrá, por tanto, que permitir la creación de nuevas cuentas de usuario. De cada usuario se deben guardar nombre y apellidos, así como dirección postal. El usuario elegirá un identificador o nombre que debe ser único, y una contraseña de al menos 5 caracteres y que habrá que almacenar cifrada. Se deberá guardar información de una o más tarjetas de pago que el usuario podrá elegir para pagar los diferentes pedidos que haga. De las tarjetas se guardará el tipo, número, titular, si está activa, y la fecha de expiración.

Cada cuenta de usuario va a disponer de un «carrito de la compra virtual», en el que puede ir añadiendo los artículos que desee comprar y quitando aquellos en los que ya no esté interesado. Puede adquirir varias unidades de un mismo artículo, especificando la cantidad al introducirlo en el carrito. No se comprobará el número de unidades almacenadas, pudiéndose añadir todos los ejemplares que se quiera. Los artículos tienen un número de referencia, título, fecha de publicación, precio de venta y cantidad disponible.

4. Diagrama de clases de los casos de uso 1 y 2

En la figura 1 aparece el diagrama de clases asociado con los casos de uso 1 y 2. Se trata de tres clases principales —*Usuario*, *Tarjeta* y *Artículo*— y otras auxiliares —*Clave* y *Número*, *Cadena* y *Fecha*—, que se relacionan con las primeras por composición, es decir, las principales

contienen varios atributos que son objetos de las clases auxiliares. Existe una relación de asociación calificada entre las clases *Usuario* y *Tarjeta* y una asociación entre *Usuario* y *Articulo* con un atributo de enlace, *cantidad*.

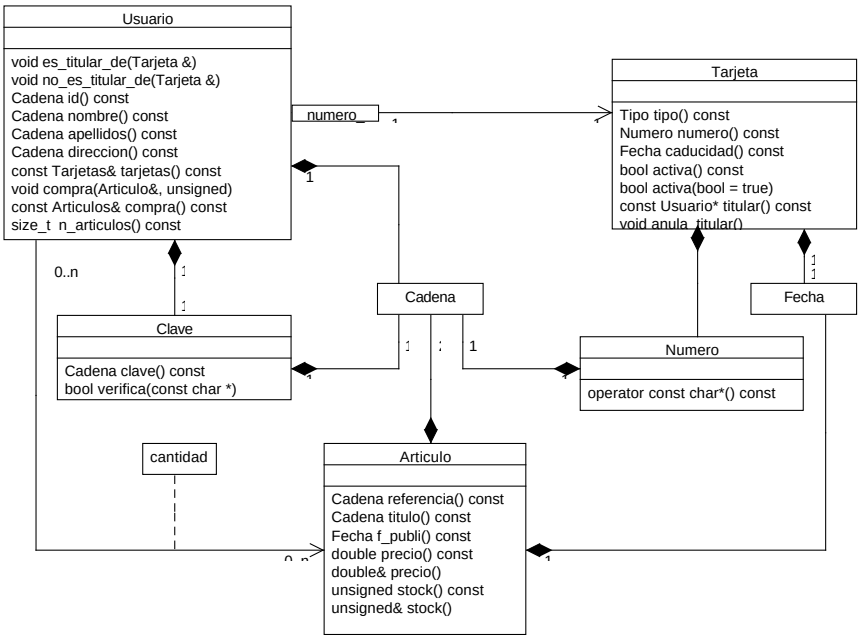

Figura 1: Diagrama de clases de los casos de uso 1 y 2

Nota importante: Aunque en circunstancias normales o «reales» se debería, no se podrá usar la clase estándar *string*, sino *Cadena* en su lugar, que por algo debe estar hecha antes de comenzar esta práctica.

