Prácticas Programación Orientada a Objetos

Índice general

Introducción a las Prácticas de POO

En este PDF vas a encontrar las prácticas de la asignatura de Programación Orientada a Objetos del curso 2023 - 2024. Contiene una explicación detallada de lo que se va a hacer en cada una de las prácticas así como el código ya implementado.

Constará de 5 prácticas donde las 2 primeras las (prácticas 0 y 1) se centrarán en la definición e implementación de dos clases para poder aprender así los principios de la Programación Orientada a Objetos, es decir, el primer parcial.

Las prácticas restantes (prácticas 2 y 3) vamos a aprender a como relacionar clases entre sí teniendo como referencia algunos diagramas de clases que nos proporcionará el enunciado de la misma.

En la práctica 4 tendremos que hacer uso de polimorismo y de mecanismos de conversiones explicitas para poder convertir punteros de la clase Base a clases Derivadas.

Cada fichero de cabecera irá en un `.hpp` y la implementación de los métodos de cada fichero de cabecera irá en un `.cpp`.

Para comprobar que las prácticas están resueltas correctamente se hará uso de un entorno de pruebas en *Docker*, en el Campus Virtual se encuentra toda la información para su correcta instalación.

1. Práctica 0

En esta primera práctica de la asignatura de Programación Orientada a Objetos, creamos las clases Fecha y Cadena.

Siendo la primera una clase la cual podremos crear Fechas en español (como su propio nombre dice) y la segunda como una çopia' a la clase `string` donde veremos el uso de memoria dinámica, el uso de varios constructores .

1.1. Clase Fecha

Esta clase Fecha, va a tener varios constructores:

- Constructor de 3 parámetros (dia, mes, año):

  ```
  Fecha a(18, 7, 1936); // 18 de julio de 1936
  ```

- Constructor de 2 parámetros (dia y mes):

  ```
  Fecha a(18, 7); // 18 de julio de 2024
  ```

- Constructor de 1 parámetro (dia), siendo el mes y año actuales:

  ```
  Fecha a(18); // 18 de julio de 2024
  ```

- Constructor sin parámetros que tendrá la Fecha del sistema:

  ```
  Fecha hoy; //fecha actual del sistema
  ```

- Constructor a partir de una cadena de caracteres de bajo nivel (`const char *`), con el formato ("dd/mm/aaaa").

  ```
  Fecha h("1/01/2000"); // 1 de enero de 2000
  ```

  En este caso, si algún campo fuera 0, por ejemplo (10/0/2010), el mes sería el del sistema. Para hacer esto, haremos uso de la librería `<ctime>`.

Estos constructores deben de comprobar que las fechas que se introduzcan sean correctas, es decir, se tiene que comprobar que hay 12 meses por año y que cada mes tiene sus dias correspondientes, comprobando si Febrero tiene 28 ó 29 (en caso de que sea año bisiesto). Tendremos un rango de años delimitados por las variables globales de la clase (`static`) `Fecha::AnnoMinimo` y `Fecha::AnnoMaximo`, siendo (1902 y 2037) los valores, respectivamente.

En el caso de que a la hora de comprobar las fechas, estas no sean correctas, en los constructores se lanzará una excepción del tipo `Fecha::Invalida`, la cual llevará información sobre por qué ha ocurrido dicha excepción en forma de cadena de caracteres de bajo nivel (parámetro al construirla), y que se devolverá con un método público llamado `por_que()`.

Vamos a poder convertir una cadena de bajo nivel a un tipo `Fecha`, por lo que tendremos un operador de conversión.

La conversión repetida de una misma fecha en cadena de caracteres conlleva un coste en tiempo que podemos reducir si mantenemos una copia de la cadena obtenida en la primera conversión, que se devolverá directente en las sucesivas conversiones solicitadas, a menos que la fecha haya cambiado, en cuyo caso se actualizará esta copia en la primera petición de conversión que ocurra tras el cambio. Así pues, se añadirán dos atributos a la clase `Fecha`: uno llamado `crep` para guardar la fecha en forma de cadena cuando sea convertida por primera vez, de tipo **vector de char** y longitud fija y suficiente para cualquier fecha; y otro **booleano** de nombre `actual`, que indicará si la cadena está actualizada.

La posible actualización de la cadena ha de ser transparente para el usuario de la clase Fecha, de modo que aunque la fecha no debe ser alterada desde el punto de vista lógico (para el usuario será la misma antes y después de la conversión), la cadena almacenada sí que ha de poder ser modificada internamente. La incorporación de los dos nuevos atributos implica cambios en algunos de los métodos ya descritos de la clase Fecha, por lo que deben ser revisados para realizar las modificaciones pertinentes.

Podemos incrementar o decrementar fechas en 1 día mediante los operadores de incremento y decremento por pre/sufijos. Una fecha tbién podrá incrementarse un número concreto de dias, mediante los operadores de suma o resta de Fecha y entero con asignación. Al igual que en los constructores habrá que comprobar que la fecha resultante de la operación sea válida, ya que podría sobrepasarse el rango de años.

Podemos asignar Fechas.

Una Fecha poseerá métodos observadores que devolverán los atributos. Estos métodos se llamarán `dia()`, `mes()` y `anno()`.

Podemos comparar dos fechas mediante los operadores lógicos (==, !=, <, <=, >, >=).

Estos operadores obtienen un resultado booleano que refleja si una fecha es menor, mayor, igual, etc., que otra.

1.1.1. Fecha.hpp

Definición de la clase Fecha:

```
#ifndef FECHA_HPP_
#define FECHA_HPP_
//incluimos las librerias
#include <iostream>
#include <ctime> //para la fecha del sistema
#include <utility> //para los operadores lógicos
#include <stdio.h> //sscanf
#include <cstring>
#include <cstdio>
```

```
class Fecha{
  public:
    //Ctor de 3 parámetros.
    explicit Fecha(int dia =0, int mes=0, int anno=0);
    //Ctor a partir de cadena de bajo nivel -> dd/mm/aaaa.
    Fecha (const char*);
    //operador de conversion de cadena de bajo nivel a fecha
    operator const char*()const noexcept;
    //atributos publicos que definen el rango de años.
    static const int AnnoMinimo = 1902;
    static const int AnnoMaximo=2037;

    //observadores de la clase
    inline int dia()const noexcept{return dia_;}
    inline int mes()const noexcept{return mes;}
    inline int anno()const noexcept{return anno_;}

    //Operadores incremento y decremento de la fecha
    Fecha& operator ++();
    Fecha operator ++(int);
    Fecha& operator --();
    Fecha operator --(int);

    Fecha operator +(int) const;
    Fecha operator -(int) const;

    Fecha& operator += (int);
    Fecha& operator -= (int);

    //clase de la excepcion
    class Invalida{
        public:
            Invalida(const char* c):motivo_(c){}
            const char* por_que()const {return motivo_;}
        private:
            const char* motivo_;
    };
  private:
    //atributos de los objetos clase
    int dia_,mes,anno_;
    mutable char crep[45];
    mutable bool actual;
    //métodos privados comprobadores de fecha
    void comprueba_fecha()const; //comprueba que la fecha sea válida
    int ultimo_dia()const; //devuelve el último dia del mes
    void actualizar()const;
};
```

```
//Operadores lógicos de comparación
bool operator == (const Fecha& , const Fecha&);
bool operator != (const Fecha&, const Fecha&);
bool operator < (const Fecha&, const Fecha&);
bool operator <= (const Fecha&, const Fecha&);
bool operator > (const Fecha&, const Fecha&);
bool operator >= (const Fecha&, const Fecha&);

#endif //Fecha.hpp
```

1.1.2. Fecha.cpp

Implementación de los métodos definidos en el fichero de cabecera `fecha.hpp`.

```
#include "fecha.hpp"
/*-----Métodos privados de la clase Fecha-----*/
void Fecha::comprueba_fecha ()const{
    //Comprobamos que el día es correcto
    if(dia_>ultimo_dia()||dia_ < 1){
       Fecha::Invalida D_Invalido("ERROR: Dia introducido es incorrecto");
        throw D_Invalido;
    }
    if(anno_ < Fecha::AnnoMinimo || anno_ > Fecha::AnnoMaximo){
       Fecha::Invalida A_Invalido("ERROR: Ano introducido es incorrecto");
        throw A_Invalido;
    }
}
int Fecha::ultimo_dia ()const{
   if(mes==1||mes==3||mes==5||mes==7||mes==8||mes==10||mes==12)return 31;
        else if(mes==4||mes==6||mes==9||mes==11)return 30;
        else if(mes == 2){
            //Comprobamos que sea bisiesto
            if((anno_%4 == 0 && anno_%100 !=0)||anno_%400==0) return 29;
            else return 28;
        }
    else{
       Fecha::Invalida M_Invalido("ERROR: Mes introducido es incorrecto");
        throw M_Invalido;
    }
}

void Fecha::actualizar()const{
    //Codificación en UTF-8
    std::locale::global(std::locale("esES.utf8"));
    //Vamos a pasar la fecha a cadena sin formato
    //Desformateamos la fecha
    std::time_t tamsistema = std::time(nullptr);
```

```
    std::tm* tamdesformato = std::localtime(&tamsistema);
    tamdesformato -> tm_year = anno_-1900;
    tamdesformato -> tm_mon = mes-1;
    tamdesformato -> tm_mday = dia_;
    mktime(tamdesformato);
    //Ahora guardamos todo en el vector de cadena
    strftime(crep,sizeof(crep),"%A %d de %B de %Y",tamdesformato);
}

/*----- Métodos públicos de la clase Fecha*/
//Implementación del constructor
Fecha::Fecha(int dia, int mes, int
 ↪  anno):dia_(dia),mes(mes),anno_(anno),actual(false){
    //Si los atributos son 0, vamos a poner la fecha del sistema (ctime)
    std::time_t tamsistema = std::time(nullptr);
    std::tm* tamformato = std::localtime(&tamsistema);
    //Comprobamos si la fecha introducida está por defecto
    if(anno_ == 0) anno_ = tamformato -> tm_year+1900;
    if(mes == 0) mes = tamformato -> tm_mon +1;
    if(dia_ == 0) dia_ = tamformato ->tm_mday;
    //Comprobamos la fecha
    comprueba_fecha();
}
Fecha::Fecha(const char* c){
    //Introducimos la fecha de la manera dia/mes/año, tenemos que sacar el
     ↪  formato
    // y guardar en cada variable los datos correspondientes
    //Comprobamos el formato
    if(sscanf(c,"%d/%d/%d",&dia_,&mes,&anno_)!=3){
        Fecha::Invalida F_Invalido("ERROR: Formato de fecha incorrecto");
    }
        //Formateamos la fecha
        std::time_t tamsistema = std::time(nullptr);
        std::tm* tamformato = std::localtime(&tamsistema);
         //Comprobamos si la fecha introducida está por defecto
        if(anno_ == 0) anno_ = tamformato -> tm_year+1900;
        if(mes == 0) mes = tamformato -> tm_mon +1;
        if(dia_ == 0) dia_ = tamformato ->tm_mday;
        //Comprobamos la fecha
        comprueba_fecha();
}
//Implementación del operador de conversion a const char*
Fecha::operator const char*()const noexcept{
    //Comprobamos que no se haya actualizado la fecha antes
    if(!actual){//se introduce por primera vez
        actualizar();
        actual = true; //actualizada
```

```
    }
    return crep;
}

//Implementación de los operadores de incremento y decremento
Fecha& Fecha::operator +=(int n){
    time_t tiempo_hasta_ahora = time (NULL);
        struct tm* fecha = localtime(&tiempo_hasta_ahora);
    fecha->tm_mday = dia_+n;
    fecha->tm_mon = mes -1;
    fecha->tm_year = anno_ -1900;
    mktime(fecha);

    //modificamos los atributos de esa fecha
    dia_ = fecha->tm_mday;
    mes = fecha->tm_mon+1;
    anno_ =fecha->tm_year+1900;

    //Comprobamos que la fecha sea válida
    comprueba_fecha();
    return *this; //devolvemos la fecha modificada
}
Fecha& Fecha::operator -= (int n){return *this += -n;}
Fecha& Fecha::operator ++(){return *this += 1;}
Fecha Fecha::operator ++(int){
    Fecha fecha = *this;
    *this +=1;
    return fecha;
}
Fecha& Fecha::operator --(){return *this += -1;}
Fecha Fecha::operator --(int){
    Fecha fecha = *this;
    *this += -1;
    return fecha;
}
Fecha Fecha::operator +(int n)const{
    Fecha fecha=*this;
    return fecha += n;
}
Fecha Fecha::operator -(int n)const{
    Fecha fecha=*this;
    return fecha += -n;
}


//Operadores aritméticos
bool operator == (const Fecha& f1, const Fecha&f2){
```

```
    return (f1.anno() == f2.anno() && f1.mes() == f2.mes() &&
     ↪  f1.dia()==f2.dia());
}
bool operator !=(const Fecha&f1, const Fecha&f2){
    return !(f1==f2);
}
bool operator <(const Fecha& f1, const Fecha& f2){
    return(f1.anno() < f2.anno() || (f1.anno() == f2.anno() && (f1.mes() <
     ↪  f2.mes()
        || (f1.mes() == f2.mes() && f1.dia() < f2.dia()))));
}
bool operator <=(const Fecha&f1, const Fecha& f2){
    return !(f2<f1);
}
bool operator >(const Fecha& f1, const Fecha& f2){
    return f2 < f1;
}
bool operator >=(const Fecha& f1, const Fecha& f2){
    return !(f1<f2);
}
```

1.2. Clase Cadena

Esta clase Cadena será una clase general para trabajar con cadenas de caracteres (char). Por tanto, no podemos hacer uso de la clase `std::string` de la STL.

