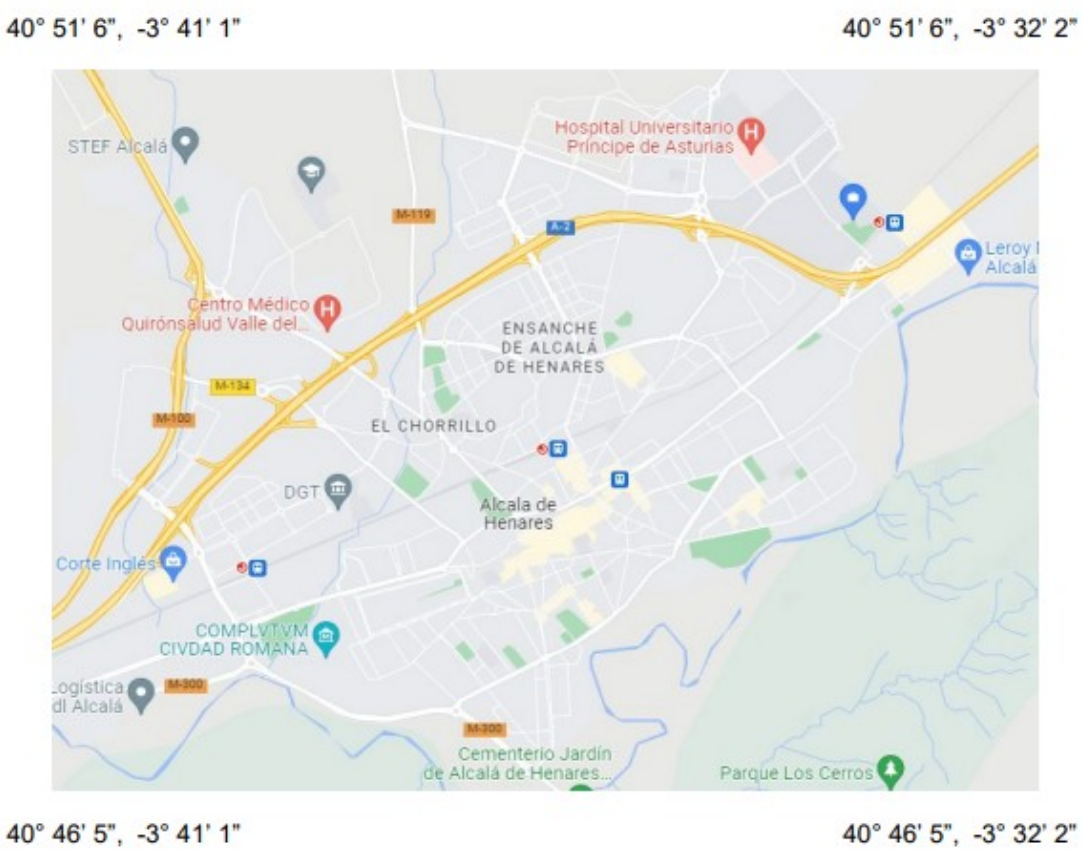
PECL 1- Estructuras de Datos

Simulación del funcionamiento de un servicio de paquetería



Creado por:

- Christian Noguerales Alburquerque
- Emilio Macías Do Santos

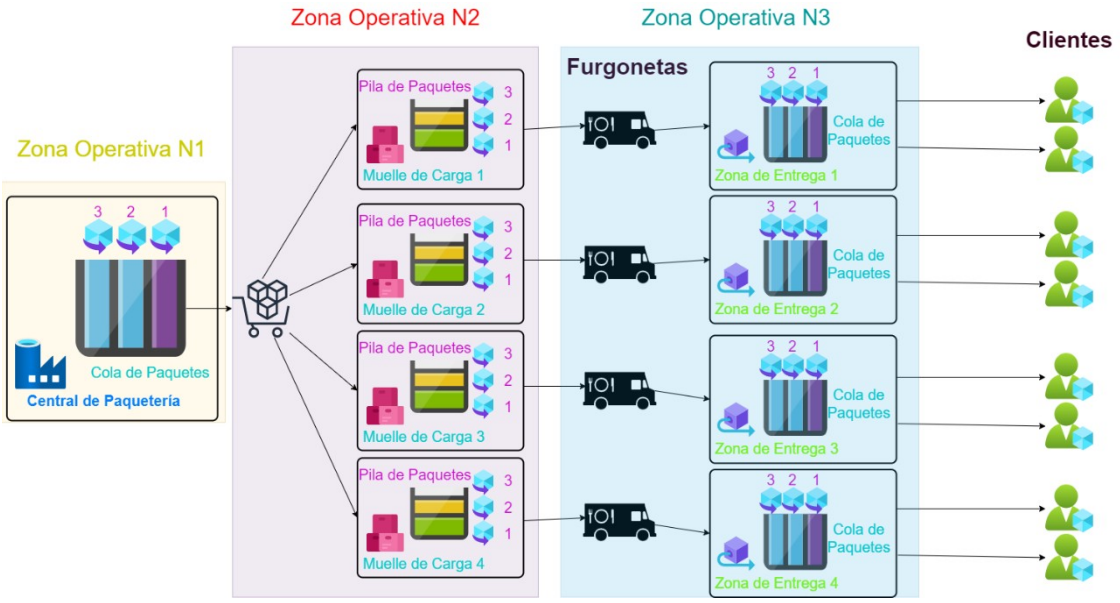
PECL 1- Estructuras de Datos. Simulación del funcionamiento de un servicio de paquetería.

Contenido

Portada de la memoria con el nombre, apellidos y DNI de quienes la hicieron.

Descripción y especificación de los TAD's implementados y estructuras de datos definidas. La especificación puede estar acompañada del pseudocódigo en PSelnt.

El proyecto tendría el siguiente planteamiento:



Tendremos las siguientes estructuras de datos:

- 2 colas
    1. Correspondiente a la Central de Paquetería donde se moverán los paquetes a los muelles de carga posteriormente. (Según su zona de entrega)
    2. Correspondiente a la Zona de Entrega, donde se almacenan los paquetes hasta que venga el Cliente a obtenerlos.
- 1 pila
    1. Correspondiente a la pila de Paquetes que se guardarán en los Muelles de Carga hasta que sean recogidos por la furgoneta para su entrega.
- 3 clases
    1. Correspondiente a la clase Paquete con las siguientes características:

- Tipo de paquete: Valor de tipo entero que indica dónde se encuentra el paquete (Zona Operativa) puede tener los valores 1, 2 ó 3.
- Etiqueta identificativa del paquete con los siguientes campos (Estructura/Struct):
    1. ID (Identificador único del paquete). Se trata de un array de 7 elementos:
        - 2 números aleatorios
        - 1 char (carácter) aleatorio
        - 4 números secuenciales
        - Por ejemplo: 46D1018
    2. GPS (Lugar de destino del paquete) Se trata de una estructura (struct) con los siguientes elementos:
        - Latitud, array de 6 números
        - Longitud, array de 6 números
        - Por ejemplo: GPS:struct[latitud:int[6] = 403830, longitud:int[6] = 403310]
    3. NIF (ó DNI del cliente, sirve para identificar al cliente) Se trata de un array de 9 elementos
        - 8 de esos elementos son números
        - 1 elemento (el último elemento) es una letra de las siguientes [TRWAGMYFPDXBNJZSQVHLCKE]
- Urgente (Atributo booleano). Se utiliza para indicar si un paquete tiene prioridad sobre otros.