5. Implementación de las clases

5.1. La clase Articulo

- Tiene cinco atributos, que corresponden al código de referencia, título, fecha de publicación, precio y número de ejemplares a la venta. Por ahora no se establecen restricciones de formato sobre el código de referencia, se admitirá cualquier *Cadena*.
- Un artículo se construye únicamente con cinco parámetros dados en el orden: referencia, título, fecha de publicación, precio y existencias.
- Los atributos serán devueltos por métodos observadores cuyos nombres son *referencia*, *titulo*, *f_publi*, *precio* y *stock*. Los dos últimos estarán sobrecargados para devolver una referencia al atributo correspondiente, con el fin de permitir su modificación.

- El operador de inserción en un flujo de salida << imprimirá referencia, título, año de publicación y precio con el formato que se muestra en el ejemplo:
`[110] "Fundamentos de C++", 1998. 29,95 €`

5.2. La clase Clave

- Su atributo es una *Cadena* que aloja la contraseña cifrada.
- Se construye a partir de una cadena de caracteres de bajo nivel que contendrá la contraseña «en claro», sin cifrar. La contraseña puede ser incorrecta por ser demasiado corta (menos de 5 caracteres, como se ha dicho), o por haber ocurrido algún error al cifrarse. En esos casos, el constructor elevará la excepción *Clave::Incorrecta*. Esta clase almacenará como atributo la razón de la incorrección, que será una enumeración llamada *Clave::Razon* con las constantes *CORTA* y *ERROR_CRYPT*. Su método observador *razon* devolverá dicho atributo.
- El método observador *clave* devolverá la contraseña cifrada.
- El método *verifica* recibe como parámetro una cadena de bajo nivel con una supuesta contraseña *en claro* y devuelve el valor booleano verdadero si se corresponde con la contraseña almacenada, o falso si no. Vea la sección 5.7 para saber cómo cifrar y descifrar la contraseña.

5.3. La clase Usuario

- Sus atributos son los siguientes:
  - Cuatro *Cadena* que representarán, por este orden: identificador, nombre, apellidos y dirección del usuario.
  - La contraseña, de tipo *Clave* (V. § 5.2).
  - Las tarjetas de pago que posea: un diccionario ordenado de números de tarjeta y punteros a tarjetas: (`std::map<Numero, Tarjeta*>`. Vea § 5.5). Por comodidad, se definirá un sinónimo público para este tipo, llámese *Tarjetas*.
  - El contenido del carrito, representado por un diccionario desordenado de punteros a artículos y el atributo de enlace de la asociación con la clase *Articulo*, que es la cantidad: (`std::unordered_map<Articulo*, unsigned int>`). Igualmente, por comodidad, se definirá un sinónimo público para este tipo, y se llamará *Articulos*.
- Un *Usuario* se construirá únicamente con 5 parámetros, que serán por este orden: identificador, nombre, apellidos, dirección y clave.
- El constructor debe comprobar que el usuario que se va a construir sea correcto; es decir, que el identificador de usuario no esté repetido (V. § 5.7).

En caso de que el identificador ya exista, el constructor elevará una excepción del tipo *Usuario::Id_duplicado*, cuyo constructor recibirá una *Cadena* representando ese identificador duplicado; su método observador *idd* devolverá dicho identificador, guardado como atributo en esta clase de excepción.