La clase Cadena tendrá este mismo nombre y contará con dos atributos, estos serán un **puntero a caracteres de tipo char** y un **entero sin signo de tipo size_t** que representará el taño de la cadena o número de caracteres en cada momento.

El puntero antedicho apuntará a una cadena de caracteres acabada en el carácter terminador nulo o "`\0`" como las cadenas de C, lo que hace la implementación mucho más fácil, por tanto, si vamos a crear la cadena "hola" (que tiene 4 caracteres) tendremos que reservar memoria para una cadena de 5 caracteres (para el carácter terminador) y sería de la manera "hola`\0`".

Si una Cadena está **vacía** (o sea, su taño es 0), no tendrá asignada memoria dinámica y el puntero a los caracteres apuntará a la misma dirección que un tercer atributo de `tipo char[1]` cuyo único carácter será el terminador.

Los atributos antedichos deben llamarse `vacia`, `t` y `s` y ser declarados en este mismo orden (primero la cadena formada solo por el terminador, después el entero y por último el puntero).

Una Cadena puede construirse:

- Con 2 parámetros, que serán por este orden: un taño inicial y un carácter de relleno.

```
Cadena a(3, 'X'); // taño y relleno: "XXX"
```

- Con 1 parámetro, que será un taño inicial; en este caso la cadena se rellenará con espacios. No se permitirá la conversión implícita de un entero a una Cadena.

```
Cadena b(5); // taño y espacios: (5 espacios)
```

- Sin parámetros: se creará una Cadena vacía, de taño 0.

```
Cadena c; // Cadena vacía, taño 0:
```

- Por copia a partir de otra cadena

```
Cadena d(a); // copia de Cadena: "XXX"
```

- A partir de una cadena de caracteres de bajo nivel, permitiéndose las conversiones no explícitas desde const char* a Cadena.

```
Cadena f("Hola"); // copia de cadena de C: "Hola"
```

Podemos asignar una Cadena a otra de tipo Cadena o a una cadena de bajo nivel, por tanto, vamos a realizar una sobrecarga del `operador =` tanto para `Cadena` como para `const char *`.

Una Cadena se puede convertir a una cadena de bajo nivel. Se prohibirán las conversiones implícitas para evitar posibles conversiones indeseadas y problemas de ambigüedad.

Tendremos un método observador `length()` que devolverá el número de caracteres de una Cadena.

Podemos concatenar dos cadenas, añadiéndose esta al final, mediante el operador de suma con asignación (+=). Si no se puede obtener memoria para llevar a cabo la concatenación, la cadena no se debe modificar.

Dos objetos Cadena podrán concatenarse mediante el operador de suma (+), resultando una nueva Cadena que será la concatenación de ambas.

Podemos comparar Cadenas mediante los operadores lógicos (==, !=, <, <=, >, >=), estos devolverán un valor booleano.

Mediante la sobrecarga del operador de índice `[]` podemos obtener un carácter determinado de una Cadena. tbien tendremos un método `at()` que compueba si el índice pasado se encuentre en el rango (`0..length()-1`) y devuele el carácter de dicha posición. Si esto no es así se lanzará la excepción `std::out_of_range`. Estos métodos se pueden aplicar tanto a Cadenas modificables como no modificables.

El método miembto `substr()` recibirá dos parámetros enteros sin signo: un índice y un taño, y devolverá una Cadena formada por tantos caracteres como indique el taño a partir del índice. Al igual que en el caso de la función at, el método substr tbién lanzará la excepción `std::out_of_range` cuando se proporcione una posición inicial después del último carácter, o cuando la subcadena pedida se salga de los límites de la cadena.

1.2.1. Cadena.hpp

Definición de la clase Cadena:

```
#ifndef CADENA_HPP_
#define CADENA_HPP_
//Incluimos las librerías
#include <iostream>
#include <cstring>
#include <stdexcept>
class Cadena{
    public:
        //Por defecto, una cadena vacía = contiene el caracter terminador
        explicit Cadena(size_t t = 0, char c = ' ');
        //Tenemos que definir el constructor de copia debido a que estos
        ↪  haciendo uso de mem dinamica
        Cadena(const Cadena& );
        Cadena(const char*);
        //Si hacemos ctor de copia debemos de hacer operador de asignación
        ↪  por copia
        Cadena& operator = (const Cadena& )noexcept;
        Cadena& operator =(const char*)noexcept;
        //conversion a cadena de bajo nivel (observadora y modificadora)
        explicit operator const char*()const noexcept{return s;}

        //método observador -> devuelve el num de caracteres de una cadena
        size_t length() const noexcept{return tam;}
        //Operadores aritmeticos
        Cadena& operator += (const Cadena& )noexcept; //concatenación de
        ↪  cadenas (misma cadena resultante)
        //operador de indice
        char& operator [](size_t index)noexcept{return s[index];}
        char& operator[](size_t index)const noexcept{return s[index];}
        char& at(size_t );
        char& at(size_t )const;

        //Método de subcadena
        Cadena substr(unsigned , size_t)const;
        //declaramos el destructor de la cadena
        ~Cadena();

    private:
        static char vacia[1]; //vacia = char [1] = '\0';
        size_t tam; //taño de la cadena de char (num caracteres de la
        ↪  misma)
        char* s; //puntero a char

};
```

```
//concatenación de cadenas (nueva cadena resultante)
Cadena operator + (const Cadena& , const Cadena& )noexcept;

//Operadores lógicos
bool operator == (const Cadena& , const Cadena& )noexcept;
bool operator != (const Cadena& , const Cadena& )noexcept;
bool operator < (const Cadena& , const Cadena& )noexcept;
bool operator <= (const Cadena& , const Cadena& )noexcept;
bool operator > (const Cadena& , const Cadena& )noexcept;
bool operator >= (const Cadena& , const Cadena& )noexcept;

#endif // !CADENA_HPP
```

1.2.2. Cadena.cpp

Implementación de los métodos de la clase Cadena:

```
#include "cadena.hpp"
//Definimos la variable estática
char Cadena::vacia[1]={'\0'};

/*-----Métodos publicos de la clase Cadena-----*/
//Constructores de la clase cadena
Cadena::Cadena(size_t t, char c):tam(t),s(vacia){
    if(tam>0){
        s = new char[tam+1];
        for(size_t i=0; i<tam ;i++) s[i]=c;
        s[tam] = vacia[0];
    }
}

Cadena::Cadena(const Cadena& other):tam(other.tam),s(vacia){
    if(tam>0){
        s= new char[tam+1];
        //hacemos uso de strcpy
        strcpy(s,other.s);
        s[tam] =vacia[0];
    }
}

Cadena::Cadena(const char* c):tam(strlen(c)), s(vacia){
    if(tam>0){
        s = new char[tam+1];
        strcpy(s,c);
        s[tam]=vacia[0];
    }
}
```

```
//Operadores de asignación por copia
Cadena& Cadena::operator = (const Cadena& other)noexcept{
    //evitos autoasignación
    if(this!=&other){
        if(tam > 0){
            delete[] s;}
        tam = other.tam;
        if(other.tam > 0){
        s = new char[tam+1];
        strcpy(s,other.s);
        }else{s = vacia;}
    }
    return *this;
}


Cadena& Cadena::operator =(const char* c)noexcept{
    if(*this!=c){
        if(tam > 0){delete [] s;}
        tam = strlen(c);
        if(strlen(c)>0){
            s = new char [tam+1];
            strcpy(s,c);
        }
        else{s = vacia;}
    }
    return *this;
}

//Concatenación de cadenas
Cadena& Cadena::operator +=(const Cadena& other)noexcept{
    //Creamos una cadena nueva que contedrá las dos cadenas concatenadas
    char* concatenada = new char [tam+1];
    //Copiamo una de las cadenas
    strcpy(concatenada,s);
    //Modificamos el taño
    tam += other.tam;
    //destruimos la cadena original y la reemplazamos
    delete [] s;
    s = new char[tam+1];
    //Concatenamos dos cadenas
    strcpy(s,concatenada);
    strcat(s,other.s);
    //eliminamos la cadena auxiliar
    delete [] concatenada;
    //devolvemos la cadena final concatenada
```

```
    return *this;
}

Cadena operator +(const Cadena&c1, const Cadena& c2)noexcept{
    //Creamos una cadena nueva -> resultante de la concatenación
    Cadena auxiliar(c1);
    //devolvemos la cadena concatenada
    return auxiliar += c2;
}

//Operadores de índice
char& Cadena::at(size_t index){
    if(index<tam)return s[index];
    else throw std::out_of_range("ERROR at(): El indice supera el taño
    ↪  maximo de la cadena");

}
char& Cadena::at(size_t index)const{
    if(index < tam){
        return s[index];
    }
    else{
        throw std::out_of_range("ERROR at(): El indice supera el taño
        ↪  maximo de la cadena");
    }
}

Cadena Cadena::substr(unsigned index, size_t t)const{
    if(index + t > tam || t > tam || index > tam){
        throw std::out_of_range("ERROR substr(): Los valores dados superan
        ↪  el taño maximo de la cadena");
    }
    else{
        //Creamos una cadena para imprimir la subcadena
        Cadena subcadena(t);
        //Guardamos esa subcadena (indice + t);
        strncpy(subcadena.s,s+index, t);
        subcadena.s[t]='\0';
        //devolvemos la subcadena
        return subcadena;
    }
}

//Destructor de cadena
Cadena::~Cadena(){
    if(tam>0){delete [] s;}
```

```
}

//Operadores lógicos
bool operator == (const Cadena& cadena1, const Cadena& cadena2)noexcept{
    return (strcmp(&cadena1[0],&cadena2[0])==0);
}
bool operator !=(const Cadena& cadena1, const Cadena& cadena2)noexcept{
    return !(cadena1==cadena2);
}

bool operator <(const Cadena& cadena1, const Cadena& cadena2)noexcept{
    return (strcmp(&cadena1[0],&cadena2[0])<0);
}
bool operator <=(const Cadena& cadena1, const Cadena& cadena2)noexcept{
    return !(cadena2<cadena1);
}

bool operator > (const Cadena& cadena1, const Cadena& cadena2)noexcept{
    return cadena2<cadena1;
}

bool operator >=(const Cadena& cadena1, const Cadena& cadena2)noexcept{
    return !(cadena1<cadena2);
}
```

2. Práctica 1

En esta práctica seguiremos con las clases Fecha y Cadena definidas previamente en la práctica 0, donde en cada una de ellas realizaremos algunas modificaciones e incluiremos nuevos operadores de inserción y extracción en flujo e iteradores.

2.1. Clase Fecha

En la clase Fecha vamos a incluir el operador de extracción e inserción en flujo (cin y cout), respectivamente.

- **Operador de extracción (>>):** Para la extracción (lectura), y con el fin de simplificar el ejercicio, se seguirá el formato DD/MM/AAAA, donde se utilizarán 1 o 2 dígitos decimales para el día (DD) y el mes (MM), y 4 para el año (AAAA).
- **Operador de inserción (<<):** Para la inserción (salida), se utilizará el formato NDS DD de NM de AAAA, siendo NDS el nombre del día de la semana, DD el día del mes, NM el nombre del mes y AAAA el año expresado con 4 dígitos decimales.

Para evitar problemas de ambigüedad, se retirará el antiguo operador de conversión implícita a **const char***, que pasará a ser un método explícito de conversión a const char* llamado `Fecha::cadena()`.

2.1.1. Fecha.hpp modificado

Definimos los nuevos métodos:

```
class Fecha{
    public:
        //código de la P0
        //Reemplazamos operator const char*()const noexcept por:
        const char* cadena() const noexcept;
    private:
        //...
};
//Operadores lógicos
//...
//Se incluye fuera de la clase
std::ostream& operator <<(std::ostream&, const Fecha& )noexcept;
std::istream& operator >>(std::istream&, Fecha&);
```

2.1.2. Fecha.cpp modificado

Implementamos los nuevos métodos:

```
//Métodos de la P0 ya implementados

const char* Fecha::cadena() const noexcept{
    if(!actual){
        actualizar();
        actual = true;
    }
    return crep;
}

//Operadores de flujo
std::ostream& operator << (std::ostream& output, const Fecha& f)noexcept{
    return output<<f.cadena();
}
std::istream& operator >>(std::istream& input, Fecha& f){
    //reservamos memoria para la nueva cadena introducida
    char nuevafecha[12]="";
    //obtenemos lo introducido por teclado+
    input.width(12);
    input.getline(nuevafecha,12);
    //comprobamos que la fecha es válida
     try {
        f = nuevafecha; // Se intenta crear la fecha desde la cadena
    } catch (const Fecha::Invalida& e) {
        input.setstate(std::ios::failbit); // Se marca el flujo de entrada
         ↪  como fallido
        throw; // Se relanza la excepción
    }
    return input;
}
```

2.2. Clase Cadena

Al igual que en la clase Fecha, en la clase Cadena también vamos a realizar varias modificaciones. Vamos a definir los operadores de inserción y extracción en flujo:

- **Operador de inserción (<<):** La Cadena se insertará en un flujo sin cambios. Gracias a este operador, deja de ser necesaria la conversión a **const char*** para insertar una Cadena en un flujo.
- **Operador de extracción (>>):** Una Cadena se podrá extraer (leer) de un flujo. Para ello, se leerá una palabra y se reemplazará con ella el contenido que tuviera la Cadena hasta ese momento.