2. Correspondiente a la clase Furgoneta con las siguientes características:
    - MuelleSalida (atributo de tipo entero) cuyos valores son [1..4], indicando desde dónde ha salido la furgoneta. Los valores están asociados a la cantidad de muelles de salida disponibles, si son diferentes al valor por defecto (4) deberán cambiarse, aumentando o disminuyéndolo según proceda.
    - ZonaEntrega (atributo de tipo entero) cuyos valores son [1..4], indicando hacia dónde debe llegar la furgoneta. Los valores están asociados a la cantidad de zonas de entrega disponibles, si son diferentes al valor por defecto (4) deberán cambiarse, aumentando o disminuyéndolo según proceda.
    - Paquetes (array de 5 elementos de tipo Paquete). El número de elementos está asociado a la cantidad de paquetes que puede llevar de una sola vez la furgoneta, si aumentamos la capacidad de carga (por ejemplo por cambiar las furgonetas por unas más grandes), debemos modificar este valor. (Lo mismo ocurriría al contrario)

- Repartiendo (atributo booleano). Indica si la furgoneta se encuentra actualmente repartiendo paquetes o está a la espera de carga/descarga.

3. Correspondiente a la clase Cliente con las siguientes características:
   - Datos (estructura/struct) se ha definido como estructura por si en un futuro aumentamos la cantidad de atributos identificadores del cliente. Es decir, aumentamos la clave primaria (NIF) insertanto claves secundarias para mejorar la identificación del cliente (por ejemplo añadiendo un campo como NombreApellidos). [Esta decisión de diseño se ha realizado para añadir identificabilidad al cliente, pensando en una posible expansión el en futuro]
     1. NIF (ó DNI del cliente, sirve para identificar al cliente) Se trata de un array de 9 elementos
        - 8 de esos elementos son números
        - 1 elemento (el último elemento) es una letra de las siguientes [TRWAGMYFPDXBNJZSQVHLCKE]

PECL 1- Estructuras de Datos. Simulación del funcionamiento de un servicio de paquetería.

Habiendo definido anteriormente las estructuras de datos que utilizaremos en el proyecto pasamos a realizar la especificación de los mismos utilizando pseudocódigo y PSeINT.

Para esta práctica utilizaremos los siguientes TAD's:

Clase Paquete

La clase consta de los siguientes atributos:

-Tipo: number, de 1,2,3

-Etiqueta: struct

-ID: array[7], 2 num aleatorios,1 char (aleatorio),4 numeros secuenciales

-GPS: struct

-latitud:array[6]:numeros

-longitud:array[6]:numeros

-NIF: array[9], 8 son numeros + 1 letra

-Urgente: bool

Se ha determinado esta definición debido a que un paquete consta de una Etiqueta que identifica al paquete y de un conjunto de carácterísticas, que dan información del paquete según el contexto sin llegar a identificarlo (Esa es la diferencia entre Etiqueta y [Tipo, Urgente]).

Quedando el siguiente pseudocódigo:

```
espec PAQUETE

usa Entero, Cadena, Logico

generos paquete

operaciones

var tipo, etiqueta_gps_latitud, etiqueta_gps_longitud:Entero

var etiqueta_id, nif:Cadena

var urgente:Logico

fespec
```

PECL 1- Estructuras de Datos. Simulación del funcionamiento de un servicio de paquetería.

Quedando la siguiente estructura de datos en PSeINT:

```
Algoritmo Paquete
    Definir tipo Como Entero
    Dimension etiqueta_id[7]
    Dimension NIF[6]
    Definir etiqueta_id,NIF Como Cadena
    Dimension etiqueta_gps_latitud[6]
    Dimension etiqueta_gps_longitud[6]
    Definir  etiqueta_gps_latitud, etiqueta_gps_longitud Como Entero
    Definir urgente Como Logico
FinAlgoritmo

```

Clase Furgoneta

Esta clase consta de los siguientes atributos:

-MuelleSalida:int 1,2,3,4

-ZonaEntrega:int 1,2,3,4

-Paquetes:array[5] de tipo Paquete

-Repartiendo:bool

Se ha determinado esta definición debido a que una furgoneta tiene una capacidad de carga de N paquetes, siendo N en nuestro caso 5, como se mencionó anteriormente puede aumentar o disminuir si mejoramos la infraestructura de transporte (las furgonetas). También posee información sobre su muelle de salida y la zona de entrega. La variable repartiendo nos indica si la furgoneta se encuentra en camino (o reparto), o si se encuentra cargando/descargando.

Quedando el siguiente pseudocódigo:

espec FURGONETA

usa Entero, Logico, Paquete

generos furgoneta

operaciones

var muelle_salida, zona_entrega:Entero

var paquetes:Paquete

var repartiendo:Logico

fespec

Quedando la siguiente estructura de datos en PSeINT:

```
1  Algoritmo Furgoneta
2      //Furgoneta
3      Definir muelle_salida Como Entero
4      Definir zona_entrega Como Entero
5      Dimension paquetes[5]
6      Definir Repartiendo Como Logico
7  FinAlgoritmo
8
```

Clase Cliente

La clase consta de los siguientes atributos:

-Datos:Struct

-NIF: array[9], 8 son numeros + 1 letra

Se ha determinado esta definición debido a que un cliente tiene una serie de datos identificadores (llave primaria y/o una/ninguna/varias llaves secundarias) para poder identificarlo y saber qué paquete le corresponde llegar a qué cliente.

¿Por qué se ha utilizado una estructura denominada Datos?

> Se ha utilizado la estructura Datos, debido a que lo que se encuentre dentro de esta estructura servirá para identificar a ese cliente sobre otros. Por ejemplo usando su NIF, su Nombre y Apellidos o incluso su correo electrónico si se desease. Utilizando esta estructuración de datos, nos garantizamos que si en un futuro deseamos añadir nuevos campos que identifiquen al cliente, podremos sin mucha complicación.

Si se añade un campo nuevo que no identifique al cliente, pero que sea importante guardarlo quedaría fuera de la estructura Datos, y se guardaría como un atributo directo en la clase Cliente.

Quedando el siguiente pseudocódigo:

> espec CLIENTE
>
> usa Cadena
>
> generos cliente
>
> operaciones
>
> var nif:Cadena
>
> fespec

Quedando la siguiente estructura de datos en PSeINT:

```
1  Algoritmo Cliente
2      Dimension nif[9]
3      Definir nif como Cadena
4  FinAlgoritmo
5
```

Estructura de Datos – MuellesdeCarga

Como se ha explicado anteriormente, los Muelles de Carga se tratan de pilas, su estructura se trata de un LILO ó FILO. Por lo tanto, cuando se reciban los paquetes N2 de la cola de Central de Paquetería, se irán guardando en la furgoneta según se reciban, a excepción de los paquetes prioritarios (los que tienen el atributo Urgente como True) que se guardarán los primeros siempre (sin excepción) en la furgoneta.

Quedando el siguiente pseudocódigo:

espec PILA[Paquete]

usa BOOLEANOS

parametro formal

generos Paquete

fparametro

generos pila

fespec

Cabe destacar que a esta estructura de datos se le pasan EXCLUSIVAMENTE objetos de la clase Paquete como parámetro, por lo que, esta estructura de datos se trata de una pila donde solamente se puede usar para paquetes.