- Un *Usuario* no podrá crearse por copia de otro, no podrá pues copiarse ni asignarse a otro; esto es lógico, pues entonces se crearía un *Usuario* con el mismo identificador, lo cual está prohibido, según se ha dicho ya en el punto anterior.
- Un *Usuario* poseerá métodos observadores que devolverán los atributos. Estos métodos se llamarán *id*, *nombre*, *apellidos*, *direccion* y *tarjetas*.
- De la asociación con la clase *Tarjeta* se encargarán los métodos de nombre *es_titular_de* y *no_es_titular_de*, que recibirán como parámetro una *Tarjeta* con la que el *Usuario* se asociará o de la que se desligará.
- El destructor tendrá que desligar sus tarjetas, llamando al método *Tarjeta::anula_titular* sobre cada una de ellas. Solamente la clase *Usuario* podrá llamar a este método de *Tarjeta*. Vea § 5.5.
- La asociación unidireccional con *Articulo* se establecerá con un método llamado *compra* que recibirá dos parámetros, artículo y cantidad (por omisión es 1). Si la cantidad es 0, el artículo se sacará del carrito; es decir, el enlace con el artículo será eliminado; si es mayor que 0, esta será la nueva cantidad de dicho artículo en el carrito. La cantidad no está limitada, ya que no se requiere comprobar el *stock* del artículo.
- Un *Usuario* deberá proporcionar otros dos métodos observadores que devuelvan la colección de artículos y cantidades que contiene el carrito y el número de artículos diferentes que hay en él. Estos métodos se llamarán *compra* (método sobrecargado) y *n_articulos*, respectivamente.
- Se sobrecargará el operador de inserción en flujo (<<) para mostrar o imprimir un *Usuario* en un flujo de salida. El formato será:

  identificador [*clave cifrada*] *nombre apellidos*
  dirección
  `Tarjetas:`
  <lista de tarjetas>

- Por último, se definirá una función externa *mostrar_carro*, que deberá mostrar o imprimir en un flujo de salida dado como primer parámetro el contenido del carro de la compra de un usuario, pasado este como segundo parámetro, con el formato de este ejemplo:

```
Carrito de compra de lucas [Artículos: 2]
 Cant. Artículo
============================================================
   1   [111] "The Standard Template Library", 2002. 42,10 €
   3   [110] "Fundamentos de C++", 1998. 35,95 €
```

En el ejemplo, *lucas* es el identificador del usuario, no su nombre (observe que la inicial está en minúscula), y se imprime la cantidad de cada artículo del carro, seguido del artículo correspondiente.

5.4. La clase Numero

- Esta clase representa el número troquelado en el anverso de una tarjeta de crédito. Se almacenará este dato como un atributo de tipo *Cadena*, ya que este «número» puede tener espacios de separación al principio, al final o, más normal, en medio.
- Por tanto su constructor recibirá como parámetro esa *Cadena* con el número. Tendrá que quitarle los blancos y comprobar que es un número válido. Si no lo fuera lanzará la excepción *Numero::Incorrecto*. Vea la sección 5.7 para saber cómo averiguar si un número de tarjeta de crédito es válido.
  - Dentro de *Numero* se definirá la enumeración *Razon* con los elementos *LONGITUD*, *DIGITOS* y *NO_VALIDO*, para representar por qué un *Numero* no es válido,
  - y la clase *Incorrecto*, con un atributo de tipo *Numero::Razon*, el constructor que recibe una *Razon* como parámetro, y el método observador *razon* que devuelve el atributo.
- La clase *Numero* tendrá también un operador de conversión a cadena de caracteres constantes de bajo nivel, y deberá definirse el operador «menor-que» para dos objetos de la clase.

5.5. La clase Tarjeta

- Sus atributos son:
  - el tipo, que es una enumeración definida dentro de la clase públicamente de nombre *Tipo* con los valores *Otro*, *VISA*, *Mastercard*, *Maestro*, *JCB* y *AmericanExpress*.
  - un *Numero*, que es el número de la tarjeta tal como viene troquelado;
  - un puntero a *Usuario* constante, que es su titular;
  - una *Fecha*, que es la de caducidad,
  - y un booleano, que representa si la *Tarjeta* está activa o no.
- Se construye únicamente a partir del *Numero*, el *Usuario* y la *Fecha* de caducidad. Si esta fecha es anterior a la actual, el constructor lanzará la excepción *Tarjeta::Caducada*. Esta clase de excepción tendrá un atributo que almacenará la fecha caducada, un constructor que la reciba como parámetro y un método observador *cuando()*, que la devolverá.

  Una *Tarjeta* se crea siempre activada.

  El constructor deberá asociar la tarjeta que se está creando con su *Usuario* correspondiente, llamando a *Usuario::es_titular_de* sobre su titular.