  Una **palabra** será una sucesión de caracteres qu no sea espacios en blancos, para poder comprobar eso, haremos uso de `isspace()`, que comprueba si el caracter es ó no un espacio en blanco. Si devuelve 0, significa que es un espacio en blanco y por tanto, se acaba la palabra. La Cadena tendrá un tamaño máximo de 32 caracteres, a lo cual habrá que sumar el terminador nulo "(`\0`)".

Por otro lado, vamos a incluir los iteradores, así como los tipos contendedores de la STL, para hacer uso de las funciones relacionadas `begin()` y `end()`.

La clase Cadena tendrá los tipos → `iterator`, `const iterator`, `reverse_iterator` y `const reverse_iterator`.

La clase Cadena tendrá los métodos miembros → `begin()`, `end()`, `cbegin()`, `cend()`, `rbegin()`, `rend()`, `crbegin()`, `crend()`.

Las funciones `(r)begin`/`(r)end` deben estar sobrecargadas para trabajar con objetos de Cadena tanto constantes como modificables.

Tenemos que incluir la librería `iterator` para poder hacer uso de estos.

Para poder crear los iteradores inversos lo vamos a realizar de esta manera:

```
typedef std::reverse_iterator<iterator> reverse_iterator;
typedef std::reverse_iterator<const iterator>
   const_reverse_iterator;
```

Las funciones `(c)rbegin`/`(c)rend` se definen creando objetos a partir de los iteradores directos, pero utilizando los del extremo opuesto:

```
inline Cadena::reverse_iterator Cadena::rbegin() noexcept{
    return reverse_iterator(end());
}

inline Cadena::reverse_iterator Cadena::rend() noexcept{
    return reverse_iterator(begin());
}
```

Del mismo modo, se definen las versiones const sobrecargadas y las explícitas con la c delante.

Por último, vamos a realizar la semántica de movimiento donde debe tener en cuenta que la Cadena inicial debe quedar en un estado válido tras la operación. Por ello, la Cadena original pasará a ser la Cadena vacía una vez realizado el movimiento.

2.2.1. Cadena.hpp modificada

Incluimos los nuevos métodos definimos:

```
class Cadena{
    public:
        //Código de la P0

        //Semantica de movimiento
        Cadena(Cadena&&); //Constructor
        //Operador asignación movimiento
        const Cadena& operator = (Cadena&& )noexcept;

        //Iteradores
        typedef char* iterator;
        typedef const char* const_iterator;
        typedef std::reverse_iterator<iterator> reverse_iterator;
        typedef std::reverse_iterator<const_iterator>
        ↪  const_reverse_iterator;

        //Podemos implementarlos aqui mediante inline, ya que son
        ↪  funciones muy cortitas
        inline iterator begin() noexcept { return s_; }
        inline const_iterator begin() const noexcept { return s_; }
        inline const_iterator cbegin() const noexcept { return s_; }
        inline iterator end() noexcept { return s_ + tam_; }
        inline const_iterator end() const noexcept { return s_ + tam_; }
        inline const_iterator cend() const noexcept { return s_ + tam_; }
        inline reverse_iterator rbegin() noexcept{
            return reverse_iterator(end()); }
        inline const_reverse_iterator rbegin() const noexcept{
            return const_reverse_iterator(end()); }
        inline const_reverse_iterator crbegin() const noexcept{
            return const_reverse_iterator(end()); }
        inline reverse_iterator rend() noexcept{
            return reverse_iterator(begin()); }
        inline const_reverse_iterator rend() const noexcept{
            return const_reverse_iterator(begin()); }
        inline const_reverse_iterator crend() const noexcept{
            return const_reverse_iterator(begin()); }
    private:
        //...
};
```

```
//Operadores de inserción y extracción de flujo
std::ostream& operator << (std::ostream& , const Cadena&)noexcept;
std::istream& operator >>(std::istream&, Cadena&)noexcept;

//Esto lo incluiremos para poder pasar los tests de la P2 en adelante.
//codigo hash Para P2 y ss .
// Especialización de la plantilla std :: hash<Key> para definir la
↪  función hash a usar
// en contenedores desordenados de Cadena,
↪  unordered_[set|map|multiset|multimap].
namespace std{ //Estaremos dentro del espacio de nombres std
    template <> // Es una especializaci ón de una plantilla para Cadena.
    struct hash <Cadena>{
        size_t operator() (const Cadena& cad) const //el operador funcion
        {
            hash<string> hs;
            auto p{(const char*)(cad)};
            string s{p};
            size_t res{hs(s)};
            return res;
        }
    };
}
```

2.2.2. Cadena.cpp modificada

Incluimos la implementación de los nuevos métodos definidos en la P1:

```
//Constructor de Movimiento
Cadena::Cadena(Cadena&& other):s_(other.s_),tam_(other.tam_){
    if(tam_ > 0){
        other.tam_ = 0;
        other.s_ = vacia;
    }
}
//operador de asignación por movimiento
const Cadena& Cadena::operator = (Cadena&& other)noexcept{
    if(this!= &other){
        if(tam_ != other.tam_){
            tam_=other.tam_;
        }
        s_ = other.s_;
        other.tam_=0;
        other.s_=vacia;
    }
    return *this;
}
```

```
//Operadores de inserción y extracción de flujo
std::ostream& operator << (std::ostream& output, const Cadena& c)noexcept{
    return output << c.operator const char *();

}

std::istream& operator >> (std::istream& input, Cadena& c)noexcept{
    //vamos a introducir una nueva cadena
    char nuevacadena[33]="";
    input.width(33);
    input>>nuevacadena;
    c = nuevacadena;
    return input;
}
```

3. Práctica 2

En las practicas 0 y 1 trabajamos con las clases Fecha y Cadena, una vez implementadas correctamente y funcionales vamos a trabajar con nuevas clases que haran uso de estas dos.

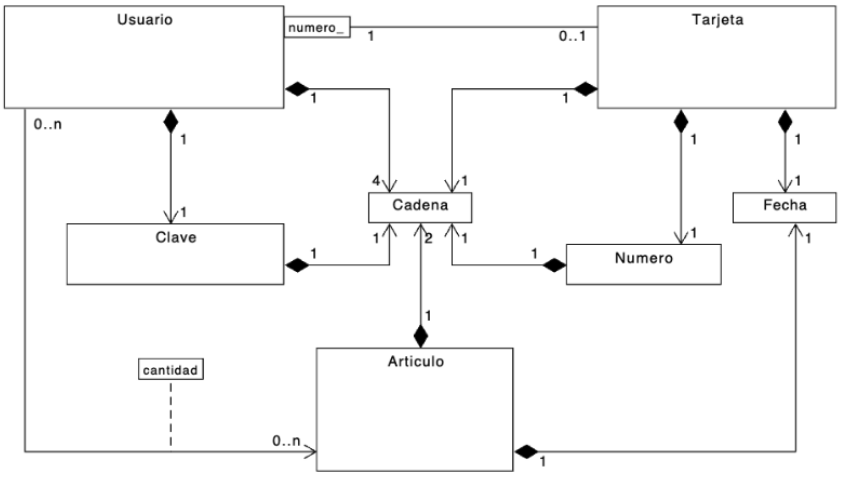

Figura 1: Diagrama de clases de los casos de uso 1 y 2

Mediante este diagrama vamos a crear las nuevas clases. La clase Artículo irá en los ficheros articulo.[ch]pp. Las clases Usuario y Clave se escribirán en los ficheros usuario.[ch]pp. Las clases Tarjeta y Numero se escribirán en los ficheros tarjeta.[ch]pp.

3.1. Clase Artículo

La clase Artículo contiene 5 atributos → código de referencia, título, fecha de publicación, precio y número de ejemplares a la venta. Los 3 primeros son no modificables.

Un objeto de la clase Artículo se construye mediante esos 5 atributos en el orden: referencia, título, fecha de publicación, precio y existencias. Este último parámetro es opcional: si no se suministra, se toma como cero.

La clase Artículo tendrá los métodos observadores: `referencia()`, `titulo()`, `f_publi()`, `precio()` y `stock()`. Estos dos últimos estarán sobrecargados para devolver una referencia al atributo correspondiente, con el fin de permitir su modificación.

Finalmente, contará con un operador de inserción en flujo (<<) que imprimirá los datos de un artículo con el formato:

```
"Fundamentos de C++", 1998. 29,95 €
```

3.1.1. Articulo.hpp

```
#ifndef ARTICULO_HPP
#define ARTICULO_HPP

//Inclusión de librerías
#include "../P1/fecha.hpp"
#include "../P1/cadena.hpp"
#include <iostream>
#include <iomanip>
#include <locale.h>
class Articulo{
public:
    Articulo(Cadena referencia, Cadena titulo ,Fecha f_publi
    ,double precio,unsigned ejemplares
    ↪  =0):referencia_(referencia),titulo_(titulo),
    fpubli_(f_publi),precio_(precio),ejemplares_(ejemplares){}

    //Observadores de la clase
    inline const Cadena& referencia() const noexcept{return referencia_;}
    inline const Cadena& titulo()const noexcept{return titulo_;}
    inline const Fecha& f_publi()const noexcept{return fpubli_;}
    inline double precio()const noexcept{return precio_;}
    inline double& precio()noexcept{return precio_;}
    inline unsigned stock()const noexcept{return ejemplares_;}
    inline unsigned& stock()noexcept{return ejemplares_;}

private:
    const Cadena referencia_,titulo_;
    const Fecha fpubli_;
    double precio_;
    unsigned ejemplares_;
};

//Operador de inserción en flujo
std::ostream& operator <<(std::ostream& , const Articulo&)noexcept;
#endif // !ARTICULO_HPP
```

3.1.2. Articulo.cpp

```
#include "articulo.hpp"

//Operador de inserción en flujo
std::ostream& operator <<(std::ostream& output, const Articulo&
↪  art)noexcept{
std::locale::global(std::locale(""));
output<<"["<<art.referencia()<<"] "<<"\""<<art.titulo()<<"\""<<", "
        <<art.f_publi().anno()<<". "
        <<std::fixed<<std::setprecision(2)<<art.precio()<<" €";
return output;
}
```

3.2. Clase Tarjeta y Número

Consta de dos clases **Numero** y **Tarjeta** las cuales ambas se definen en tarjeta.hpp y se implementan en tarjeta.cpp.

3.2.1. Clase Numero

Un Numero contendrá un atributo de tipo Cadena, ya que este «número» puede tener espacios de separación al principio, al final o, más normal, en medio.

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**. Para saber si un dígito de dicho número es un caracter en blanco, recorreremos el número entero y mediante `isspace()`.

Dentro de la clase Numero vamos a declarar 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.

También contendrá un operadore de conversión a cadena de caracteres de bajo nivel, y deberá definirse el operador «menor que» para dos objetos de la clase.

3.2.2. Clase Tarjeta

En la clase Tarjeta vamos a encontrar un tipo enum para indicar el tipo de la tarjeta, siendo las opciones: Otro, VISA, Mastercard, Maestro, JCB y AmericanExpress.

Tendrá varios atributos como:

- Un Numero constante, que es el número de la tarjeta que viene troquelado,
- Un puntero a Usuario, que es el titular,
- Un Fecha constante, que es la de cadudidad,
- Un booleano que indicará si la Tarjeta está o no activa, siempre se crea activa.

La Tarjeta se construirá solamente a partir del Numero, el Usuario y la Fecha de caducidad. Esta última es importante debido a que vamos a declarar la clase de excepción **Tarjeta::Caducada**.

Esta clase de excepción tendrá un atributo que almacena la fecha caducada, un constructor que la reciba como parámetro observador `cuando()`, que lo devolverá.

En el constructor vamos a crear la asociación entre Tarjeta y Usuario mediante el método `es_titular_de()`.

También vamos a declara una clase de excepción **Tarjeta::Num_Duplicado**, debido a que no pueden haber dos Tarjetas con el mismo número. Esta clase de excepción tiene un atributo que es el Numero, y un método observador `que()` que lo devuelve.

Vamos a eliminar tanto el **constructor de copia** como el **operador de asignación por copia**, debido a que no se puede crear una Tarjeta que tenga los mismo atributos que otra.

Vamos a tener varios métodos observador que van a devolver los atributos de la Tarjeta, estos métodos serán: `numero()`, `titular()`, `caducidad()` y `activa()`, este último se sobrecargará que recibirá el nuevo estádo de la Tarjeta en un parámetro booleano y devolverá dicho estado.

Además vamos a tener el método observador `tipo()` devolverá el Tipo de la Tarjeta. El tipo de Tarjeta estará determinado por los primeros dígitos del Numero que la contiene:

- **AmericanExpress:** Los dos primeros dígitos del Numero son 34 ó 37.
- **JCB:** El primer dígito es 3 a excepción de 34 o 37.
- **VISA:** El primer dígito es 4.
- **Mastercard:** El primer dígito es 5.
- **Maestro:** El primer dígito es 6.
- **Otro:** El cualquier otro dígito.

Cuando se detruya un Usuario, como las tarjetas no se destruyen segurían "vivas" pero sin ningún titular asociado, por tanto, se llamará al método `anula_titular()` que dará al puntero que representa al titular el valor nulo y desactivará la Tarjeta poniendo a

false el atributo correspondiente, es decir, activa pasa a falso. El destructor de la clase Usuario llamará a este método para cada una de sus tarjetas.

A la hora de destruir una Tarjeta, en el destructor tenemos que eliminar la relación entre Tarjeta y Usuario, para ello llamamos al método **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).