Quedando la siguiente estructura de datos en PSeINT:

```
1  Algoritmo PilaMuelleCarga
2  +    Definir pila como Paquete; // se define el tipo de dato que va a tener el array que contendrá la pila.
       en inst. 1: Falta tipo de dato o tipo no válido.
3      Definir max como Entero; //  contiene el valor del tamaño máximo de la pila.
4      Definir tope como Entero; //contiene la posición superior actual de la pila.
5      Definir seleccion como Entero; //Contiene la opción que el usuario ha elegido
6  +    Definir dato como Paquete; //el valor del dato del tipo entero a guardar en las posiciones del array de la pila
       en inst. 1: Falta tipo de dato o tipo no válido.
7      Definir band como logico; //band actua como bandera lógica
8  FinAlgoritmo
9
```

Estructura de Datos – CentralPaqueteria

Como se ha explicado anteriormente, la Central de Paquetería se trata de una cola, su estructura sería FIFO ó LILO. Por lo tanto, cuando se almacenen temporalmente los paquetes N1 para su posterior entrega como paquetes N2 en los Muelles de Carga, los primeros que se hayan guardado serán los primeros en ser trasladados.

Quedando el siguiente pseudocódigo:

```
espec COLA[Paquete]

usa BOOLEANOS

parametro formal

generos Paquete

fparametro

generos cola
```

Cabe destacar que a esta estructura de datos se le pasan EXCLUSIVAMENTE objetos de la clase Paquete como parámetro, por lo que, esta estructura de datos se trata de una cola donde solamente se puede usar para paquetes.

Estructura de Datos – ZonaEntrega

Como se ha explicado anteriormente, la Zona de Entrega se trata de una cola, su estructura sería FIFO ó LILO. Por lo tanto, cuando se almacenen temporalmente los paquetes N3 para su posterior entrega a los clientes (se supone que los clientes van a las zonas de entrega para obtener los paquetes, y no los reciben en casa), los primeros que se hayan guardado serán los primeros en ser trasladados.

Quedando el siguiente pseudocódigo:

```
espec COLA[Paquete]

usa BOOLEANOS

parametro formal

generos Paquete

fparametro

generos cola
```

Cabe destacar que a esta estructura de datos se le pasan EXCLUSIVAMENTE objetos de la clase Paquete como parámetro, por lo que, esta estructura de datos se trata de una cola donde solamente se puede usar para paquetes.

Definición de sus operaciones.

Se han definido las siguientes operaciones para tratar con las estructuras de datos especificadas anteriormente.

Estructura de Datos – MuellesdeCarga

Debido a que se trata de una pila, se han utilizado las definiciones indicadas en Teoría.

En las pilas utilizaremos las siguientes operaciones:

operaciones

{crear una pila vacía} *pvacía: → pila* *{poner un elemento en la pila}* *apilar: elemento pila → pila*	Generadoras
{quitar un elemento de la pila} **parcial** *desapilar: pila → pila*	Modificadoras
{observar la cima de la pila} **parcial** *cima: pila → elemento* *{para ver si la pila está vacía}* *vacía?: pila → bool*	Observadoras

Quedando las siguientes ecuaciones básicas:

{Ahora que ya sabemos cuándo puede usarse una operación vamos a ver cómo se usa. Para ello ponemos los datos como si se hubiesen obtenido mediante las generadoras (cuando sea posible)}

ecuaciones

```
desapilar( apilar(x,p) ) = p
cima( apilar(x,p) ) = x
vacía?( pvacía ) = T
vacía?( apilar(x,p) ) = F
```

fespec

Se ha utilizado esta operación para obtener los elementos que residen en la pila:

Ejemplo: Contar cuántos elementos tiene una pila.

- Es una operación observadora (devuelve un natural)

  contar: pila → natural

- Las ecuaciones pueden ser

  contar(pvacía) = 0

  contar(apilar(x, p)) = suc(contar(p))

Definiéndolo en pseudocódigo de la siguiente forma:

Ejemplo: Contar cuántos elementos tiene una pila.

```
func contar (p:pila) dev n:natural                    {recursiva}
    si vacia?(p) entonces Devolver 0
                 si no devolver 1+ contar(desapilar(p))
finsi
finfunc
```

Para obtener la pila invertida sería la siguiente definición:

Ejemplo: Obtener la inversa de una pila, es decir, la pila resultante al cambiar el orden de los datos.

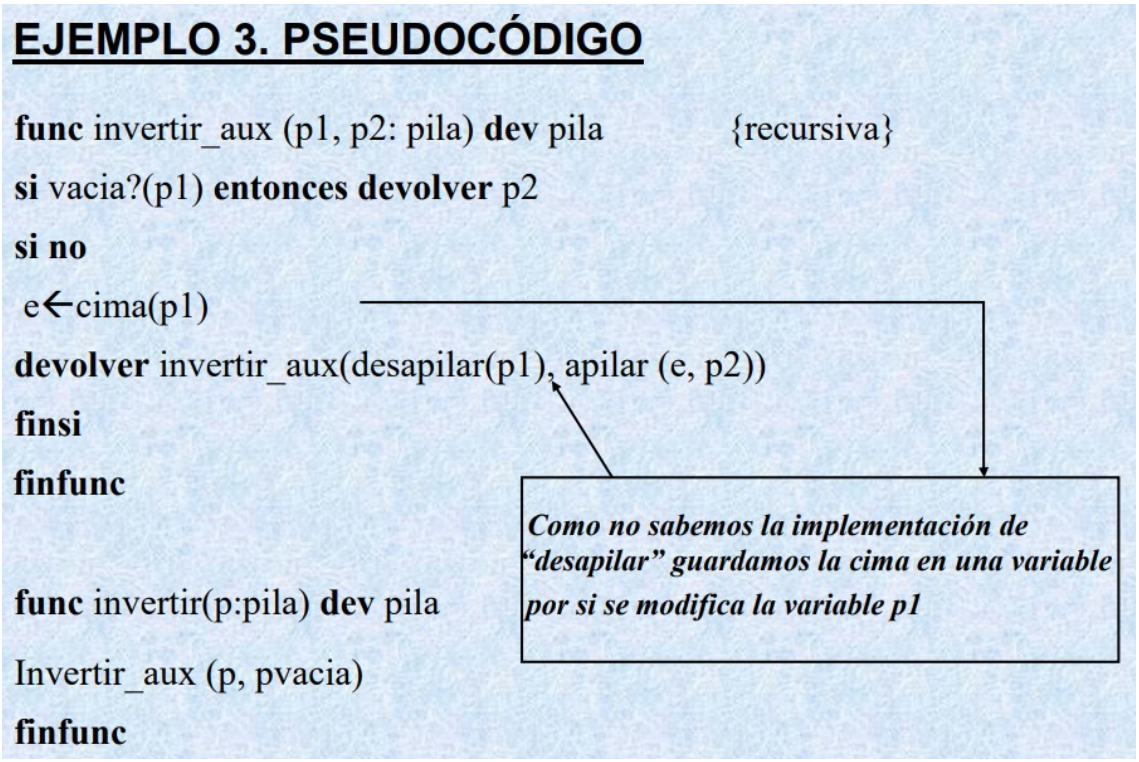
- Vamos a ir poniendo los datos de una pila en otra auxiliar hasta que no quede ninguno en la primera, y entonces se devuelve la pila auxiliar.

  invertir_aux: pila pila → pila
  invertir_aux(pvacía, p2) = p2
  invertir_aux(apilar(x,p1), p2) =
  invertir_aux(p1, apilar(x,p2))

- La operación que invierte una pila usa ***invertir_aux*** usando una pila vacía como pila auxiliar.

  invertir: pila → pila
  invertir(p) = invertir_aux(p, pvacía)

Quedando en pseudocódigo:



De esta forma se realiza la definición de las operaciones de la Pila de paquetes utilizadas en los muelles de carga.

