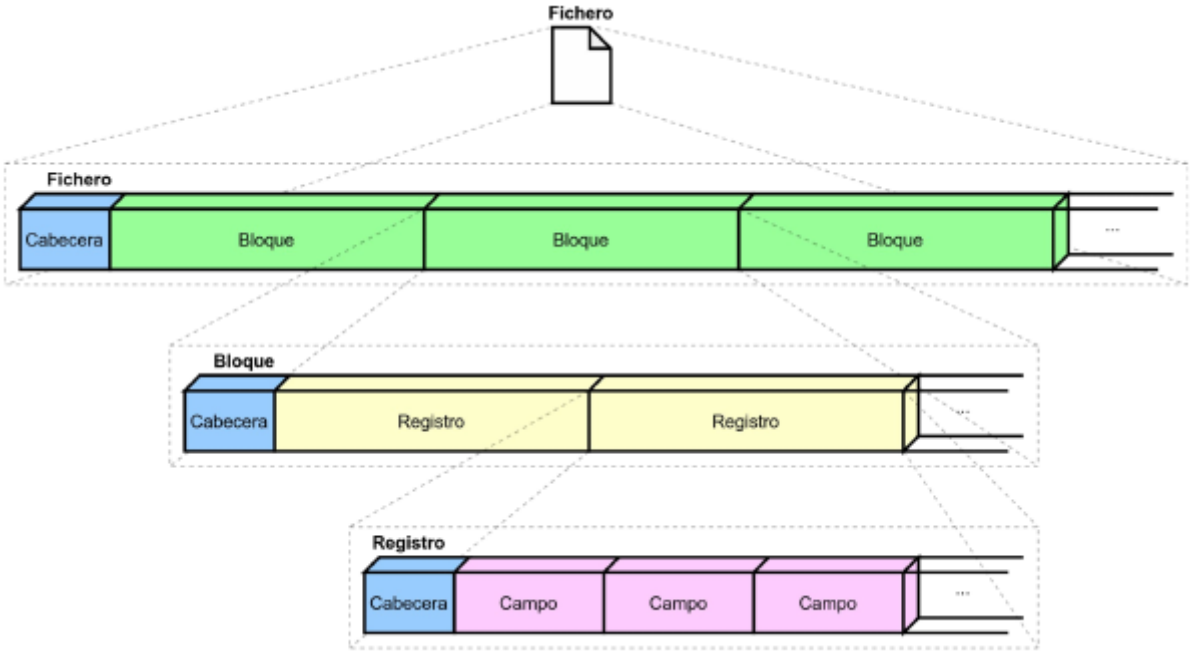
ABD 21/22 - Práctica 1: organización de ficheros

Clara María Romero Lara -- Grupo A2

1. Introducción

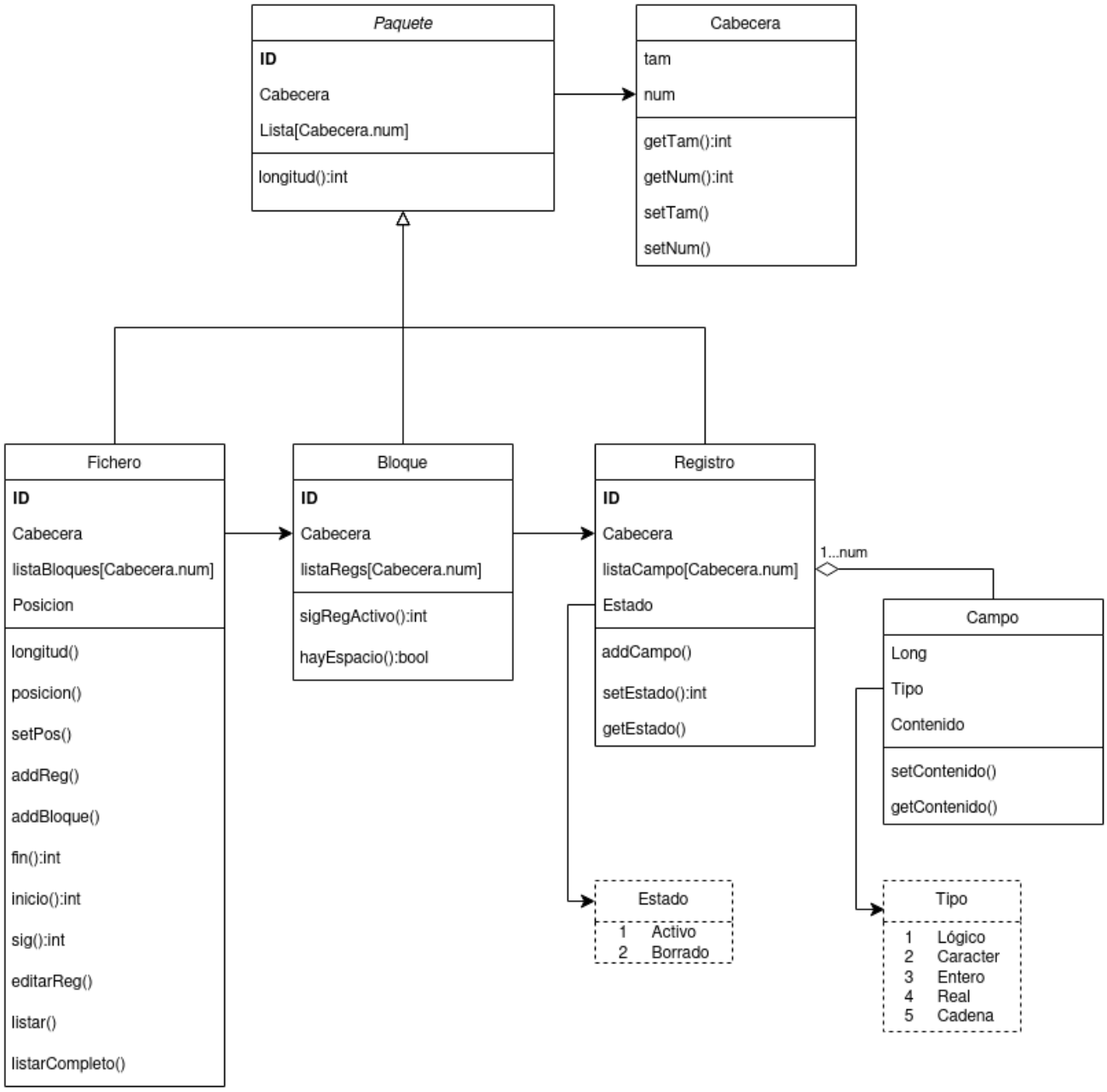
El objetivo de esta práctica es diseñar las estructuras de datos para la organización de archivos secuenciales físicos. La estructura corresponde a la del siguiente esquema:



Debemos diseñar las estructuras de datos necesarias así como la especificación de sus operaciones, incluyendo qué operaciones requieren de lectura/escritura al disco, apertura/cierre de fichero y desplazamiento en el disco.

2. Estructuras de datos

El diagrama planteado para la implementación es el siguiente. A continuación desarrollaremos las especificaciones de las estructuras de datos y clases planteadas.



Paquete y cabecera

Partimos de una clase genérica a la que llamaremos Paquete, que contiene un objeto de clase Cabecera. De esta clase heredarán Fichero, Bloque y Registro.

- Clase Cabecera. Contiene los atributos:
    - `int tam`, el tamaño de la cabecera.
    - `int num`, el número de elementos contenidos en el Paquete.

  Esta clase contiene estos métodos:
    - Métodos de lectura y escritura de los atributos:
        - `int getTam()`
        - `void setTam(int t)`
        - `int getNum()`
        - `void setNum(int t)`

- Clase Paquete. Contiene los siguientes atributos:
    - Un identificador.
    - Una cabecera.
    - Una lista genérica. Su longitud la conocemos por el atributo num de la cabecera.

  Esta clase contiene el método `int longitud()`, que devuelve el tamaño del paquete.

Fichero

- Clase Fichero, hereda de Paquete. Contiene los siguientes atributos:
    - Un identificador.
    - Una cabecera.
    - Una lista de datos tipo Bloque, cuya longitud es la establecida en la cabecera (atributo `num`).
    - Un indicador de la posición actual. Inicializa a 0+`tam` de la cabecera de fichero.

  Esta clase contiene estos métodos:
    - `int posicion(Registro reg)` - dado un registro, devuelve su posición.
    - `bool setPosicion(int pos)` - modifica la posición actual. Devuelve 0 si la nueva posición es posterior a la previa, y 1 en el caso de ser anterior.

- `void addReg()` - añade un registro al final del fichero. Para ello, emplea las funciones `posicion()`, `fin()` y `hayEspacio()` de Bloque. Si la función `hayEspacio()` devuelve 0, llama a la función `addBloque()` y luego se llama a sí misma (buscando otra vez la posición del último registro y viendo que, ahora sí, hay espacio para añadir el registro).
- `void addBloque()` - es llamada por `addReg()` y no debería llamarse por su cuenta para evitar que se use innecesariamente antes de que se llenen los bloques. Añade un bloque al final del fichero haciendo uso de la posición de `fin()`.
- `Registro fin()` - este método devuelve el último registro activo del fichero. Lo hace empleando la función de Bloque `sigRegActivo()` desde la posición actual hasta la última posición, determinada por el `num` de la cabecera (así que existe la posibilidad de que lea registros de más, si se da el caso de que los últimos registros de la lista no estén activos).
- `Registro inicio()` - este método devuelve el primer registro activo del fichero. Lo hace empleando la función `sigRegActivo()` de Bloque desde el inicio+`tam` cabecera de fichero hasta encontrar el primer registro activo.
- `Registro sig(int pos)` - usando la posición actual y la función `sigRegActivo()`, devuelve el siguiente registro activo.
- `void listar(int pos_ini, int pos_fin)` - lista todos los registros desde la posición de inicio hasta la posición final. Si no se indican, se asume por defecto desde la posición actual hasta la última posición.
- `void listarCompleto()` - lista todos los registros del archivo.