Vamos a sobrecargar el operador de inserción en flujo **operator <<** para mostrar los atributos de la clase Tarjeta siguiendo el formato:

```
tipo tarjeta
numero tarjeta
titular facial
Caduca: MM/AA
```

```
 --------------------
/                    \
| American Express   |
| 378282246310005    |
| SISEBUTO RUSCALLEDA|
| Caduca: 11/28      |
\--------------------/
```

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: 11/28 sería noviembre de 2028.
El titular facial es el nombre y apellido concatenados en mayúsculas del usuario titular de la tarjeta.
Las lineas del formato no hace falta que las implementes, es más por estética.

Para imprimir el nombre del tipo de la tarjeta (VISA, American Express...), deberá sobrecargar también el operador de inserción para **Tarjeta::Tipo**, el cual imprimirá el texto «Tipo indeterminado» cuando el valor sea **Tipo::Otro**.

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

Tarjeta.hpp

```
#ifndef TARJETA_HPP
#define TARJETA_HPP

//Inclusión de librerias
#include "../P1/fecha.hpp"
#include "../P1/cadena.hpp"
#include "usuario.hpp"
#include <iostream>
#include <set>
#include <algorithm> //remove_if
#include <cctype> //isspace
#include <cstring>
//declaraciones adelantadas
class Usuario;
class Clave;
```

```
/*-----Clase Numero-----*/
class Numero{
public:
    //tipos de excepciones
    typedef enum {LONGITUD,DIGITOS,NO_VALIDO}Razon;
    Numero(const Cadena&);

    //operador de conversion a const char*
    inline operator const char*()const{return numero_.operator const char
    ↪  *();}
    //sobrecarga del operador < para comparar numeros
    friend bool operator < (const Numero&, const Numero&);
    //clase de la excepcion
    class Incorrecto{
        Razon razon_;
        public:
            Incorrecto(const Razon& r):razon_(r){};
            const Razon& razon()const{return razon_;}
    };

private:
    Cadena numero_;
    //metodos extras para el constructor
    Cadena eliminar_espacios(const Cadena&); //devuelve la cadena sin
    ↪  espacios
    Cadena longitud(const Cadena&); //comprueba la longitud de la cadena
    ↪  es correcta o no
};

/*-----Clase Tarjeta-----*/
class Tarjeta{
public:
    //tipos de tarjetas
    typedef enum {Otro,VISA,Mastercard,Maestro,JCB,AmericanExpress} Tipo;
    Tarjeta(const Numero&, Usuario&, const Fecha&);
    //no se pueden crear tarjetas por copia de otras
    Tarjeta(const Tarjeta&)=delete;
    Tarjeta operator =(const Tarjeta&)=delete;
    //Observadores de la clase
    const Numero& numero()const noexcept{return numero_;}
    const Usuario* titular()const noexcept{return titular_;}
    const Fecha& caducidad()const noexcept{return caducidad_;}
    bool activa()const noexcept{return activa_;}
    bool activa(bool estado) noexcept {activa_=estado;return
    ↪  activa_;}//version modificadora de la anterior
    Tipo tipo()const noexcept;
```

```
    //destructor de la clase
    ~Tarjeta();
    //Clase de la excepcion tarjeta caducada
    class Caducada{
        Fecha fecha_;
        public:
            Caducada(const Fecha fecha):fecha_(fecha){};
            const Fecha& cuando()const{return fecha_;}
    };
    //Clase de la excepcion tarjeta duplicada
    class Num_duplicado{
        Numero num_;
        public:
            Num_duplicado(const Numero& num):num_(num){};
            const Numero& que()const{return num_;}
    };
    //Clase de la excepcion tarjeta desactivada
    class Desactivada{};
private:
    const Numero numero_;
    Usuario* const titular_;
    const Fecha caducidad_;
    bool activa_;

    //método privado de la clase
    //hacemos que la clase usuario sea amiga para poder hacer uso de este
    ↪  método
    friend class Usuario;
    void anula_titular();

    //conjunto de tarjetas
    static std::set<Numero>tarjetas_;
};
std::ostream& operator <<(std::ostream&, const Tarjeta& )noexcept;
std::ostream& operator <<(std::ostream&, const Tarjeta::Tipo& )noexcept;
//sobrecarga del operador <, para ordenar tarjetas
bool operator <(const Tarjeta&,const Tarjeta&);
#endif // !TARJETA_HPP
```

Tarjeta.cpp

```
#include "tarjeta.hpp"
//inicialización del conjunto estático de tarjetas
std::set<Numero> Tarjeta::tarjetas_;
//Hacemos uso del algoritmo de luhn para ver si el numero de la tarjeta es
↪  correcto o no
```

```
bool luhn(const Cadena& numero);
/*-----Clase Numero-----*/
Numero::Numero(const Cadena& numero):numero_(longitud(numero)){
    char caracteres[] =
    ↪  "abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ./";
    //Comprobamos que está en el rango de caracteres
    if(strcspn(numero_.operator const char
    ↪  *(),caracteres)<numero_.length())
        throw Incorrecto(Razon::DIGITOS);
    //comprobamos que sea correcto
    if(!luhn(numero_))throw Incorrecto(Razon::NO_VALIDO);
}
bool operator < (const Numero& a, const Numero& b){
    return strcmp(a,b)<0;
}

//metodos privados de la clase
Cadena Numero::eliminar_espacios(const Cadena& cadena){
    //creamos una cadena nueva como copia de la introducida
    Cadena aux(cadena);
    const char* original = cadena.operator const char *();
    int j =0;
    for(size_t i =0; i!=strlen(original); i++){
        if(!isspace(original[i])){
            aux[j++] = original[i];
        }
    }
    aux[j] = '\0';
    return Cadena(aux.operator const char *());
}
Cadena Numero::longitud(const Cadena& cadena){
    //creamos una cadena como copia de la introducida para calcular la
    ↪  longitud sin espacios
    Cadena aux = eliminar_espacios(cadena);
    if(aux.length()> 19 || aux.length() < 13 || aux.length() <= 0)
        throw Incorrecto(Razon::LONGITUD);
    return aux;
}

/*-----Clase Tarjeta-----*/
Tarjeta::Tarjeta(const Numero& numero,  Usuario& titular, const Fecha&
↪  caducidad):

    ↪  numero_(numero),titular_(&titular),caducidad_(caducidad),activa_(true){
    if(caducidad_ < Fecha())throw Caducada(caducidad_); //caducada¿?
    //Comprobamos que la tarjeta no está registrada
    if(!tarjetas_.insert(numero).second)throw Num_duplicado(numero);
```

```
    //No caducada y numero correcto -> se asigna al usuario
    titular_ -> es_titular_de(*this);
}
Tarjeta::Tipo Tarjeta::tipo()const noexcept{
    switch (numero_[0]){
    case '3':
        if(numero_[1]=='4' || numero_[1]=='7') return
        ↪  Tipo::AmericanExpress;
        else return Tipo::JCB;
        break;
    case '4': return Tipo::VISA; break;
    case '5': return Tipo::Mastercard; break;
    case '6': return Tipo::Maestro; break;
    default: return Tipo::Otro; break;
    }
}

Tarjeta::~Tarjeta(){
    //Para poder eliminar una tarjeta, primero debemos de desvincularla de
    ↪  su titular
    if(Usuario* user = const_cast<Usuario*>(titular_))
    ↪  user->no_es_titular_de(*this);
    tarjetas_.erase(numero_); //eliminamos despues de desvincular
}

bool operator <(const Tarjeta& a,const Tarjeta& b){
    return a.numero() < b.numero();
}

std::ostream& operator <<(std::ostream& output, const Tarjeta& t)noexcept{
    //primero tenemos que concatenar el nombre y el apellido del titular
    Cadena nombre_apell = t.titular()->nombre() + " " +
    ↪  t.titular()->apellidos();
    //las tenemos que imprimir en mayusculas
    int i =0;
    while(nombre_apell[i]!='\0'){
        if(islower(nombre_apell[i]))
            nombre_apell[i]=toupper(nombre_apell[i]);
        i++;
    }
    output<<std::setw(2)<<std::setfill(' ')<<' '<<t.tipo()
    <<std::setw(2)<<std::setfill(' ')<<' '<<t.numero()<<"\n"
    <<std::setw(2)<<std::setfill(' ')<<' '<<nombre_apell<<"\n"
    <<std::setw(2)<<std::setfill(' ')<<' '<<"Caduca:
    ↪  "<<std::setfill('0')<<std::setw(2)<<t.caducidad().mes()
    <<"/"<<std::setw(2)<<(t.caducidad().anno() % 100)<<std::endl;
    return output;
```

```
}

std::ostream& operator <<(std::ostream&output, const Tarjeta::Tipo& tipo
↪  )noexcept{
    switch (tipo){
    case Tarjeta::Otro: output<<"Otro"<<std::endl; break;
    case Tarjeta::VISA: output<<"VISA"<<std::endl; break;
    case Tarjeta::Mastercard: output<<"Mastercard"<<std::endl;break;
    case Tarjeta::Maestro: output<<"Maestro"<<std::endl; break;
    case Tarjeta::JCB: output<<"JCB"<<std::endl; break;
    case Tarjeta::AmericanExpress: output<<"AmericanExpress"<<std::endl;
    ↪  break;
    default: output<<"Otra"<<std::endl;break;
    }
return output;
}
void Tarjeta::anula_titular(){
    (Usuario*&) titular_=nullptr;
    activa_=false;
}
```

3.3. Clase Usuario y Clave

Consta de dos clases Clave y Usuario las cuales ambas se definen en usuario.hpp y se implementan en usuario.cpp, como hemos comentado anteriormente.

3.3.1. Clase Clave

Esta clase se va a crear mediante una Cadena que alojará una contraseña que cifraremos. El constructor recibirá una cadena caracteres de bajo nivel `const char *` que contendrá la contraseña sin cifrar. Declaramos las clases de excepción **Clave::Incorrecta** que devolverá la razón por la que se ha llevado a cabo la excepción serán (CORTA y ERROR_CRYPT), esta clase de excepción devolverá el atributo correspondiente mediante el observador `razon()`.

Tendremos un observador llamado `clave()` que devuelve la contraseña cifrada almacenada.

Por último encontramos el método `verifica()` que recibirá como parámetro una cadena de caracteres de bajo nivel (contraseña sin cifrar) y devuelve un booleano si se corresponde con la contraseña almacenada, en este caso devuelve **true**, si no **false**.

3.3.2. Clase Usuario

La clase Usuario contrendrá estos atributos:

- 4 Cadenas que serán: identificador, nombre, apellido y dirección (por este orden).
- Una contraseña que será de tipo Clave.
- Un diccionario de tarjetas público (`std::map<Numero,Tarjeta*>`), que se tiene que llamar Tarjetas.
- El contenido del carrito, que será un diccionario de artículos junto con la cantidad de los mismos (`std::unordered_map<Articulo*,unsigned>`) que es público y se tiene que llamar Articulos.

El constructor recibirá 5 parámetros que son: identificador, nombre, apellido, dirección y clave. Este constructor debe de comprobar que el Usuario se crea correctamente:

- Comprueba que el identificador no esté repetido, si no se lanza la excepción **Usuario::Id_duplicado**, que devuelve el atributo a través del observador `idd()`.
- Un Usuario NO puede crearse por copia a través de otro (ni constructor ni operador de asignción por copia.)

La clase tendrá varios métodos observadores: `id()`, `nombre()`, `apellido()`, `direccion()`, `tarjetas()` y `compra()`. No devolvemos la clave.

Como la clase Usuario y Tarjeta están relacionadas entre sí, vamos a declarar el método `es_titular_de()` y `no_es_titular_de()` que recibe como parámetro la tarjeta y que se encargarán de crear y eliminar los enlaces de las clases.

A la hora de destruir una tarjeta primero la tenemos que desvincular de su titular, por eso llamará al método **Tarjeta::anula_titular()**

La clase Usuario y Articulo se relacionan mediante una asociación undireccional mediante el método `compra()`, que será una sobrecarga del observador, pero este va a recibir dos parámetros (Articulo y cantidad). Si la cantidad que se introduce es 0, se elimina, si es mayor que 0, se añade al carrito. Con este método sobrecargado podemos eliminar Articulos de la compra, pero también tenemos el método `vaciar_carro()` que eliminará todos los artículos del carro.

Mediante el observador `n_articulos()` devolvemos el número de articulos que contiene el carrito.