Estructura de Datos – CentralPaqueteria y ZonaEntrega

Como se ha explicado anteriormente, se tratan de colas con la misma definición en su estructura de datos, por ello, tendrán las mismas operaciones.

Habiendo ya realizado su definición, sus operaciones serían las siguientes:

operaciones

{crear una cola vacía} *cvacía: → cola* *{poner un elemento en la cola}* *añadir: elemento cola → cola*	Generadoras
{quitar un elemento de la cola } **parcial** *eliminar: cola → cola*	Modificadoras
{ver el principio de la cola} **parcial** *primero: cola → elemento* *{ver si la cola está vacía}* *vacía?: cola → bool*	Observadoras

Tendrá las siguientes ecuaciones:

ecuaciones

```
eliminar( añadir(x,cvacía) ) = cvacía

vacía?(c)=F ⇒ eliminar( añadir(x,c) ) =
                          añadir( x,eliminar(c) )

primero( añadir(x,cvacía) ) = x

vacía?(c)=F ⇒ primero( añadir(x,c) ) =
                                    primero(c)

vacía?( cvacía ) = T

vacía?( añadir(x,c) ) = F
```

fespec

Para contar los elementos de la cola realizaremos lo siguiente:

Ejemplo: Contar cuántos elementos tiene una cola.

- La operación (es observadora) es la siguiente:

  contar: cola → natural

- Las ecuaciones pueden ser (basadas en generadoras)

  contar(cvacía) = 0

  contar(añadir(x, c)) = suc(contar(c))

- Las ecuaciones pueden ser (basadas en propiedades)

  vacía?(c) = T ⇒ contar(c) = 0

  vacía?(c) = F ⇒

  contar(c) = suc(contar(eliminar(c)))

Que en pseudocódigo sería:

Ejemplo: Contar cuántos elementos tiene una cola.

```
func contar (c:cola) dev n:natural                    {recursiva}
    si vacia?(c) entonces devolver 0
    sino   desencolar(c)
           devolver 1+ contar(c)
    finsi
finfunc
```

Para invertir la cola realizaríamos:

Ejemplo: Especificar una operación para obtener la inversa de una cola, es decir, la cola resultante al cambiar el orden de los datos.

- Para invertir una cola, cogemos el primer dato que esté en la cola y lo ponemos al final de los otros datos, y se repite hasta que no queden cosas en la cola.

  invertir: cola → cola

  invertir(cvacía) = cvacía

  vacía?(c)=F ⇒ invertir(c) = añadir(primero(c),invertir(eliminar(c)))

- No es necesario usar un acumulador (como en las pilas) porque las colas tienen dos puntos de acceso distintos.

Y para concatenar dos colas (unirlas) sería:

Ejemplo: Concatenar dos colas, poniendo la segunda después de la primera.

- Para concatenar una cola tras otra se pasan los datos uno a uno, de la segunda a la primera, hasta que no queden.
  concatenar: cola cola → cola
- Usando propiedades:
  vacía?(c2)=T ⇒ concatenar(c1,c2) = c1
  vacía?(c2)=F ⇒ concatenar(c1,c2) = concatenar(añadir(primero(c2),c1),eliminar(c2))
- Usando generadores:
  concatenar(c1,cvacía) = c1
  concatenar(c1,añadir(x,c2)) = añadir(x,concatenar(c1,c2))

Estas son las estructuras de datos más importantes y a las cuales es necesario definir sus operaciones.

Para las siguientes clases, debido a su simplicidad, no se ha optado por definir sus operaciones ya que utilizarán las operaciones básicas de sus datos simples (operaciones de enteros, strings, etc):

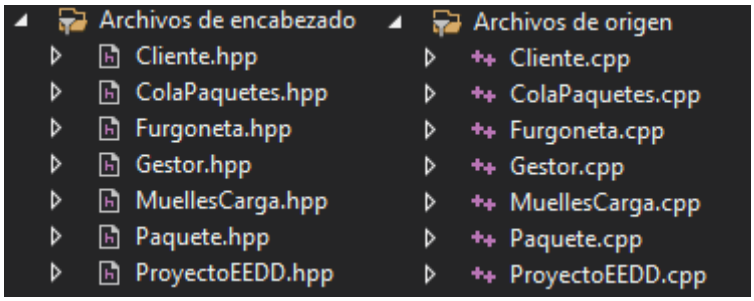
- Clase Paquete
- Clase Cliente

- Clase Furgoneta

Explicación del funcionamiento del programa y de los métodos/funciones implementadas.

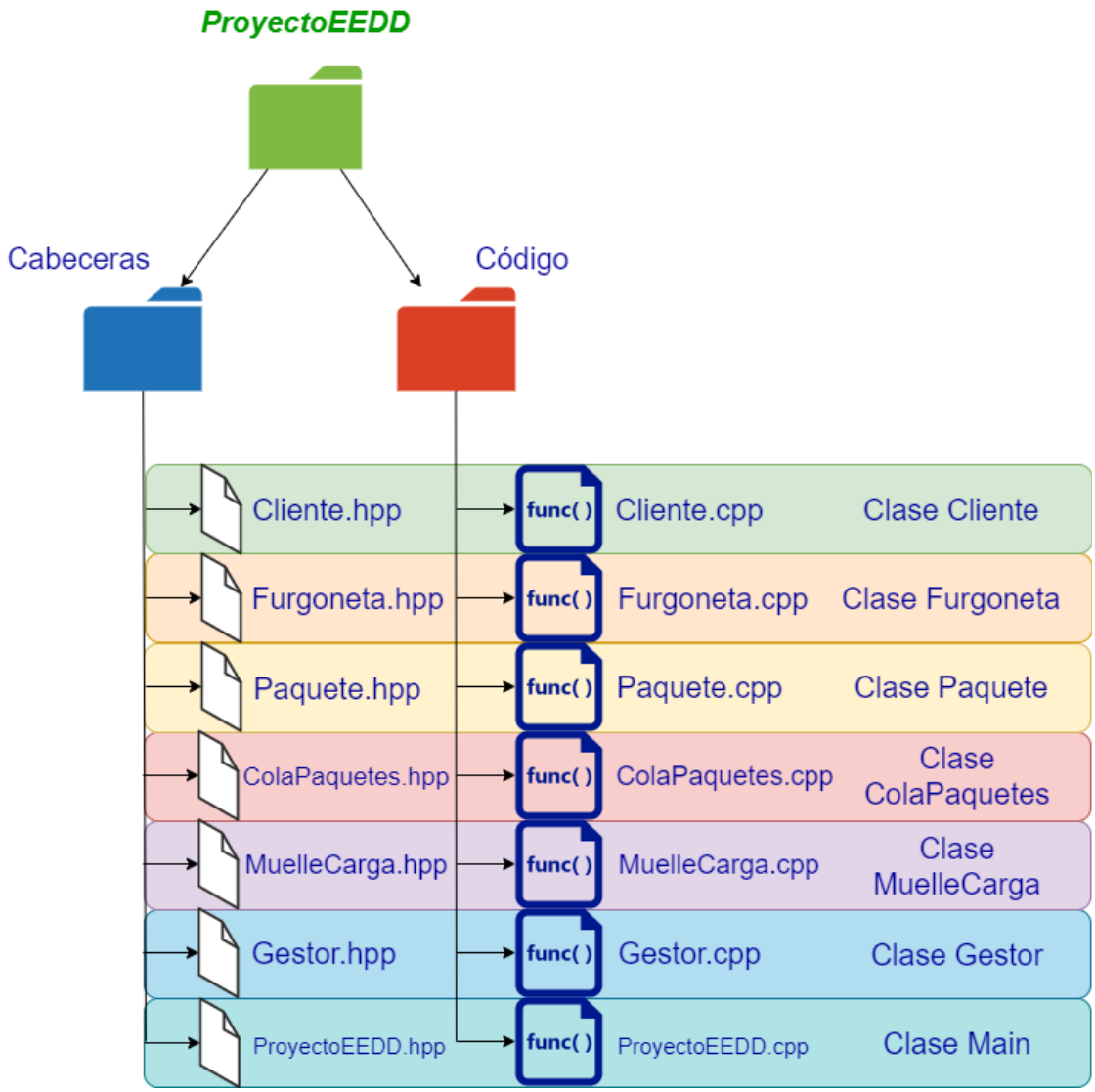
Siguiendo la definición y especificación explicadas anteriormente de los TADs, se ha realizado la siguiente implementación:

Se han creado las siguientes clases implementando los ficheros:



En las cabeceras (ficheros .hpp) se ha definido la estructura de los datos, y en los ficheros origen (.cpp) se ha realizado la implantación de las funciones y/o métodos.

Quedando una estructura de ficheros de la siguiente forma:



A continuación se realizará una explicación del código y las clases:

Clase Paquete

Se ha definido su estructura de datos a partir de datos simples como (struct, int char y string).

Se han encapsulado unas dentro de otras para establecer los datos necesarios:

```
// Estructuras
struct Latitud {
    int grados;
    int minutos;
    int segundos;
    Latitud();
    Latitud(int grados_, int minutos_, int segundos_);
};
struct Longitud {
    int grados;
    int minutos;
    int segundos;
    Longitud();
    Longitud(int grados_, int minutos_, int segundos_);
};
struct Gps {
    Latitud latitud;
    Longitud longitud;
    Gps();
    Gps(Latitud latitud_, Longitud longitud_);
    Gps(int grados_lat, int minutos_lat, int segundos_lat, int grados_lon, int minutos_lon, int segundos_lon);
};
struct ID
{
    int dos_primeros; // 2 elementos
    char letra;
    std::string numeros_restantes; // 4 elementos
    ID();
    ID(std::string numeros_restantes_);
    ID(int dos_primeros_, char letra_, std::string numeros_restantes_);
};

struct Etiqueta {
    ID id; // 7 elementos
    Gps gps;
    std::string nif; // 9 elementos
    Etiqueta();
    Etiqueta(ID id_, Gps gps_, std::string nif_);
};
```

```
// Atributos
int tipo;
Etiqueta etiqueta;
bool urgente;
```

Posteriormente se ha realizado la especificación de los constructores y funciones de la siguiente forma:

- Función generadora de la estructura Latitud, que genera la Latitud de forma “aleatoria”.

```
Paquete::Latitud::Latitud()
{
    grados = 40;
    minutos = 32 + rand() % (42 - 32);
    segundos = 5 + rand() % (7 - 5);
}
```

- Función generadora de la estructura Longitud, que genera la Longitud de forma “aleatoria”.

```
Paquete::Longitud::Longitud()
{
    grados = -3;
    minutos = 46 + rand() % (52 - 46);
    segundos = 5 + rand() % (7 - 5);
}
```

Función generadora del identificador de la etiqueta, que genera los dos primeros números de forma “aleatoria” según un rango; y la letra de igual manera:

```
Paquete::ID::ID(std::string numeros_restantes_)
{
    char letras[24] = "TRWAGMYFPDXBNJZSQVHLCKE";

    dos_primeros = rand() % 90 + 10; // in the range 10-99
    letra = letras[rand()%23];
    numeros_restantes = numeros_restantes_;
}
```

PECL 1- Estructuras de Datos. Simulación del funcionamiento de un servicio de paquetería.

- Función generadora del NIF de la etiqueta del paquete en función del NIF de uno de los integrantes del proyecto:

```
std::string Paquete::generarNIF()
{
    std::string s = "";
    for (int i = 0; i < 8; i++)
    {
        char num = "06021665"[rand() % 8];
        s += num;
    }
    s += "N";
    return  s;
}
```

- Un ejemplo del constructor de la clase:

```
Paquete::Paquete(std::string id_, int grados_lat_, int minutos_lat_, int segundos_lat_, int grados_lon_, int minutos_lon_, int segundos_lon_,
    bool urgente_)
{
    tipo = 0;
    etiqueta.id = ID(id_);
    etiqueta.gps.latitud.grados = grados_lat_;
    etiqueta.gps.latitud.minutos = minutos_lat_;
    etiqueta.gps.latitud.segundos = segundos_lat_;
    etiqueta.gps.longitud.grados = grados_lon_;
    etiqueta.gps.longitud.minutos = minutos_lon_;
    etiqueta.gps.longitud.segundos = segundos_lon_;
    etiqueta.nif = generarNIF();
    urgente = urgente_;
}
```

Clase Furgoneta:

Depende de la clase Paquete para su ejecución.

- Se han definido los siguientes atributos y métodos:

```
class Furgoneta
{
public:
    // Atributos
    int muelle_salida;
    int zona_entrega;
    Paquete* paquetes;
    bool repartiendo;

    // Constructor
    Furgoneta();
    Furgoneta(int muelle_salida_, int zona_entrega_, Paquete* paquetes_);
    Furgoneta(int muelle_salida_, int zona_entrega_, Paquete* paquetes_, bool repartiendo_);
    // Métodos
    bool estado();
    void reparte();
    void espera();
};
```

- Se ha establecido la siguiente lógica para saber cuando una furgoneta está disponible para salir o está repartiendo:

```
void Furgoneta::reparte()
{
    repartiendo = true;
}

void Furgoneta::espera()
{
    repartiendo = false;
}

bool Furgoneta::estado()
{
    return repartiendo;
}
```

- Ejemplo de constructor utilizado en la clase:

```
Furgoneta::Furgoneta(int muelle_salida_, int zona_entrega_, Paquete* paquetes_, bool repartiendo_)
{
    muelle_salida = muelle_salida_;
    zona_entrega = zona_entrega_;
    paquetes = paquetes_;
    repartiendo = repartiendo_;
}
```

PECL 1- Estructuras de Datos. Simulación del funcionamiento de un servicio de paquetería.