Bloque

- Clase Bloque, hereda de Paquete. Contiene los siguientes atributos:
  - Un identificador.
  - Una cabecera.
  - Una lista de datos tipo Registro, cuya longitud es la establecida en la cabecera (atributo `num`).

  Esta clase contiene estos métodos:
  - `Registro sigRegActivo(int pos)` - dada una posición de inicio, busca el siguiente registro de estado activo hasta la posición de `fin()`.
  - `bool hayEspacio(int tam)` - dado el tamaño de un registro, se comprueba si cabe en el bloque en base al `tam` especificado en la cabecera.

Registro y campo

- Clase Registro, hereda de Paquete. Contiene los siguientes atributos:
    - Un identificador.
    - Una cabecera.
    - Una lista de datos tipo Campo, cuya longitud es la establecida en la cabecera (atributo `num`).
    - `int estado`, un bit de 2 posibles valores:
        1. activo
        2. borrado

    Esta clase contiene estos métodos:

    - `void addCampo()` - añade un campo al registro siempre que el `tam` del registro lo permita.
    - `int getEstado()` - devuelve el estado actual de un registro.
    - `void setEstado()` - cambia el estado de un registro (de activo a borrado, y viceversa).

- Clase Campo. Contiene los siguientes atributos:
    - `int long`, la longitud del campo.
    - `int tipo`, un bit de 5 posibles valores:
        1. lógico
        2. carácter
        3. entero
        4. real
        5. cadena de caracteres
    - El contenido del campo, con un espacio determinado por `long`.

    Esta clase contiene estos métodos:

    - `void setContenido(string cont)`
    - `string getContenido()`

    Estos métodos se implementan con cadenas de caracteres porque su tipo ya está definido en el bit `tipo`, se harían las conversiones adecuadas en caso de ser necesarias.

3. Nivel interno

Lectura y escritura a disco

Los métodos que emplean L/E a disco son:

- `addReg()` - Lectura y escritura. Se leen `Bloque.tam` Bytes para las comprobaciones de tamaño y posicionamiento. Se escriben `Registro.tam` Bytes.
- `addBloque()` - Escritura. Se escriben `Bloque.tam` Bytes.

- `fin()` - Lectura. Si tenemos N entradas, se leen `N-posición * Registro.tam` Bytes, desde la posición actual hasta el final.
- `inicio()` - Lectura. En el peor caso (todos los registros borrados), se leen `N * Registro.tam` Bytes.
- `sig()` - Lectura. Se leen `N-fin * Registro.tam` Bytes. No hay que olvidar las lecturas que estamos haciendo también de `fin()`.
- `listar()` - Lectura. Se leen `pos_fin - pos_ini * Registro.tam` Bytes.
- `listarCompleto()` - Lectura. Se leen `Archivo.tam` Bytes.

- `sigRegActivo()` - Lectura. `N-fin * Registro.tam` Bytes.
- `hayEspacio()` - Lectura. Se leen `Bloque.tam - (Bloque.tam - posicion * Registro.tam)` Bytes.
- `addCampo()` - Lectura y escritura. Se leen `Bloque.tam - (Bloque.tam - posicion * Registro.tam)` Bytes. Se escriben `Campo.long` Bytes.

- *setters* (Estado, Campo) - Lectura y escritura. Se lee el `Bloque.tam` para el Estado, y el `Registro.tam` para el Campo. Se escriben `Estado.tam` y `Campo.long` Bytes respectivamente.
- *getters* (Estado, Campo) -Lectura. Se leen `Estado.tam` y `Campo.long` Bytes respectivamente.

Apertura y cierre del fichero

La apertura y cierre del fichero se manejará desde un main adaptado para la lectura del archivo. Obviamente, la apertura de un fichero secuencial físico implica su lectura completa desde el disco.

Desplazamiento

Los métodos que requieren desplazamiento en el disco son los siguientes:

- `setPos()` - La dirección puede ser cualquiera, nos devolverá mediante un bool si es hacia delante (0) o hacia atrás (1). Avanza en términos de `Registro.tam` Bytes.

- `addReg()` - Hacia delante. Avanza `Registro.tam` Bytes.
- `addBloque()` - Hacia delante. Avanza `Bloque.tam` Bytes.

- `fin()` - Hacia delante y hacia atrás. Avanza `N-posicion * Registro.tam` Bytes (alcanzar el final de la lista de Registros, estén activos o no), y luego retrocede `P * Registro.tam` Bytes, donde P es el número de registros borrados desde el último activo hasta el final del bloque.
- `inicio()` - Hacia atrás. Retrocede `posicion * Registro.tam` Bytes.
- `sig()` - Hacia delante. Avanza `Q * Registro.tam` Bytes, donde Q es el número de registros borrados por los que ha pasado hasta encontrar el activo.
- `sigRegActivo()` - Hacia delante. Avanza `Q * Registro.tam` Bytes, donde Q es el número de registros borrados por lo que ha pasado hasta encontrar el activo.

Una parte de la gestión del desplazamiento se desarrolla en la función `setPos()`, que no solo establece la posición sino que nos permite conocer si se ha podido simplemente avanzar en la posición o si por el contrario hemos tenido que dar la vuelta desde el principio.