Vamos a sobrecargar el operdor de inserción en flujo `operator <<` para mostrar los datos del Usuario con el formato:

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

Por último, vamos a definir otro método `mostrar_carro()` (externa a la clase), para mostrar el contenido del carro en el flujo de salida con el formato:

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

3.3.3. Usuario.hpp

```
#ifndef USUARIO_HPP
#define USUARIO_HPP

//Inclusión de librerías
#include "../P1/cadena.hpp"
#include "tarjeta.hpp"
#include "articulo.hpp"
#include <random>
#include <map>
#include <unordered_map>
#include <unordered_set>
#include <iomanip>

//Declaraciones adelantadas
class Numero;
class Tarjeta;
/*-----Clase Clave-----*/
class Clave{
    public:
        typedef enum {CORTA,ERROR_CRYPT}Razon;
        Clave(const char*);

        //observador de la clase
        inline Cadena clave()const {return password_;}
        bool verifica(const char* )const;
        //Clase de la excepción
        class Incorrecta{
            Razon razon_;
            public:
                Incorrecta(const Razon& r):razon_(r){};
                const Razon& razon()const{return razon_;}
        };
    private:
        Cadena password_;
};

/*-----Clase Usuario-----*/
class Usuario{
    public:
        typedef std::map<Numero,Tarjeta*> Tarjetas;
        typedef std::unordered_map<Articulo*,unsigned int>Articulos;

        Usuario(const Cadena& ,const Cadena& , const Cadena& , const
        ↪  Cadena& , const Clave& );
        //eliminamos el ctor de copia y el operador de asignacion
```

```
        Usuario(const Usuario&)=delete;
        Usuario operator = (const Usuario&)=delete;
        //métodos observadores
        inline Cadena id()const noexcept{return identificador_;}
        inline Cadena nombre()const noexcept{return nombre_;}
        inline Cadena apellidos()const noexcept{return apellidos_;}
        inline Cadena direccion()const noexcept{return direccion_;}
       inline const Tarjetas& tarjetas()const noexcept{return tarjetas_;}
       inline const Articulos& compra()const noexcept{return articulos_;}

        //metodos de la asociacion con clase Tarjeta
        void es_titular_de(Tarjeta&) noexcept;
        void no_es_titular_de(Tarjeta&) noexcept;
        //métodos de la asociación unidireccional con Articulos
        void compra( Articulo&, size_t cantidad = 1) noexcept;
        inline void vaciar_carro() noexcept {articulos_.clear();}
        inline size_t n_articulos()const noexcept {return
        ↪  articulos_.size();}

        //destructor de la clase usuario
        ~Usuario();
        //Clase de la excepcion
        class Id_duplicado{
            Cadena id_;
            public:
                Id_duplicado(const Cadena& id):id_(id){};
                const Cadena& idd()const{return id_;}
        };
        friend std::ostream& operator <<(std::ostream& , const
        ↪  Usuario&)noexcept;
    private:
        const Cadena identificador_, nombre_,apellidos_,direccion_;
        Clave clave_;
        Tarjetas tarjetas_;
        Articulos articulos_;
       static std::unordered_set<Cadena> usuario_; //conjunto de usuarios
        ↪  no repetidos del programa
};
void mostrar_carro(std::ostream& , const Usuario&);
#endif // !USUARIO_HPP
```

3.3.4. Usuario.cpp

```
#include "usuario.hpp"
#include <unistd.h> //crypt()

//Inicialización del conjunto global de usuarios
std::unordered_set<Cadena>Usuario::usuario_;
/*---Clase Clave----*/
Clave:: Clave(const char* c){
    //Comprobamos tamaño de la clave
    if(strlen(c)<5){
        throw Incorrecta(Razon::CORTA);
    }
    else{
        //Encriptamos la clave nueva
        const char* valores
        ↪  ="abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ0123456789./";
        std::random_device random;
        std::uniform_int_distribution<>distribucion(0,63);
        char
        ↪  cifrado[2]={valores[distribucion(random)],valores[distribucion(random)]};
        //comprabamos que el cifrado sea correcto, si no excepción
        if(crypt(c,cifrado)){
            password_ = crypt(c,cifrado);
        }else{
            throw Incorrecta(Razon::ERROR_CRYPT);
        }
    }
}
bool Clave::verifica(const char* clave)const{
    //compara la encriptación de la clave que hemos metido concuerda con
    ↪  una de las claves
    return !(strcmp(crypt(clave, password_.operator const char *()),
                password_.operator const char *()));
}

/*-----Clase Usuario-----*/
Usuario::Usuario(const Cadena& id ,const Cadena& nom, const Cadena& apell
↪  ,const Cadena& dir ,const Clave& clave):

    ↪  identificador_(id),nombre_(nom),apellidos_(apell),direccion_(dir),clave_(clave)
    //Vamos a compromar si el usuario no existe en la lista de usuarios
    ↪  del programa
    if(!usuario_.insert(id).second)throw Usuario::Id_duplicado(id);
}
void Usuario::es_titular_de(Tarjeta& t) noexcept{
    //asosicamos el usuario con la tarjeta
```

```
    if(this==t.titular()){
        tarjetas_.insert(std::make_pair(t.numero(),&t));
    }
}

void Usuario::no_es_titular_de(Tarjeta& t) noexcept{
    //desvinculamos el usuario con la tarjeta
    t.anula_titular();
    tarjetas_.erase(t.numero());
}

void Usuario::compra( Articulo& art, size_t cantidad) noexcept{
    //si la cantidad del articulo es > 0 -> se añade; si no se elimina
    if(cantidad > 0)articulos_[&art]=cantidad;
    ↪  //articulos_.insert(std::make_pair(&art,cantidad));
    else articulos_.erase(&art);
}

Usuario::~Usuario(){
    //Para eliminar al usuario tenemos que desligar todas sus tarjetas y
    ↪  luego
    //eliminarlo
    for(auto i = tarjetas_.begin(); i!=tarjetas_.end();i++){
        i->second->anula_titular();
    }
    usuario_.erase(identificador_);
}

void mostrar_carro(std::ostream& output , const Usuario& user){
    output << "Carrito de compra de "<<user.id()<<" [Artículos:
    ↪  "<<user.n_articulos()<<"]\n"
           << "Cant. Artículo"<<std::endl;
    output <<std::setw(95)<<std::setfill('=')<<' '<<std::endl; //estética
    ↪  del formato
    //creamos una variable para tener control del numero de articulos del
    ↪  carro
    int n_art = user.n_articulos();
    while(n_art > 0){
        for(auto i=user.compra().begin();i!=user.compra().end();i++){
            output<<std::setw(4)<<std::setfill(' ')<< i->second
                <<" ["<<(*i->first).referencia()<<"] "<<"\""
                <<(*i->first).titulo()<<"\""<<", "
                <<(*i->first).f_publi().anno()<<". "
              <<std::fixed<<std::setprecision(2)<<(*i->first).precio()<<"
                ↪  €"<<std::endl;
            n_art--;
        }
```

```
    }
}
std::ostream& operator <<(std::ostream& output , const Usuario&
↪  user)noexcept{
    output << user.id() <<" ["<<user.clave_.clave().operator const char
    ↪  *()<<"] "<<user.nombre()<<" "
    <<user.apellidos()<<"\n"<<user.direccion()<<std::endl;
    output<<"Tarjetas: \n"<<std::endl;
    for(auto i = user.tarjetas().begin(); i!=user.tarjetas().end(); i++)
        output<< *i->second<<std::endl;
    return output;
}
```

4. Práctica 3

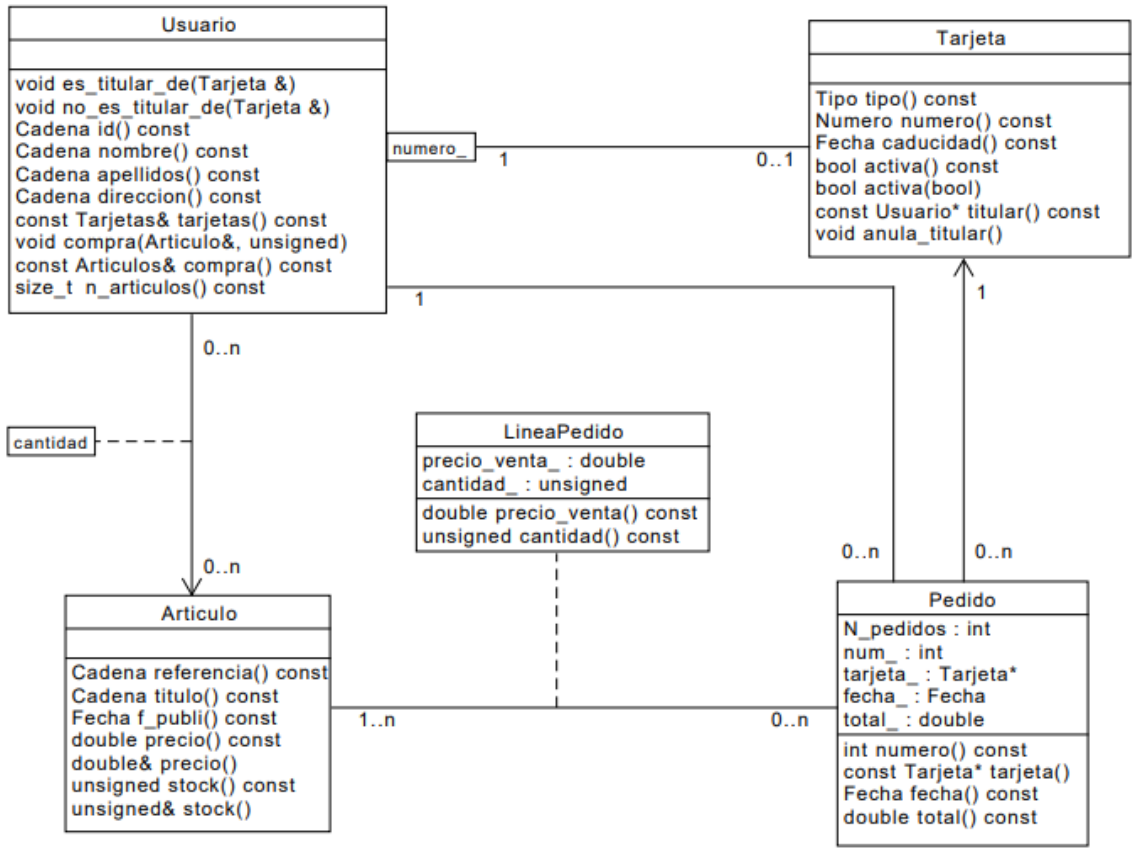

Figura 4.1: Diagrama de clases de la práctica 3.

En esta práctica haremos modificaciones en la clase Número para poder trabajar con predicados y objetos a funciones. Además crearemos nuevas clases como Pedido y LineaPedido y las clases de asociacion Pedido_Articulo, Usuario_Pedido.

4.1. Modificación de la clase Numero

En la práctica anterior definimos e implementamos dos métodos para quitar los espacios y obtener la longitud del Numero (ambos métodos privados de la clase).

Ahora vamos a reimplementar el método de quitar los espacios mediante el uso de dos algoritmos de la STL `remove_if()` y `find_if()`, donde ambos reciben un predicado.

Para el método `remove_if()` vamos a declarar un objeto a función llamado **Numero::EsBlanco** o bien hacer uso de una función Lambda para detectar si el caracter es un espacio o no mediante el método `isspace()`.

Por otro lado, para el método `find_if()` vamos a hacer uso de un objeto a función llamado **Numero::EsDigito** para detectar dígitos, y combínelo con el adaptador de negación unaria disponible en `<functional>` para detectar caracteres que no sean dígitos. Esta clase la tenemos que definir como miembro público de la clase Numero.

Todo esto se realiza en el constructor de la clase Numero por lo que los métodos anteriores (longitud y eliminar_espacios) desaparecen.

4.1.1. Numero.hpp

```
class Numero{
public:
  // tipos de excepciones
  typedef enum{ LONGITUD,DIGITOS,NO_VALIDO} Razon;
  Numero(const Cadena &);

  // operador de conversion a const char*
  inline operator const char *() const{
      return numero_.operator const char *();}
  // sobrecarga del operador < para comparar numeros
  friend bool operator<(const Numero &, const Numero &);
  // clase de la excepcion
  class Incorrecto{
    Razon razon_;
  public:
    Incorrecto(const Razon &r) : razon_(r){};
    const Razon &razon() const { return razon_; }
  };
  // Objeto a función para comprobar si son digitos
  class EsDigito : public std::unary_function<char, bool>{
  public:
    bool operator()(char caracter) const { return isdigit(caracter); }
  };
private:
  Cadena numero_;
};
```

4.1.2. Numero.cpp

```
Numero::Numero(const Cadena& numero){
  Cadena aux(numero);
  //hacemos uso de los predicados para los algoritmos remove_if, find_if
  //Eliminamos los espacios del numero
  auto quitaespacios = std::remove_if(aux.begin(),aux.end(),[](char c){
    return isspace(c);});
  //Modificamos el número fisicamente
  if(quitaespacios != aux.end()){
      aux = aux.substr(0, quitaespacios-aux.begin());
  }
  //negamos que sea un digito y buscamos caracteres que no sean digitos
  auto j = std::find_if(aux.begin(),aux.end(),not_fn(EsDigito()));

  if(j!=aux.end()) //si encuentra un caracter no numérico, excepción
      throw Incorrecto(Razon::DIGITOS);

  if(aux.length()>19 || aux.length() < 13 || aux.length() == 0)
      throw Incorrecto(Razon::LONGITUD);

  //comprobamos que sea correcto
  if(!luhn(aux))
      throw Incorrecto(Razon::NO_VALIDO);

  //actualizamos el numero sin espacios ni caracteres no numéricos
  numero_ = aux;
}
```

4.2. Clase Pedido

Vamos a crear una clase nueva llamada Pedido la cual va a contener información sobre los pedidos que van a poder realizar los Usuarios.

Como vemos en el diagrama contiene 5 atributos:

- Dos tipos enteros uno para el número total de pedidos (N_pedidos) y otro para el número del pedido que se va a realizar.
- Un tipo puntero a Tarjeta, que es la tarjeta con la que se realiza el pago.
- Un tipo Fecha que es la fecha del pedido.
- Un tipo double que es el precio total del pedido.

El constructor va recibir los siguientes parámetros:

- La asociación entre Usuario y Pedido (Usuario_Pedido).
- La asociación entre Articulo y Pedido (Articulo_Pedido).
- El Usuario que realiza el Pedido.
- La Tarjeta con la que se realiza el pago.
- La Fecha del pedido, por defecto es la del sistema.

Cuando se va a crear el pedido vamos a llamar a los métodos correspondientes para que se relacionen la clases entre sí, estas estarán en las clases de asociación previamente comentadas.