  El constructor deberá calcular el *Tipo*; este está determinado por los primeros dígitos del *Numero*. Para simplificar (aunque no sea muy correcto), supondremos que el tipo es:

AmericanExpress Si el *Numero* empieza por 34 o 37,

JCB Si el *Numero* empieza por 3, salvo 34 o 37,

VISA Si el *Numero* empieza por 4,

Mastercard Si el *Numero* empieza por 5,

Maestro Si el *Numero* empieza por 6, y

Otro En cualquier otro caso.

Al igual que en el caso de *Usuario*, no puede haber 2 tarjetas con el mismo número, por lo que habrá de seguirse la misma estrategia, como se describe en 5.7. En el caso de que el constructor reciba como parámetro un *Numero* ya existente, se lanzará la excepción *Tarjeta::Num_duplicado*, cuyo constructor recibe el *Numero* en cuestión. Esta clase de excepción tiene un atributo que es el *Numero*, y un método observador *que* que lo devuelve.

- Dos tarjetas no se pueden copiar, ni al crearse ni por asignación. Esto sería ilegal, no puede haber dos tarjetas iguales.
- Habrá un método observador para cada atributo, que lo devuelva. Se llamarán *tipo*, *numero*, *titular*, *caducidad* y *activa*.
- Habrá también un método modificador para activar o desactivar una *Tarjeta*. Se llamará *activa* (siendo por tanto una sobrecarga del método observador del punto anterior) y podrá recibir un parámetro booleano, cuyo valor predeterminado será *true*, y devolverá un booleano que será el nuevo valor después de la activación o desactivación. Se definirá también una clase de excepción *Tarjeta::Desactivada*, vacía, para uso posterior en otros casos de uso.
- Cuando un *Usuario* se destruya, sus tarjetas asociadas seguirán «vivas», aunque obviamente ya no pertenezcan a nadie. Por lo tanto, la clase *Tarjeta* tendrá un método modificador llamado *anula_titular* que dará al puntero que representa al titular el valor nulo y desactivará la *Tarjeta* poniendo a *false* el atributo correspondiente. El destructor de la clase *Usuario* llamará a este método para cada una de sus tarjetas.
- Al destruirse un objeto de tipo *Tarjeta* deberá desasociarse de su *Usuario*, llamando a *Usuario::no_es_titular_de* sobre su titular, en caso de que este no haya sido destruido previamente; de lo contrario, la *Tarjeta* habrá sido desligada de su *Usuario* al ser destruido este (vea el punto anterior).
- Se sobrecargará el operador de inserción en flujo (<<) para mostrar o imprimir una *Tarjeta* en un flujo de salida. El formato será:

  tipo
  número
  titular facial
  `Caduca:` *MM/AA*

  donde *MM* es el mes de la fecha de caducidad, expresado con dos dígitos y *AA* son los dos últimos dígitos del año; por ejemplo: 05/09 sería mayo de 2009.

El *titular facial* es el nombre y apellidos del propietario de la tarjeta, en mayúsculas.

Si quiere, por estética, puede dibujar líneas rodeando la información impresa de la tarjeta, simulando, aun pobremente, su aspecto.

Para imprimir el nombre del tipo de la tarjeta (VISA, American Express...), deberá sobrecargar también el operador de inserción para *Tarjeta::Tipo*.

Ejemplo de impresión de una *Tarjeta* (las líneas son opcionales):

```
 --------------------
/                    \
| American Express   |
| 378282246310005    |
| SISEBUTO RUSCALLEDA|
| Caduca: 11/23      |
\--------------------/
```

- Dos *Tarjeta* podrán ordenarse por sus números. Para ello tendrá que definir el operador *menor-que* de dos tarjetas.