Clase Cliente

- Se ha realizado la siguiente definición de clase:

```
class Cliente
{
public:
    // Atributos
    struct {
        std::string nif;
    } datos;

    // Constructor
    Cliente();
    Cliente(std::string nif_);
    // Metodos
    std::string getNif();
};
```

- Ejemplo de constructor:

```
Cliente::Cliente()
{
    datos.nif = "";
}

Cliente::Cliente(std::string nif_)
{
    datos.nif = nif_;
}
```

Esta clase permite fácilmente modificar la lógica del programa para detectar qué paquetes han recibido determinados clientes.

Es una clase simple creada para simplificar y abstraer la complejidad de conexión de los diferentes elementos que componen el proyecto.

Clase ColaPaquetes

Clase que depende de la clase Paquete.

- Se compone de una clase interna anidada llamada Nodo, con la siguiente estructura:

```
class ColaPaquetes
{
private:
    class Nodo // Clase Nodo para controlar los elementos de la cola
    {
    public:
        Nodo();
        Nodo(Paquete p, Nodo* sig = NULL);
        ~Nodo();
        Paquete paquete;
        Nodo* siguiente;
    };
    // Atributos ColaPaquetes
    Nodo* frente_;
    Nodo* final_;
```

- Cuyos métodos de clase son:

```
public:
    // Constructor
    ColaPaquetes();
    ~ColaPaquetes();
    // Metodos
    void encolar(Paquete p);
    Paquete desencolar();
    Paquete vaciar();
    bool es_vacia();
};
```

Se tienen dos atributos Nodo para tener constancia del primer elemento de la cola y el último, así también para saber cómo navegar por la cola al realizar el desencolado o el vaciado. Esta clase se ha implementado siguiendo lo explicado en la especificación de las operaciones de datos, basándose en la teoría de la asignatura.

- Ejemplo de constructores de Nodo y ColaPaquetes:

```
ColaPaquetes::Nodo::Nodo()
{
    paquete = Paquete();
    siguiente = nullptr;
}

ColaPaquetes::Nodo::Nodo(Paquete p, Nodo* sig)
{
    paquete = p;
    siguiente = sig;
}
```

```
ColaPaquetes::ColaPaquetes()
{
    frente_ = NULL;
    final_ = NULL;
}
```

- Cabe destacar que el destructor de ColaPaquetes debe desencolar los paquetes que se encuentren en la cola (si no estuviese vacía) descartándolos.

```
ColaPaquetes::~ColaPaquetes()
{
    while (frente_) {
        desencolar();
    }
}
```

PECL 1- Estructuras de Datos. Simulación del funcionamiento de un servicio de paquetería.

- El encolado se ha realizado de la siguiente forma: [No se explica porque los comentarios del código lo hacen]

```
void ColaPaquetes::encolar(Paquete p)
{
    Nodo* nuevo = new Nodo(p); // Se crea un nodo nuevo
    // Si cola no vacia, se annade el nuevo a continuacion del ultimo
    if (final_) {
        final_->siguiente = nuevo;
    }
    // El ultimo elemento de la cola es el nuevo nodo
    final_ = nuevo;
    // Si frente es NULL, la cola esta vacia y el nuevo nodo pasa a ser el primero
    if (!frente_) {
        frente_ = nuevo;
    }
}
```

- El desencolado de esta manera: [No se explica porque los comentarios del código lo hacen]

```
Paquete ColaPaquetes::desencolar()
{
    Nodo* nodo; // Variable auxiliar para manipular nodo
    Paquete p; // Variable auxiliar para retorno del valor
    // Nodo apunta al primer elemento de la cola
    nodo = frente_;
    if (!nodo) {
        throw "ERROR -- No hay elementos en la cola."; // Si no hay nodos en la cola se devuelve 0
    }
    // Se asigna a frente la direccion del segundo nodo
    frente_ = nodo->siguiente;
    // Se guarda el valor de retorno
    p = nodo->paquete;
    delete nodo;
    // Si cola vacia, el ultimo debe ser NULL tambien
    if (!frente_) {
        final_ = NULL;
    }
    return p;
}
```

- El vaciado de la cola: [Retorna el último valor de la cola]

```
Paquete ColaPaquetes::vaciar()
{
    Nodo* nodo; // var aux para manipular nodo
    Paquete p; // var aux para retorno del valor
    nodo = frente_;
    // Se guarda el valor de retorno
    p = nodo->paquete;
    while (frente_) {
        desencolar();
    }
    return p;
}
```

- Comprobación para saber si la cola de paquetes está vacía:

```
bool ColaPaquetes::es_vacia()
{
    return ((frente_ == NULL) && (final_ == NULL));
}
```

Cabe destacar que esta clase está programada solamente para tratar con valores de tipo Paquete, no acepta ningún otro tipo de clase. Por lo tanto, es una clase específica creada para este proyecto.

Clase MuellesCarga

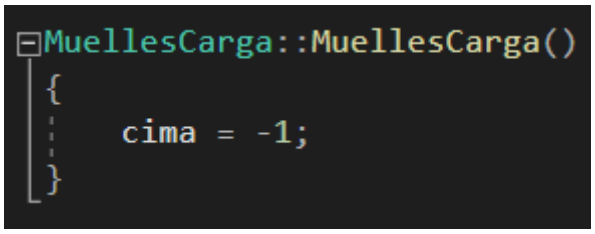
Depende de la clase Paquete y la cabecera ProyectoEEDD (Clase principal de ejecución).

Se trata de una pila estática que solamente acepta objetos Paquete, por lo que no podrá usarse en otros contextos (no es Genérica).

- Se ha realizado la siguiente implementación de la clase:

```
class MuellesCarga // Zona operativa N2
{
private:
    static const int elementos_pila = NELEMENTOSMUELLESCARGA;
    int cima;
    Paquete paquetes[elementos_pila];
public:
    // Constructor
    MuellesCarga();
    // Destructor
    ~MuellesCarga();
    //Metodos
    bool vacio();
    bool lleno();
    void introducir(Paquete p);
    Paquete obtener();
    int size();
};
```

- El atributo elementos_pila se utiliza para especificar el número de elementos máximos que tendrá la pila.
- La cima indica el elemento que se encuentra en la parte de arriba de la pila e indica el siguiente elemento a sacar.
- El atributo paquetes es un array que utiliza la memoria estática (stack) para almacenar los objetos de tipo Paquete.
- El resto de métodos/funciones son sirven para dar lógica a la pila y se explicarán a continuación.
- Ejemplo de constructor:


MuellesCarga::MuellesCarga()
{
cima = -1;
}

- Métodos para determinar si la pila está vacía o llena, respectivamente:

```
bool MuellesCarga::vacio()
{
    if (cima == -1)
        return true;
    else
        return false;
}

bool MuellesCarga::lleno()
{
    if (cima == elementos_pila - 1)
        return true;
    else
        return false;
}
```

- Métodos para introducir nuevos paquetes a la pila o para sacarlos, así como un método para determinar el tamaño de la pila (número de elementos actuales):

```
void MuellesCarga::introducir(Paquete p)
{
    if (!lleno()) {
        paquetes[++cima] = p;
    }
    else {
        throw "ERROR -- No se puede añadir el paquete.";
    }
}

Paquete MuellesCarga::obtener()
{
    if (!vacio()) {
        return paquetes[cima--];
    }
    else {
        throw "ERROR -- No se pueden extraer valores.";
    }
}

int MuellesCarga::size()
{
    return cima+1;
}
```

Clase Gestor

Una de las clase principales y más importantes del proyecto, se ha programado pensando en el paradigma de Modelo-Vista-Controlador, por lo que las clases anteriores se corresponderían a los modelos, y esta clase se trata del controlador. Es la encargada de aplicar la lógica e interconexión entre los diferentes modelos (estructuras de datos) para la correcta ejecución del programa.