Vamos a crear la clase de excepción **Pedido::Vacio** para evitar así crear pedidos vacíos. Esta clase tiene un atributo que es un puntero al usuario que realiza el pedido, un constructor que recibe el usuario y un método observador usuario que lo devuelve.

También vamos a declarar la clase de excepción **Pedido::Impostor**, para poder comprobar que el Usuario que realiza el pedido es el mismo que el titular de la Tarjeta. Al igual que en la clase de excepción anterior, el atributo almacena un puntero al usuario, el constructor recibe el usuario y el observador lo devuelve.

Como los articulos tienen un stock definido, puede ser que algún articulo del carrito supere la cantidad de existencias por eso vamos a crear la clase de excepción **Pedido::SinStock** que lanzará la excepción y se eliminará el contenido del carrito. Esta clase de excepción consta de un atributo de tipo puntero a Articulo, un constructor que recibe el primer artículo del carrito sin existencias suficientes y un método observador articulo que lo devuelve.

No podemos realizar un pago de un pedido con una tarjeta que esté caducada por tanto, en el construtor de Pedido vamos a lanzar la excepción **Tarjeta::Caducada** cuando la fecha de caducidad de la tarjeta sea menor a la fecha del pedido. Si la Tarjeta está desactivada vamos a lanzar la excepción **Tarjeta::Desactivada**. Esta última es una clase de excepción vacía que habrá que añadir a la clase Tarjeta definida en la práctica anterior.

También contamos con varios métodos observadores: `numero()`, `tarjeta()`, `fecha()`, `total()` y `n_total_pedidos()` que devuelven los atributos correspondientes.

Por último, realizaremos una sobrecargar del operador de inserción en flujo (`operator <<`) para imprimir un Pedido por pantalla.
Mediante el formato:

```
Núm. pedido: 1
Fecha: jueves 10 de marzo de 2016
Pagado con: VISA n.º: 4539451203987356
Importe: 149,95 €
```

4.2.1. Pedido.hpp

```
#ifndef _PEDIDO_HPP_
#define _PEDIDO_HPP_

//Inclusión de librerias y cabeceras
#include "tarjeta.hpp"
#include "usuario.hpp"
#include "articulo.hpp"
#include "usuario-pedido.hpp"
#include "pedido-articulo.hpp"

//Declaraciones adelantadas
class Tarjeta;
class Pedido_Articulo;
class Usuario_Pedido;

class Pedido{
  public:
    Pedido(Usuario_Pedido& ,Pedido_Articulo& ,Usuario& ,const Tarjeta&
    ↪  ,const Fecha& f=Fecha());

    //Observadores de la clase
    int numero()const noexcept {return num_;}
    const Tarjeta* tarjeta()const noexcept{return tarjeta_;}
    const Fecha& fecha()const noexcept{return f_pedido_;}
    double total()const noexcept{return importe_Total_;}
    static int n_total_pedidos() noexcept{return n_pedidos_;}

    //clase de la excepcion vacio
    class Vacio{
        Usuario* user_;
      public:
```

```
        Vacio(Usuario* user):user_(user){}
        Usuario& usuario()const{return *user_;}
    };

    //clase de la excepción Impostor
    class Impostor{
        Usuario* user_;
      public:
        Impostor(Usuario* user):user_(user){}
        Usuario& usuario()const{return *user_;}
    };

    //clase de la excepción SinStock
    class SinStock{
        Articulo* articulo_;
      public:
        SinStock(Articulo* articulo):articulo_(articulo){}
        Articulo& articulo()const{return *articulo_;}
    };

  private:
    int num_;
    const Tarjeta* tarjeta_;
    Fecha f_pedido_;
    double importe_Total_;
    static int n_pedidos_;
};

//Sobrecarga del operador de flujo
std::ostream& operator <<(std::ostream& ,const Pedido& )noexcept;

#endif //!PEDIDO.HPP
```

4.2.2. Pedido.cpp

```
#include "pedido.hpp"

//Inicializamod la variable static
int Pedido::n_pedidos_ = 0;

//Constructor
Pedido::Pedido(Usuario_Pedido& up,Pedido_Articulo& pa,Usuario& u, const
↪  Tarjeta& t,const Fecha&
↪  f):num_(n_pedidos_+1),tarjeta_(&t),f_pedido_(f),importe_Total_(0.0){
  //Vamos a comprobar todas la excepciones del constructor de pedido
  //Si el usuario no tiene una compra realizada, el pedido está vacio
```

```
  if(u.compra().empty()) throw Pedido::Vacio(&u);

  //Si el usuario es otro al que realiza la compra, excepción
  if(t.titular()!=&u) throw Pedido::Impostor(&u);

  //Si la tarjeta está caducada o desactivada, excepción
  if(t.caducidad() < f_pedido_) throw Tarjeta::Caducada(t.caducidad());

  if(!t.activa()) throw Tarjeta::Desactivada();
  //Si no hay stock del articulo, excepción
  for(auto i : u.compra()){
    if(i.first->stock()<i.second){
      u.vaciar_carro();
      throw Pedido::SinStock(i.first);
    }
  }
  //Creamos la asociación entre las clases una vez que todo está correcto
  /*Asociación Usuario-Pedido*/
  up.asocia(*this,u);
  //actualizamos el importe del pedido y pedimos el articulo
  for(auto& i : u.compra()){
    //importe_Total_ += precio_articulo * cantidad del mismo.
    importe_Total_ += i.first->precio() * i.second;
    pa.pedir((*i.first),*this,i.first->precio(),i.second);

    //restamos el stock del articulo que se ha pedido
    i.first->stock() -= i.second; //sobrecarga del operador -=
  }
  //Aumentamos el numero de pedidos y vaciamos el carro del usuario
  u.vaciar_carro();
  n_pedidos_++;
}

//Insersión en flujo
std::ostream& operator <<(std::ostream& output , const Pedido&
↪  ped)noexcept{
  output<<"Núm. pedido: "<<ped.numero()<<std::endl
    <<"Fecha:\t"<<ped.fecha()<<std::endl
    <<"Pagado con: "<<ped.tarjeta()->tipo()<<" n.º:
    ↪  "<<ped.tarjeta()->numero()<<std::endl
    <<"Importe: "<<std::fixed<<std::setprecision(2)<<ped.total()<<"
    ↪  €"<<std::endl;
  return output;
}
```

4.3. Clase asociación Articulo_Pedido

Como vemos en el diagrama de clase la asociación entre las clases Articulo y Pedido tiene una clase de asociación llamada LineaPedido.

Esta clase la vamos a incluir en el fichero de cabecera Articulo_Pedido.hpp y la implementaremos en el fichero Articulo_Pedido.cpp.

4.3.1. Clase LineaPedido

La clase LineaPedido va a contener algunos atributos y métodos necesarios para que la relación bidireccional entre Articulo y Pedido se lleve a cabo. Cuenta con los atributos:

- Un tipo double que es el precio de la venta (precio_venta).
- Un entero sin signo que almacena la cantidad de ariculos (cantidad_).

Teniendo claro esto, un objeto de LineaPedido se crea mediante estos dos atributos que se le pasarán como parámetros, los cuales tendrán valores por omisión, (0.0) para el precio de venta y 1 para la cantidad.

Declararemos el constructor como `explicit` debido a que no se permiten conversiones implícitas.

La clase LineaPedido tendrá un par de métodos observadores llamados `precio_venta()` y `cantidad()`.

Finalmente, declararemos una sobrecarga del operador de inserción en flujo `operator <<` para imprimir un objeto LineaPedido. **Ejemplo: 35,20_€ TAB 3**.

4.3.2. Clase de asociación Articulo_Pedido

Como hemos comentado anteriormente esta clase representará la relación bidireccional entre las clases Articulo y Pedido, por tanto, va a contener todos los métodos imprescindibles para que dicha relación se lleve a cabo.

Si nos fijamos en el diagrama de clases, vemos que la relación entre ambas clases es 1..N - M, por tanto, la clase va a tener dos atributos que serán dos diccionarios:

- Relación Articulo - Pedido: Este diccionario será del tipo `std::map<Pedido*,ItemsPedido,OrdenaPedidos>`, donde **ItemsPedido** es un método público de la clase definido como otro diccionario: `std::map<Articulo*,LineaPedido,OrdenaArticulos>`. **OrdenaPedidos** y **OrdenaArticulos** serán dos tipos públicos definidos como dos clases anidadas. Se trata de dos clases de objetos función para ordenar ascendentemente los pedidos y artículos por número y referencia, respectivamente, dentro de los diccionarios.
- Relación Pedido - Articulo: Se representará por un diccionario del tipo `std::map<Articulo*,Pedidos,OrdenaArticulos>`, donde Pedidos será un tipo público definido como `std::map<Pedido*,LineaPedido,OrdenaPedidos>`.

NO es necesario definir los constructores.

Como tenemos varios diccionarios para la creación de estos enlaces vamos a declarar varios métodos para insertar los elementos en los diccionarios correspondientes, para ello contamos con el método `pedir()` que recibe por parámetro el Articulo, Pedido, el precio y la cantidad (por defecto es 1). Tenemos que realizar una sobrecarga de dicho método invirtiendo el orden del Articulo y Pedido, este método es de tipo void.

Contamos con un observador llamado `detalle()` que dado un Pedido, nos devuelve el diccionario de los articulos de un pedido, es decir **ItemsPedido** mediante una referencia constante, ya que la multiplicidad es "1..N".

También contamos con un observador que nos devuelve los pedidos **Pedidos**, que ha realizado un Articulo, este método se llamará `ventas()`. Aquí devolvemos una copia debido a que la multiplicidad es "N" y por tanto, pueden haber objetos que no estén instanciados.

Realizaremos dos sobrecargas del operador de inserción en flujo `operator <<` para los tipos de datos **Pedido_Articulo::ItemsPedido** y **Pedido_Articulo::Pedidos**. El primero mostrará los detalles del pedido y, además, el importe total. El segundo mostrará precio, cantidad y fecha de cada venta, así como el importe total de las ventas del artículo y el número de ejemplares vendidos.

Ejemplo de salida de los detalles de un pedido:

```
PVP   Cantidad  Artículo
================================================================
35,20 € 2 [100] "Programación Orientada a Objetos"
29,95 € 1 [110] "Fundamentos de C++"
================================================================
Total 100,35 €
```

Ejemplo de salida de pedidos:

```
[Pedidos: 3]
================================================================
PVP   Cantidad  Fecha de venta
================================================================
39,95 € 2 miércoles 19 de abril de 2017
34,90 € 1 jueves 20 de abril de 2017
34,90 € 1 lunes 5 de abril de 2010
================================================================
149,70 € 4
```

También tendremos un método llamado `mostrarDetallePedidos()` imprimirá en el flujo de salida proporcionado el detalle de todos los pedidos realizados hasta la fecha, así como el importe de todas las ventas.

Ejemplo de salida de mostrarDetallePedidos:

```
Pedido núm. 1
Cliente: lucas Fecha: viernes 10 de marzo de 2017

detalle de ese pedido, como en ejemplo anterior
Resto de los pedidos ...

TOTAL VENTAS 981,60 €
```

Por último, un método llamado `mostrarVentasArticulos()` visualizará en el flujo de salida proporcionado todas las ventas agrupadas por artículos.

Ejemplo de salida de mostrarVentasArticulos:

```
Ventas de [110] "Fundamentos de C++"
pedidos de ese artículo, como en ejemplo anterior
Resto de los artículos vendidos ...
```

4.3.3. Articulo_Pedido.hpp

```
#ifndef PEDIDO_ARTICULO_HPP
#define PEDIDO_ARTICULO_HPP

//Inclusión de librerias y cabeceras
#include "pedido.hpp"
#include "articulo.hpp"
#include <map>

//Contendrá LineaPedido
class LineaPedido{
  public:
    explicit LineaPedido(double precio = 0.0, unsigned cant=1)
        :precio_venta_(precio),cantidad_venta_(cant){}
    //Métodos observadores de la clase
    double precio_venta()const noexcept{return precio_venta_;}
    unsigned cantidad()const noexcept{return cantidad_venta_;}
  private:
    double precio_venta_;
    unsigned cantidad_venta_;
};
//Operador de inserción en flujo de LineaPedido
std::ostream& operator <<(std::ostream& , const LineaPedido&)noexcept;

//Clase de asociación entre Pedido - Articulo
class Pedido_Articulo{
  public:
  struct OrdenaArticulos{
    bool operator () (const Articulo*, const Articulo* )const;
  };

  struct OrdenaPedidos{
    bool operator()(const Pedido* , const Pedido* )const;
  };
    //Diccionarios de la clase Pedido - Articulo
    typedef std::map<Articulo*,LineaPedido,OrdenaArticulos>ItemsPedido;
    typedef std::map<Pedido*,ItemsPedido,OrdenaPedidos> PedidoArticulo;
    //Diccionarios de la clase Articulo - Pedido
    typedef std::map<Pedido*,LineaPedido,OrdenaPedidos>Pedidos;
    typedef std::map<Articulo*,Pedidos,OrdenaArticulos> ArticuloPedido;
    //Creacion de enlaces
    void pedir(Pedido&,Articulo&,double,unsigned cantidad = 1); //directa
    void pedir(Articulo&,Pedido&,double,unsigned cantidad = 1); //inversa
```

```
    //observador que devuelve los articulos de un pedido
    const ItemsPedido& detalle(Pedido& )const;
    //observador que devuelve los pedidos de un articulo
    Pedidos ventas(Articulo& )const;
    //Métodos que imprimen detalles de los pedidos y ventas
    std::ostream& mostrarDetallePedidos(std::ostream&)const noexcept;
    std::ostream& mostrarVentasArticulos(std::ostream&)const noexcept;
  private:
    //Diccionarios de la clase
    PedidoArticulo pedido_articulo_; //directa
    ArticuloPedido articulo_pedido_; //inversa
};

//Sobrecarga de los operadores de inserción en flujo
std::ostream& operator << (std::ostream& , const
↪  Pedido_Articulo::ItemsPedido&)noexcept;
std::ostream& operator << (std::ostream& , const
↪  Pedido_Articulo::Pedidos&) noexcept;
#endif // !PEDIDO_ARTICULO_HPP
```