5.6. El Makefile

Se proporcionará a través del campus virtual el *Makefile* para la compilación. Observe que al principio hay una parte configurable que puede querer cambiar (compilador y sus opciones), y una variable que deberá cambiar imperiosamente para que funcione el objetivo *dist*, descrito más adelante. Esta variable se llama *NOMBREALUMNO* y debe llamarse como el nombre del directorio superior de trabajo, que según se pide en las instrucciones de entrega de prácticas, debe ser de la forma `Ap1_Ap2_Nom`, siendo `Ap1` el primer apellido con la inicial en mayúscula, `Ap2` lo mismo para el segundo apellido, y `Nom` para el nombre de pila. Por ejemplo, el hipotético (más vale que lo sea) alumno de nombre Pánfilo Pancracio y de apellidos Povedilla Putiérrez, nombrará su directorio así: `Povedilla_Putierrez_PanfiloPancracio`.

Dentro de ese directorio general habrá uno por cada práctica, nombrado `P`i, siendo i el número de práctica: `P0`, `P1`...

El directorio donde debe estar el fichero `luhn.cpp`, que contiene el código de la función *luhn* para comprobar si un número de tarjeta es válido, está al mismo nivel de los directorios de prácticas Px, para que no haya que copiarlo en cada práctica, ya que siempre es el mismo. Lo mismo ocurre con el directorio `Tests-auto`, donde están los programas de pruebas unitarias automáticas.

En resumen, la estructura de directorios sería:

```
Povedilla_Putierrez_PanfiloPancracio/P1/cadena.[ch]pp
                                      |  |fecha.[ch]pp
                                      |  |Makefile
                                      |  | ...
                                      /Tests-auto/...
                                      |luhn.cpp
```

```
                                        /P2/
                                           |Makefile
                                           |test-caso1-consola.cpp
                                           |usuario.[hc]pp
                                           |tarjeta.[hc]pp
                                           |articulo.[hc]pp
                                           |Make_check.mk
                                           |compra_check.cpp
  Povedilla_Putierrez_PanfiloPancracio.tar.gz    # creado por make dist
```

Si utiliza el compilador GNU C++ o LlVM clang++, se recomienda dejar las opciones *-Wall* y *-pedantic* en *CXXFLAGS*. Debe dejar también en la variable *STD* el valor *c++11* o *c++14* según la versión del compilador, puesto que se usan algunas características de C++11.

El primer objetivo (*all*) construye dos programas de prueba, los cuales se pueden ejecutar (y compilar si fuera necesario) con los objetivos *test-consola* y *test-auto*. El objetivo *clean* limpia el directorio de ficheros sobrantes (módulos objeto, respaldos del editor, ejecutables, etc.). Y el objetivo *dist* crea un archivo *tar.gz*, con los ficheros que tendrá que entregar, en el directorio superior. Por último, *distclean* borra también este archivo.

El objetivo *check* compila si es preciso y ejecuta un programa que hace algunas comprobaciones sobre el código fuente; se suministra el código del programa en el fichero *compra_check.cpp* y un *makefile* auxiliar llamado *Make_check.mk*.

Lea los comentarios en el *Makefile* para más indicaciones y objetivos.

5.7. Pistas e indicaciones

Cifrado de la contraseña en la clase *Clave* Lo más simple, aparte de otros métodos triviales, pero aún manteniendo cierta dificultad para el descifrado «ilícito», puede ser el viejo algoritmo DES, implementado en la función de C (estándar POSIX.1) *crypt*, declarada en `<unistd.h>`. Vea en el Manual de UNIX *crypt(3)*; p. ej., en consola, con la orden `man 3p crypt` o `man crypt`. Para que la *sal* sea aleatoria, deberá hacer también uso de otras funciones de C como *(s)rand* (en la cabecera `<cstdlib>`, vea *rand(3)* como antes) o, si es valiente, las clases de la cabecera `<random>` de C++11:
http://en.cppreference.com/w/cpp/numeric/random .