En este proyecto, se ha considerado como vísta la clase main de ejecución (ProyectoEEDD) y se ha considerado como visualización el modo texto o consola, se podría adaptar fácilmente para añadirle una interfaz gráfica.

- La clase consta de las siguientes dependencias (debido a que se encarga de interconectar los diferentes modelos del proyecto):

```
#include "Paquete.hpp"
#include "Furgoneta.hpp"
#include "MuellesCarga.hpp"
#include "ColaPaquetes.hpp"
#include <vector>
#include <iostream>
```

- Consta de los siguientes atributos:

```
// Atributos
ColaPaquetes central;
std::vector<MuellesCarga> muelles_carga;
std::vector<Furgoneta> furgonetas;
std::vector<ColaPaquetes> zona_entregas;
int max_furgonetas;
```

- Y de los siguientes métodos (cuya implementación se explicará a continuación):

```
// Metodos
void generarPaquetes(int n1); // El parametro es N1
void transportarPaquetesMuelles(int n2);
void transportarMuelleFurgoneta(int n3);
void transportarFurgonetaEntrega(int n3);
void mostrarEntregas();
void crearFurgoneta(int muelle_, int zona_entrega_, Paquete* paquetes);
void reutilizarFurgoneta(int muelle_, int zona_entrega_, Paquete* paquetes);
```

PECL 1- Estructuras de Datos. Simulación del funcionamiento de un servicio de paquetería.

- Método para generar los paquetes de la cola Central de Paquetería:

```
void Gestor::generarPaquetes(int n1)
{
    int secuencia = 0;
    std::string ssecuencia;
    for (int i = 0; i < n1; i++) {
        ssecuencia = std::to_string(secuencia);
        if (ssecuencia.size() < 4) {
            int temp = ssecuencia.size();
            for (int u = 0; u < (4 - temp); u++) {
                ssecuencia.insert(0, "0");
            }
        }
        central.encolar(Paquete(ssecuencia));
        secuencia++;
    }
}
```

- Método para transportar los paquetes a los muelles para su posterior envío:

```
void Gestor::transportarPaquetesMuelles(int n2)
{
    Paquete p; // Variable auxiliar paquete.
    MuellesCarga muelle; // Variable auxiliar muelle.
    for (int i = 0; i < n2; i++) {
        p = central.desencolar();
        // Inserto el paquete de la central en los muelles de carga segun su latitud.
        int index = p.etiqueta.gps.latitud.minutos % muelles_carga.size();
        muelle = muelles_carga.at(index);
        muelle.introducir(p);
        muelles_carga.at(index) = muelle;
        std::cout << "\tSacando paquete " << p.etiqueta.id.dos_primeros << p.etiqueta.id.letra
            << p.etiqueta.id.numeros_restantes << " para introducirlo en el muelle de carga "
            << index
            << std::endl;
    }
}
```

- Método para meter los paquetes de la pila de muelles a las furgonetas:

```
void Gestor::transportarMuelleFurgoneta(int n3)
{
    Paquete* paquetes;
    Paquete aux; // Variable auxiliar de paquete
    for (int i = 0; i < muelles_carga.size(); i++) { // Recorro cada muelle de carga
        paquetes = new Paquete[n3]();
        // Comprobamos si los paquetes almacenados en el muelle caben en una furgoneta
        if (muelles_carga.at(i).size() >= n3) {
            for (int u = 0; u < n3; u++) {
                // Preparamos los paquetes para ser almacenados en la furgoneta
                aux = muelles_carga.at(i).obtener();
                paquetes[u] = aux;
            }
            if (furgonetas.size() == NFURGONETAS) {
                reutilizarFurgoneta(i, i, paquetes);
            }
            else
            {
                crearFurgoneta(i, i, paquetes);
            }
        }
    }
}
```

- Método para pasar los paquetes de las furgonetas a las zonas de entrega:

```
void Gestor::transportarFurgonetaEntrega(int n3)
{
    ColaPaquetes* aux; // var auxiliar de Zona_Entrega
    Furgoneta f;
    for (int i = 0; i < furgonetas.size(); i++) {
        f = furgonetas.at(i);
        if (f.estado()) {
            aux = &zona_entregas.at(f.zona_entrega);
            for (int i = 0; i < n3; i++) {
                aux->encolar(f.paquetes[i]);
            }
            zona_entregas.at(f.zona_entrega) = *aux;
            f.espera(); // Cerramos la furgoneta
            furgonetas.at(i) = f;
            std::cout << "\tFurgoneta " << i << " a la espera."
                << std::endl;
        }
    }
}
```

- Métodos para crear furgonetas o reutilizarlas según proceda:

```
void Gestor::crearFurgoneta(int muelle_, int zona_entrega_, Paquete* paquetes)
{
    if (furgonetas.size() <= max_furgonetas) {
        furgonetas.push_back(Furgoneta(muelle_,zona_entrega_, paquetes, true));
        std::cout << "\tFurgoneta creada " << muelle_ << " - " << zona_entrega_
            << " - " << paquetes
            << std::endl;
    }
}

void Gestor::reutilizarFurgoneta(int muelle_, int zona_entrega_, Paquete* paquetes)
{
    Furgoneta f;
    for (int i = 0; i < furgonetas.size(); i++) {
        f = furgonetas.at(i);
        // Suponemos que las furgonetas vuelven al muelle de donde salieron cuando finalizan su trayecto.
        if (f.estado() == false) {
            f.muelle_salida == muelle_;
            f.paquetes = paquetes;
            f.zona_entrega = zona_entrega_;
            f.reparte();
            furgonetas.at(i) = f;
            std::cout << "\tFurgoneta reutilizada " << muelle_ << " - " << zona_entrega_
                << " - " << paquetes
                << std::endl;
            break;
        }
    }
}
```

- El método mostrarEntregas te da estadísticas e información de los paquetes recibidos:

```
void Gestor::mostrarEntregas()
{
    Paquete p;
    int z0 = 0, z1 = 0, z2 = 0, z3 = 0, aux = 0; // Contadores para sacar estadisticas de las zonas.
    std::cout << "Mostrando los paquetes entregados...\n";
    for (int i = 0; i < zona_entregas.size(); i++) {
        while (!zona_entregas.at(i).es_vacia()) {
            p = zona_entregas.at(i).desencolar();
            std::cout << "\tZona de Entrega " << i << " -- Paquete: ID: "
                << p.etiqueta.id.dos_primeros << p.etiqueta.id.letra << p.etiqueta.id.numeros_restantes
                << ", GPS: " << p.etiqueta.gps.latitud.grados << " " << p.etiqueta.gps.latitud.minutos
                << " " << p.etiqueta.gps.latitud.segundos << " _ " << p.etiqueta.gps.longitud.grados
                << " " << p.etiqueta.gps.longitud.minutos << " " << p.etiqueta.gps.longitud.segundos
                << ", NIF: " << p.etiqueta.nif <<  std::endl;
```