4.3.4. Articulo_Pedido.cpp

```
//Implementación de la clase LineaPedido
std::ostream& operator <<(std::ostream& output , const LineaPedido&
↪  lp)noexcept{
  return output << std::fixed<<std::setprecision(2)<<lp.precio_venta()<<"
  ↪  €\t" <<lp.cantidad();
}

//Implementación de la clase Pedido_Articulo
bool Pedido_Articulo::OrdenaArticulos::operator()(const Articulo* a1,
↪  const Articulo* a2) const{ return (a1->referencia() <
↪  a2->referencia()); }

bool Pedido_Articulo::OrdenaPedidos::operator()(const Pedido* p1, const
↪  Pedido* p2)const{ return (p1->numero() < p2->numero()); }

void Pedido_Articulo::pedir(Pedido& p, Articulo& a, double precio,
↪  unsigned cant){
    //Creamos una LineaPedido nueva
    LineaPedido lp(precio,cant);
    //creamos los enlaces directa e inversamente
    pedido_articulo_[&p].insert(std::make_pair(&a,lp));
    articulo_pedido_[&a].insert(std::make_pair(&p,lp));
}
```

```
void Pedido_Articulo::pedir(Articulo& a,Pedido& p,double precio,unsigned
↪  cant){
    pedir(p,a,precio,cant);//Delegamos en el método anterior
}

const Pedido_Articulo::ItemsPedido& Pedido_Articulo::detalle(Pedido&
↪  p)const{
  //Como siempre habrá como minimo un articulo en el pedido, devolvemos
  ↪  directamente los articulos de dicho pedido.
  return pedido_articulo_.find(&p)->second;
}

Pedido_Articulo::Pedidos Pedido_Articulo::ventas(Articulo& a)const{
    //buscamos el articulo, para ver si tiene un pedido asignado o no
    auto i = articulo_pedido_.find(&a);
    if(i!=articulo_pedido_.end())return i->second; //devolvemos pedidos
    else{//no hay pedido asignado, devolvemos conjunto vacio.
        Pedido_Articulo::Pedidos PedidoVacio;
        return PedidoVacio;
    }
}

//Implementación de los métodos de inserción en flujo
std::ostream& operator << (std::ostream& output,const
↪  Pedido_Articulo::ItemsPedido& IP)noexcept{
    //nos creamos una variable local para obtener el precio total
    double total = 0.;
    output <<"PVP\t Cantidad\t Artículo"<<std::endl
    << std::setw(65)<<std::setfill('=')<<' '<<std::endl; //formato
    //insertamos el pedido
    for(auto& i :IP){
        total += i.second.precio_venta()*i.second.cantidad();
     output<<std::fixed<<std::setprecision(2)<<i.second.precio_venta()<<"
        ↪  € "
        <<i.second.cantidad()<<"\t"
        <<"["<<i.first->referencia()<<"]
        ↪  \""<<i.first->titulo()<<"\""<<std::endl;
    }
    output <<  std::setw(65)<<std::setfill('=')<<' '<<std::endl
    <<"Total "<<std::fixed<<std::setprecision(2)<<total<<" €";
    return output; //devolvemos el flujo de salida
}
```

```
std::ostream& operator <<(std::ostream& output,const
↪  Pedido_Articulo::Pedidos& Ped)noexcept{
    //Nos creamos una variable local para obtener el precio total y el
    ↪  numero de pedidos
    double total = 0.;
    unsigned npedido =1;
    output<<"[Pedidos: "<<npedido<<"]"<<std::endl
      <<std::setw(65)<<std::setfill('=')<<" "<<std::endl//formato
      <<"PVP \t Cantidad \t Fecha de venta"<<std::endl
      <<std::setw(65)<<std::setfill('=')<<" "<<std::endl;//formato
    for(auto& i : Ped){
    //actualizamos el numero de pedido y el total
      total+=i.second.precio_venta()*i.second.cantidad();
      npedido++;
      output<<i.second.precio_venta()<<" €\t"<<i.second.cantidad()<<"\t"
      <<i.first->fecha()<<std::endl;
    }
    output<<std::setw(65)<<std::setfill('=')<<" "<<std::endl//formato
          <<std::fixed<<std::setprecision(2)<<total<<" €\t
          ↪  "<<npedido<<std::endl;
    return output;
}
std::ostream& Pedido_Articulo::mostrarDetallePedidos(std::ostream&
↪  output)const noexcept{
    //Creamos variables locales para alamecenar el numero de pedido y el
    ↪  total de ventas
    double total=0.;
    unsigned npedido=0;
    for(auto& i : pedido_articulo_){
      npedido++;
      output<<"\nPedido núm. "<<npedido<<std::endl
            <<"Cliente: "<< *i.first->tarjeta()->titular()
            <<detalle(*i.first)<<std::endl;
      total += i.first->total();
    }
    output<<"TOTAL VENTAS\t "<<std::fixed<<std::setprecision(2)<<total<<"
    ↪  €"<<std::endl;
    return output;
}
std::ostream& Pedido_Articulo::mostrarVentasArticulos(std::ostream&
↪  output)const noexcept{
  for(auto& i : articulo_pedido_){
    output<<"Ventas de ["<<i.first->referencia()<<"]
    ↪  "<<"\""<<i.first->titulo()<<"\"\n"<<ventas(*i.first)<<std::endl;
  }
  return output;
}
```

4.4. Clase de asociación Usuario_Pedido

Si nos fijamos en el diagrama de clases de la práctica vemos que la relación entre las clases Usuario y Pedido es una relación de asociación pero sin atributos de enlace cuya multiplicidad es 1 - N.

Esta clase de asociación va a almacenar dos atributos:

- Un diccionario de puntero Usuario a un conjunto de elementos Pedidos (set):

```
typedef std::set<Pedido*>Pedidos;
typedef std::map<Usuario*,Pedidos>Usuario_Pedido;
```

- Un diccionario de puntero Pedido a puntero Usuario.

```
typedef std::map<Pedido*,Usuario*>Pedido_Usuario;
```

Vamos a contar con dos métodos llamados `asocia()` que recibe por parámetro una referencia a Usuario y a Pedido. Debemos de realizar una sobrecarga del método intercambiando el orden de los parámetros.

Por otro lado, tenemos dos métodos observadores:

- Método llamado `pedidos()` que recibe como parámetro una referencia a Usuario y devuelve los Pedidos del mismo. Como puede haber un Usuario sin pedidos devolvemos una copia.
- Método llamado `cliente()` que dado una referencia de un Pedido, este devuelve la dirección de memoria del Usuario que ha hecho el pedido.

4.4.1. Usuario_Pedido.hpp

```
#ifndef USUARIO_PEDIDO_HPP
#define USUARIO_PEDIDO_HPP
// Incluimos cabeceras
#include "usuario.hpp"
#include "pedido.hpp"

class Usuario_Pedido{
public:
  typedef std::set<Pedido *> Pedidos;
  typedef std::map<Usuario *, Pedidos> UsuarioPedidos;
  typedef std::map<Pedido *, Usuario *> PedidoUsuario;

  void asocia(Usuario &, Pedido &) noexcept;
  void asocia(Pedido &, Usuario &) noexcept;
```

```
  // pedidos recibe un Usuario y devuelve los pedidos que éste ha
  ↪  realizado.
  Pedidos pedidos(Usuario &) const noexcept;
  // cliente recibe un Pedido y devuelve la dirección de memoria del
  ↪  usuario que ha hecho el pedido.
  Usuario *cliente(Pedido &) const noexcept;

private:
  UsuarioPedidos usuariopedidos_;
  PedidoUsuario pedidousuario_;
};

#endif //! USUARIO_PEDIDO.HPP
```

4.4.2. Usuario_Pedido.cpp

```
// Implementación de los métodos de la clase de asociación
void Usuario_Pedido::asocia(Usuario &u, Pedido &p) noexcept{
  usuariopedidos_[&u].insert(&p);
  pedidousuario_[&p] = &u;
}

void Usuario_Pedido::asocia(Pedido &p, Usuario &u) noexcept{
  // delegamos en el método anterior
  asocia(u, p);
}

Usuario_Pedido::Pedidos Usuario_Pedido::pedidos(Usuario &u) const
↪  noexcept{
  // comprobamos que existe el usuario
  auto i = usuariopedidos_.find(&u);
  if (i != usuariopedidos_.end())
    return i->second;
  else{
    Usuario_Pedido::Pedidos pedidovacio;
    return pedidovacio;
  }
}

Usuario *Usuario_Pedido::cliente(Pedido &p) const noexcept{
  // devolvemos el usuario de dicho pedido
  return pedidousuario_.find(&p)->second;
}
```

5. Práctica 4

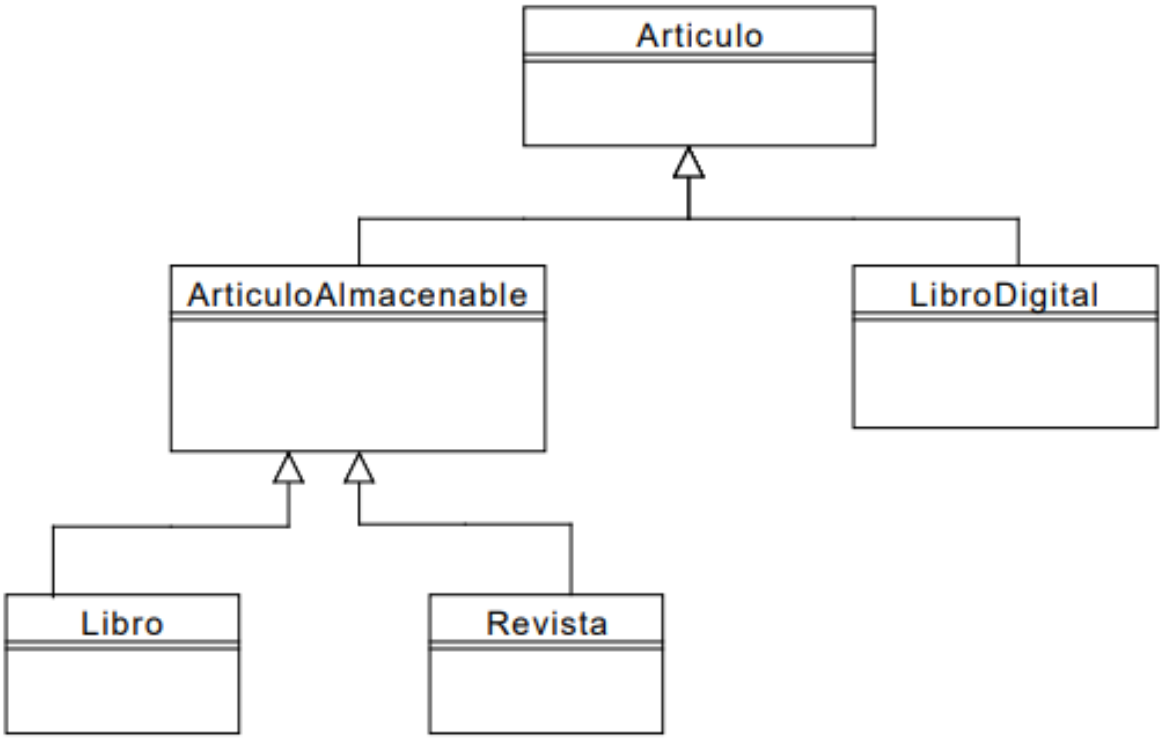

Figura 5.1: Diagrama de clases de la práctica 4.

Esta es la última práctica, y se corresponde con el tema de polimorismo.

Como vemos en el diagrama la clase Articulo se especializa en dos subclases, debido a esto vamos a tener que realizar ciertas modificaciones en dicha clase.

5.1. Modificación de la clase Articulo

La clase Articulo la vamos a declarar ahora como una clase abstracta, es decir, no se pueden crear objetos de ella ya que tiene un método definido como virtual puro, ese método será:

```
virtual void impresion_especifica(std::ostream&)const = 0;
```

El constructor de Articulo también sufre cambios, siendo ahora el primer parámetro del mismo una referencia constante a un conjunto de Autores de una nueva clase llamada `Autor`.

El atributo `stock_` deja de pertencer a la clase Articulo, y ahora pertenece a la clase ArticuloAlmacenable, por tanto, los observadores del atributo también se pasan a la clase ArticuloAlmacenable, el resto de los parámetros son referencia, titulo, fecha de publicación y el precio del articulo.

Vamos a definir una clase de excepcion **Articulo::Autores_vacios**, ya que un Articulo tiene que tener como mínimo un Autor.

El método `impresion_especifica()` ('comentado previamente'), realizará modificaciones en el método `mostrar_carro()` por tanto, tenemos que reescribir esa función para que no dependa del operador de inserción de Articulo y conserve el formato de salida original.