Validez del número de la tarjeta Un número de tarjeta de pago válido debe contener solo dígitos, y esta cadena de solo dígitos debe tener una longitud comprendida entre 13 y 19, incluidos. Así que primero, en el constructor de *Numero*, hay que quitar los espacios en blanco y ver si hay algún carácter no dígito. Después hay que comprobar la longitud, y por último la validez, ya que el último dígito de la tarjeta es de control. Para esta tarea se debe usar el algoritmo de Luhn, que puede consultar por ejemplo en la *Wikipedia*:

http://en.wikipedia.org/wiki/Luhn_algorithm

Se proporciona el fichero *luhn.cpp* con una implementación en C++.

Comprobación de usuario duplicado Para evitar la duplicidad de usuarios, guardaremos los identificadores en un conjunto desordenado (*unordered_set*) estático privado. Como

un conjunto no admite elementos repetidos, al insertar en él un identificador que ya esté, el método *unordered_set::insert()*, que devuelve un *par* (clase *pair*) formado por un iterador apuntando al sitio de la inserción y un booleano, devolverá el par *(unordered_set::end(), false)*, así que se puede comprobar por ejemplo el segundo (*second*) valor del par: si da falso es que el identificador ya existía en el conjunto y por tanto está duplicado.

Comprobación de tarjeta duplicada Para evitar la duplicidad de tarjetas, guardaremos los *Numero* en un conjunto ordenado (*set*) estático privado. Como un conjunto no admite elementos repetidos, al insertar en él un *Numero* que ya esté, el método *set::insert*, que devuelve un *par* (clase *pair*) formado por un iterador apuntando al sitio de la inserción y un booleano, devolverá el par *(set::end(), false)*, así que se puede comprobar por ejemplo el segundo (*second*) valor del par: si da falso es que el *Numero* ya existía en el conjunto y por tanto está duplicado.

Contenedores desordenados Los contenedores desordenados (*unordered_...*) requieren una función *hash* sobre su clave, así como que estas se puedan comparar con el operador de igualdad ==. Por defecto, se usa como función *hash* la suministrada como plantilla por la Biblioteca Estándar `std::hash<Key>`, siendo `Key` el tipo de la clave; la Biblioteca proporciona una especialización de esa función para cada tipo básico, incluidos los punteros, y algunos de la Biblioteca como *std::string*.

Como en nuestro caso se ha dicho en el apartado anterior sobre comprobación de usuario duplicado que se cree un conjunto desordenado de identificadores y estos son de tipo *Cadena*, entonces tenemos que definir una función *hash* para este tipo nuestro.

La forma más fácil es aprovechar la función estándar *hash* para *string*, haciendo uso de conversiones. Esto lo lograremos añadiendo al fichero `../P1/cadena.hpp` el siguiente código:

```
// Para P2 y ss.
// Especialización de la plantilla hash<T>para definir la
// función hash a utilizar con contenedores desordenados de
// Cadena, unordered_[set|map|multiset|multimap].
namespace std {
  template <> struct hash<Cadena> {
    size_t  operator()(const Cadena& cad) const
    { // conversión const char* ->string
      return hash<string>{}(cad.c_str());
    }
  };
}
```

Debe incluirse también en dicho fichero la cabecera `<functional>`, que es donde se define la plantilla `hash`.

Excepcionalmente, este es el único sitio de la práctica donde se permite el uso de la palabra *string*.

Para mayor claridad, se expone el código anterior con más comentarios y más largo, con objetos intermedios:

```
namespace std { // Estaremos dentro del espacio de nombres std
  template <> // Es una especialización de una plantilla para Cadena
  struct  hash<Cadena> { // Es una clase con solo un operador publico
    size_t  operator() (const Cadena& cad) const // el operador función
    {
      hash<string> hs;              // creamos un objeto hash de string
      const char* p = cad.c_str(); // obtenemos la cadena de la Cadena
      string  s(p);                 // creamos un string desde una cadena
      size_t  res  = hs(s);         // el hash del string. Como hs.operator()(s);
      return res;                   // devolvemos el hash del string
   }
  };
}
```