Clase ProyectoEEDD

En el fichero cabecera de esta clase (ProyectoEEDD.hpp), se han definido diferentes parámetros para cambiar la ejecución del programa según se necesite.

```
#pragma once
#define N1 100 // Numero de paquetes generados en la central de paqueteria
#define N2 10  // Numero de paquetes que se obtendran de CentralPaqueteria y se repartiran entre los MuellesCarga
#define N3 5   // Numero de paquetes que almacenan las furgonetas
#define NELEMENTOSMUELLESCARGA 10 // Numero de paquetes que puede almacenar la pila de MuellesCarga
#define NMUELLESCARGA 4 // Numero de muelles de carga
#define NZONAENTREGA 4 // Numero de Zonas de entrega
#define NFURGONETAS 4 // Numero de furgonetas
```

Posteriormente en el fichero ProyectoEEDD.cpp donde se encuentra la ejecución principal del programa, se han realizado las siguientes llamadas a la clase controlador Gestor:

```
Gestor* g = new Gestor(NMUELLESCARGA, NFURGONETAS, NZONAENTREGA);
g->generarPaquetes(N1);
cout << "\t\t\tPaquetes generados correctamente... " << endl;
cout << "\tPresione enter para continuar" << endl;
getchar();

for (int contador = 0; contador < (N2 % N1); contador++) {
    cout << "\t\tTransportando paquetes a los muelles..." << endl;
    g->transportarPaquetesMuelles(N2);
    cout << "\t\t\tPaquetes transportados correctamente... " << endl;
    cout << "\t\tTransportando paquetes de los muelles a las furgonetas..." << endl;
    g->transportarMuelleFurgoneta(N3);
    cout << "\t\t\tPaquetes transportados correctamente... " << endl;
    cout << "\t\tEntregando los paquetes de las furgonetas a las zonas entrega..." << endl;
    g->transportarFurgonetaEntrega(N3);
    cout << "\t\t\tPaquetes entregados correctamente... " << endl;
    cout << "\tPresione enter para continuar" << endl;
    getchar();
}
cout << "\t\tMostrando entregas realizadas..." << endl;
g->mostrarEntregas();
cout << "Fin de la ejecucion.\n";
cout << "\tPresione enter para salir" << endl;
getchar();
```

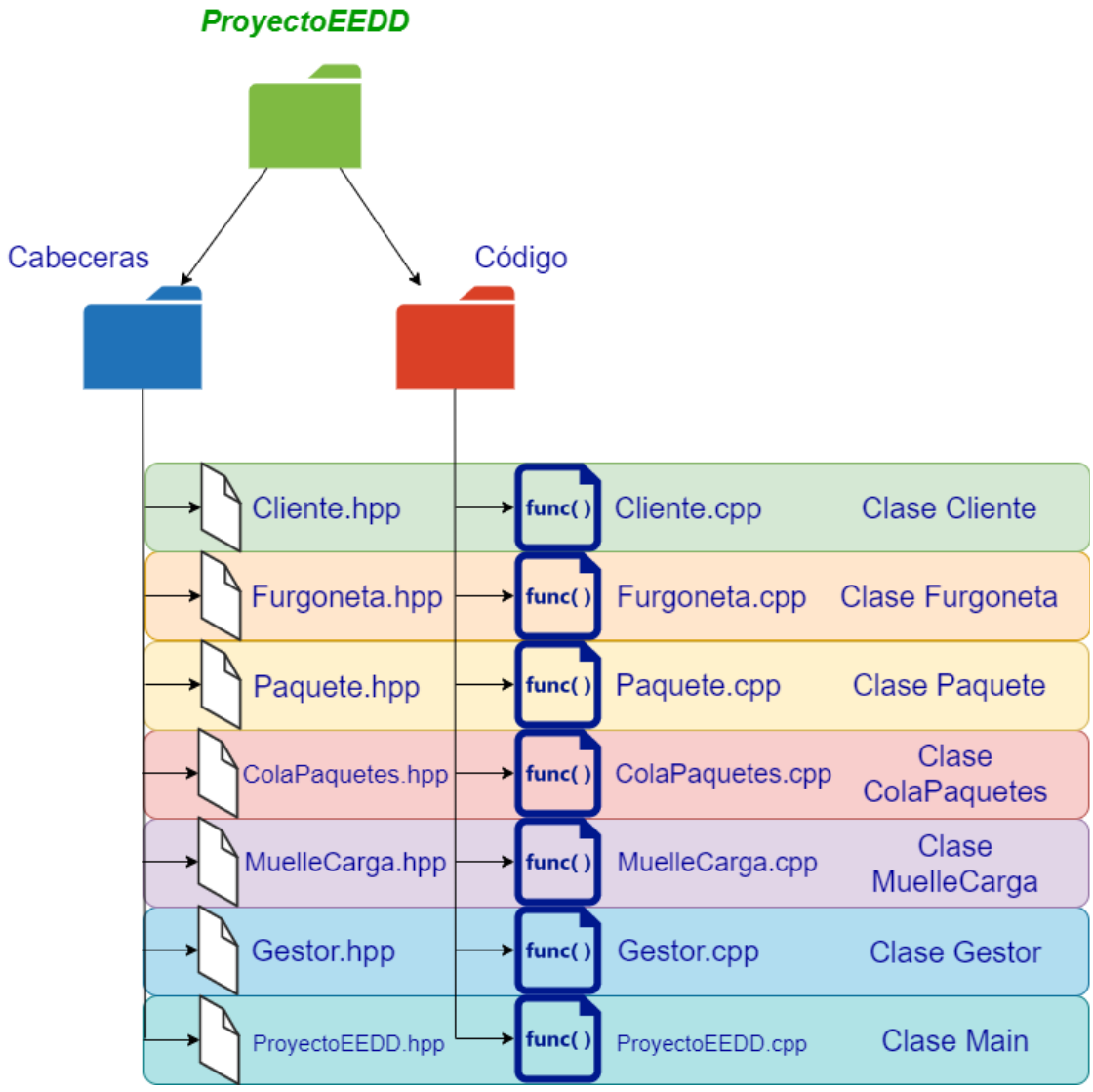
Cabe destacar que esta clase solamente depende del Gestor para su ejecución.

A continuación se procederá a explicar las diferentes problemáticas encontradas en el proyecto.

Problemas encontrados durante el desarrollo de la práctica y solución adoptada.

Estructuración adecuada del proyecto para poder aislar correctamente las clases siguiendo el paradigma de programación de objetos:

- Se ha optado por una estructura de clases que se basa en el Modelo-Vista-Controlador para la estructuración del proyecto quedando de esta manera.



Implementación de la cola de paquetes:

Se ha optado por la implementación utilizada en teoría, debido a que es la más sencilla y cumple con el propósito del proyecto. Cabe destacar que se trata de una cola dinámica. El uso de nodos simplifica mucho la abstracción del TAD, lo cuál ha resultado muy útil:

```
class Nodo // Clase Nodo para controlar los elementos de la cola
{
public:
    Nodo();
    Nodo(Paquete p, Nodo* sig = NULL);
    ~Nodo();
    Paquete paquete;
    Nodo* siguiente;
};
// Atributos ColaPaquetes
Nodo* frente_;
Nodo* final_;
```