El formato de salida del método será:

- Libros:

```
[103] "Brácula, el no vivo", de Echo, Rey. 1902. 39,95 €
455 págs., 4 unidades.
```

- Revistas:

```
[701] "Byter", de Verde. 2022. 15,70 €
Número: 101, Periodicidad: 30 días.
Próximo número a partir de: miércoles 14 de septiembre de 2022.
```

- LibrosDigitales:

```
[036] "55 días en Pasquín", de Fuertes. 2007. 12,00 €
A la venta hasta el martes 30 de septiembre de 2008.
```

5.2. Clase Autor

Como hemos comentado en la modificación de la clase Articulo, esta recibe como parámetro una referencia constante de un Autor.

Esta clase Autor contiene los datos de los autores (nombre, apellido y dirección), donde no contemplamos ninguna excepción.

Además contendrá observadores correspondientes a los atributos llamados `nombre()`, `apellidos()` y `direccion()`, que devuelven una referencia constante de la Cadena.

5.3. La subclase ArticuloAlmacenable

Esta clase al igual que Articulo será una clasa abstracta ya que también contiene el método:

```
virtual void impresion_especifica(std::ostream&)const = 0;
```

Esta clase almacena el atributo `stock` que previamente pertenecía a la clase Articulo, ya que como bien dice su nombre, un ArticuloAlmacenable debe de ser un Articulo que se pueda almacenar, es decir, que tiene un stock.

Podemos consultar el stock de un ArticuloAlmacenable mediante el observador `stock()` y su sobrecarga para modificarlo.

Como esta clase es una especialización de Articulo, ArticuloAlmacenable en su constructor tendrá que recibir todos los atributos con los que se inicializa un Articulo más el stock propio del ArticuloAlmacenable.

5.4. La subclase LibroDigital

Es una clase derivada de la clase Articulo, debido a que no tenemos que almacenar el número de ejemplares de los mismos.

Tenemos que tener en cuenta la fecha de expiración(`f_expir()`) donde los ebooks dejan de venderse.

Esta fecha no se puede modificar una vez se cree el objeto y se va a crear con los mismos parámetros que la clase artículo más el atributo propio de la fecha de expiración.

Si la fecha de expiración es menor a la del Pedido no se podrá comprar. Para que esto se cumpla tendremos que modificar el constructor de la clase Pedido si un *Usuario* ha introducido un *LibroDigital* expirado, es decir, este no se añadirá al pedido.

Si el Pedido que vacío por este motivo, se lanzará la excepción **Pedido::Vacio**.

5.5. La subclase Libro

Esta clase es una derivación de la clase de ArticuloAlmacenable, o sea, que tiene también un stock asociado, además que tiene como atributo propio el número de paginas de cada objeto libro.

Podemos consultar el número de páginas (que es no modificable) mediante el observador `_pag()`.

El constructor de Libro tiene los mismo parámetros que ArticuloAlmacenable pero añadiendole el atributo propio del número de páginas, que por defecto es 0.

5.6. La subclase Revista

También es una clase que contiene un stock, por tanto, es derivada de la clase ArticuloAlmacenable.

Tenemos un numero con el observador `numero()` y la periodicidad con el observador `periodicida()`. No se pueden modificar ninguno de los dos cuando se cree el objeto.

5.7. Modificación clase Articulo.hpp

```
  #ifndef ARTICULO_HPP
#define ARTICULO_HPP

// Inclusión de librerías
#include "../P1/fecha.hpp"
#include "../P1/cadena.hpp"
#include <iostream>
#include <iomanip>
#include <locale.h>
#include <set>

/*-----Clase Autor de la P4-----*/
class Autor{
public:
  // No definimos la relación con Articulo aqui ya que no
  // nos hace falta.
  Autor(const Cadena &nombre, const Cadena &apellido, const Cadena
  ↪  &direccion)
      : nom_(nombre), apell_(apellido), dir_(direccion) {}
  // Consultores de la clase
  inline const Cadena &nombre() const noexcept { return nom_; }
  inline const Cadena &apellidos() const noexcept { return apell_; }
  inline const Cadena &direccion() const noexcept { return dir_; }

private:
  Cadena nom_, apell_, dir_;
};

/*-----Clase Aticulo de la P4-----*/
class Articulo{
public:
  /*-----Relación entre Articulo - Autor-----*/
  // podemos definirlo como una asociación o agregación, que cada Articulo
  ↪  contendrá
  // un conjunto de Autores.
```

```
  typedef std::set<Autor *> Autores;

  // Como un Articulo puede tener como mínimo 1 autor y además sin autor
  ↪  no hay articulo
  Articulo(const Autores &autores, Cadena referencia, Cadena titulo,
           Fecha f_publi, double precio);

  // Observadores de la clase
  inline const Cadena &referencia() const noexcept { return referencia_; }
  inline const Cadena &titulo() const noexcept { return titulo_; }
  inline const Fecha &f_publi() const noexcept { return fpubli_; }
  inline double precio() const noexcept { return precio_; }
  inline double &precio() noexcept { return precio_; }
  // Desaparece en esta práctica
  //  inline unsigned stock()const noexcept{return stock_;}
  //  inline unsigned& stock()noexcept{return stock_;}
  inline const Autores &autores() const noexcept { return autores_; }
  // Exepcion de la clase
  class Autores_vacios{
  };

  // Nuevo operador de inserción en flujo
  virtual void impresion_especifica(std::ostream &) const = 0;

  // Como esta clase se va a especializar en otras su destructor debe de
  ↪  ser
  // virtual para que puedan llamarse a los destructores de las clases
  ↪  derivadas
  virtual ~Articulo() = default;

private:
  Autores autores_; // relación de las clases Articulo - Autor
  const Cadena referencia_, titulo_;
  const Fecha fpubli_;
  double precio_;
  // unsigned stock_; Desaparece en esta práctica
};

// Operador de inserción en flujo
std::ostream &operator<<(std::ostream &, const Articulo &) noexcept;

/*-----Clase Derivada ArticuloAlamacenable-----*/
class ArticuloAlmacenable : public Articulo{
  public:
    ArticuloAlmacenable(const Autores &autores, Cadena referencia, Cadena
    ↪  titulo,Fecha f_publi, double precio, unsigned stock = 0)
```

```
        : Articulo(autores, referencia, titulo, f_publi, precio),
        ↪  stock_(stock) {}

    // Observadores de la clase
    inline unsigned stock() const noexcept { return stock_; }
    inline unsigned &stock() noexcept { return stock_; }

  protected: // se pone protected para que las clases derivadas de esta
  ↪  puedan hacer uso del atributo
    // Atributo de la clase Articulo que lo define
    unsigned stock_;
};

/*-----Clase Derivada Libro-----*/
class Libro final : public ArticuloAlmacenable{
  public:
    Libro(const Autores &autores, Cadena referencia, Cadena titulo,
          Fecha f_publi, double precio, unsigned paginas = 0, unsigned
          ↪  stock = 0) : ArticuloAlmacenable(autores, referencia,
          ↪  titulo, f_publi, precio, stock), n_pag_(paginas) {}
    // Observadores de la clase
    inline const unsigned &n_pag() const noexcept { return n_pag_; }
    // Método virtual heredado
    void impresion_especifica(std::ostream &) const override final;

  private:
    // Atributo de la clase que lo define
    const unsigned n_pag_;
};

/*-----Clase Derivada Revista-----*/
class Revista final : public ArticuloAlmacenable{
public:
    Revista(const Autores &autores, Cadena referencia, Cadena titulo,
            Fecha f_publi, double precio, const unsigned numero, const
            ↪  unsigned f, unsigned stock = 0) :
            ↪  ArticuloAlmacenable(autores, referencia, titulo, f_publi,
            ↪  precio, stock), numero_(numero), periodicidad_(f) {}
    // Observadores de la clase
    inline unsigned numero() const noexcept { return numero_; }
    inline unsigned periodicidad() const noexcept { return periodicidad_;
    ↪  }
    // Método virtual heredado
    void impresion_especifica(std::ostream &) const override final;

  private:
    const unsigned numero_, periodicidad_;
```

```
};

/*-----Clase Derivada LibroDigital-----*/
class LibroDigital final : public Articulo{
  public:
    LibroDigital(const Autores &autores, Cadena referencia, Cadena titulo,
                 Fecha f_publi, double precio, const Fecha &f_expir) :
                  ↪  Articulo(autores, referencia, titulo, f_publi,
                  ↪  precio), f_expir_(f_expir) {}
    // Observador de la clase
    inline const Fecha &f_expir() const noexcept { return f_expir_; }
    // Método virtual heredado
    void impresion_especifica(std::ostream &) const override final;

  private:
    const Fecha f_expir_;
};

#endif // !ARTICULO_HPP
```

5.8. Modificación clase Articulo.cpp

```
#include "articulo.hpp"
/*-----Implementación de la clase Articulo-----*/
Articulo::Articulo(const Autores &autor, Cadena referencia, Cadena
↪  titulo, Fecha f_publi, double precio) :
   autores_(autor), referencia_(referencia), titulo_(titulo),
    ↪  fpubli_(f_publi), precio_(precio){
  // Lo unico que tenemos que comprobar que dicho articulo tenga como
   ↪  mínimo un autor
  if (autores_.empty()) throw Autores_vacios();
}

// Operador de inserción en flujo
std::ostream &operator<<(std::ostream &output, const Articulo &art)
↪  noexcept{
  std::locale::global(std::locale(""));
  output << "[" << art.referencia() << "] " << "\"" << art.titulo() <<
   ↪  "\"" << ", de ";
  // recorremos el conjunto de autores del articulo
  auto i = art.autores().begin();
  output << (*i)->apellidos();
  for (i++; i != art.autores().end(); i++){
    output << ", " << (*i)->apellidos();
  }
```

```
  output << ". " << art.f_publi().anno() << ". "
         << std::fixed << std::setprecision(2) << art.precio() << " €" <<
          ↪  std::endl
         << "\t";
  // llamamos al método heredado.
  art.impresion_especifica(output);
  return output;
}
/*-----Implementación del Método virtual-----*/
void Libro::impresion_especifica(std::ostream &output) const{
  output << n_pag_ << " págs., " << stock_ << " unidades.";
}
void Revista::impresion_especifica(std::ostream &output) const{
  output << "Número: " << numero_ << ", Periodicidad: " << periodicidad_
  ↪  << " días."
    << "\n"<< "\t"<< "Próximo número a partir de: " << Fecha(f_publi() +
    ↪  periodicidad_) << ".";
}
void LibroDigital::impresion_especifica(std::ostream &output) const{
  output << "A la venta hasta el " << f_expir_ << ".";
}
```

5.9. Modificación clase Pedido.cpp

Como la clase Articulo ya no tiene más el atrbuto stock, a la hora de comprarlo en el constructor de la clase Pedido, tenemos que realizar una conversión explicita de Articulo ya sea a ArticuloAlmacenable.

También tendremos que comprobar la fecha de expiración de los LibrosDigitales, por lo que también tendremos que realizar una conversión explicita de Articulo a LibroDigital.

Tenemos que hacer esto debido a que la clase Pedido trabaja con un conjunto de Articulos.

```
Pedido::Pedido(Usuario_Pedido &up, Pedido_Articulo &pa, Usuario &u,const
↪  Tarjeta &t, const Fecha &f)
  : num_(n_pedidos_ + 1), tarjeta_(&t), f_pedido_(f), importe_Total_(0.0){
  // Vamos a comprobar todas la excepciones del constructor de pedido
  // Si el usuario no tiene una compra realizada, el pedido está vacio
  if (u.compra().empty()) throw Pedido::Vacio(&u);

  // Si el usuario es otro al que realiza la compra, excepción
  if (t.titular() != &u)  throw Pedido::Impostor(&u);

  // Si la tarjeta está caducada o desactivada, excepción
  if (t.caducidad() < f_pedido_)  throw Tarjeta::Caducada(t.caducidad());
```

```
  if (!t.activa())  throw Tarjeta::Desactivada();
  // Si no hay stock del articulo, excepcion.
  // Hacemos la conversión primero y luego comprobamos
  Usuario::Articulos carrito = u.compra();
  for (auto i : carrito){ // carrito del usuario
    if (ArticuloAlmacenable *AA = dynamic_cast<ArticuloAlmacenable
    ↪  *>(i.first)){ // hacemos conversion
      // comprobamos el stock
      if (AA->stock() < i.second){
        u.vaciar_carro();
        throw Pedido::SinStock(AA);
      }
    }
    else if (LibroDigital *LD = dynamic_cast<LibroDigital *>(i.first)){
      // Si la fecha de expiración es < a la fecha del pedido, se añade
      ↪  con cantidad 0
      if (LD->f_expir() < Fecha()){ // Ha expirado
        u.compra(*LD, 0); // Añadimos al pedido dicho articulo pero con la
        ↪  cantidad a 0
      }
    }
    else throw std::logic_error("Pedido::Pedido - Tipo de articulo no
    ↪  conocido");
  }
  // Comprobamos que no se quede vacio el Pedido
  if (u.compra().empty()) throw Pedido::Vacio(&u);
  // Realizamos la asociación Usuario - Pedido
  up.asocia(*this, u);

  // Vamos a pedir el articulo
  for (auto &i : u.compra()){
    importe_Total_ += i.first->precio() * i.second;
    pa.pedir(*this, *(i.first), i.first->precio(), i.second);

    // Ahora para restar el stock de dicho articulo hacemos lo mismo que
    ↪  arriba
    if (ArticuloAlmacenable *AA = dynamic_cast<ArticuloAlmacenable
    ↪  *>(i.first)){
      AA->stock() -= i.second;
    }
  }
  // Aumentamos el numero de pedidos y vaciamos el carro del usuario
  u.vaciar_carro();
  n_pedidos_++;
}
```