Entrega 2 PROP

Q1 2022-23

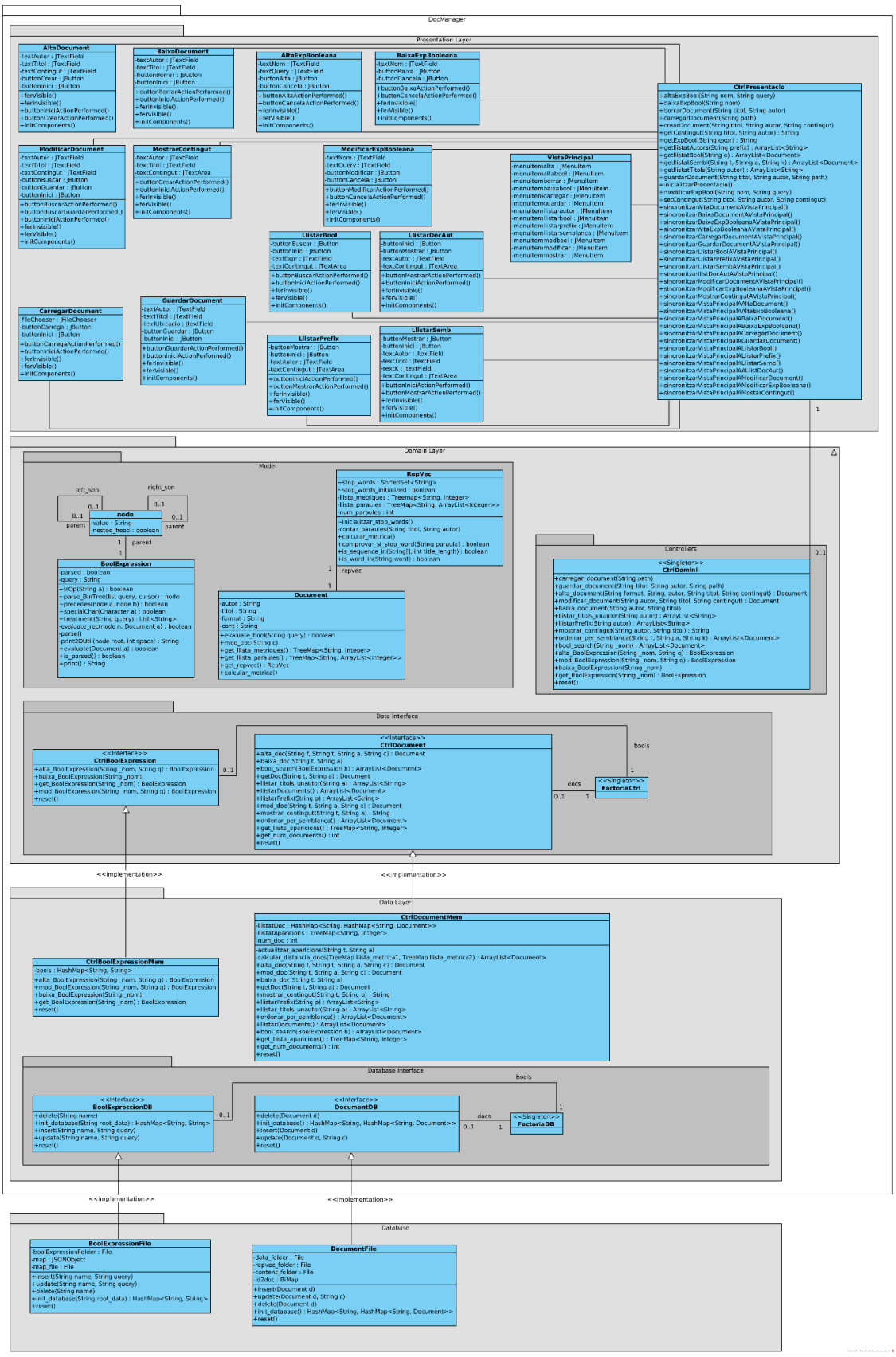
Integrants:

Lucas Ares, Miquel Muñoz, Carles Pedrals i Pau Cuscó.

Índex

1. Diagrama UML de tota l'aplicació i de cada capa

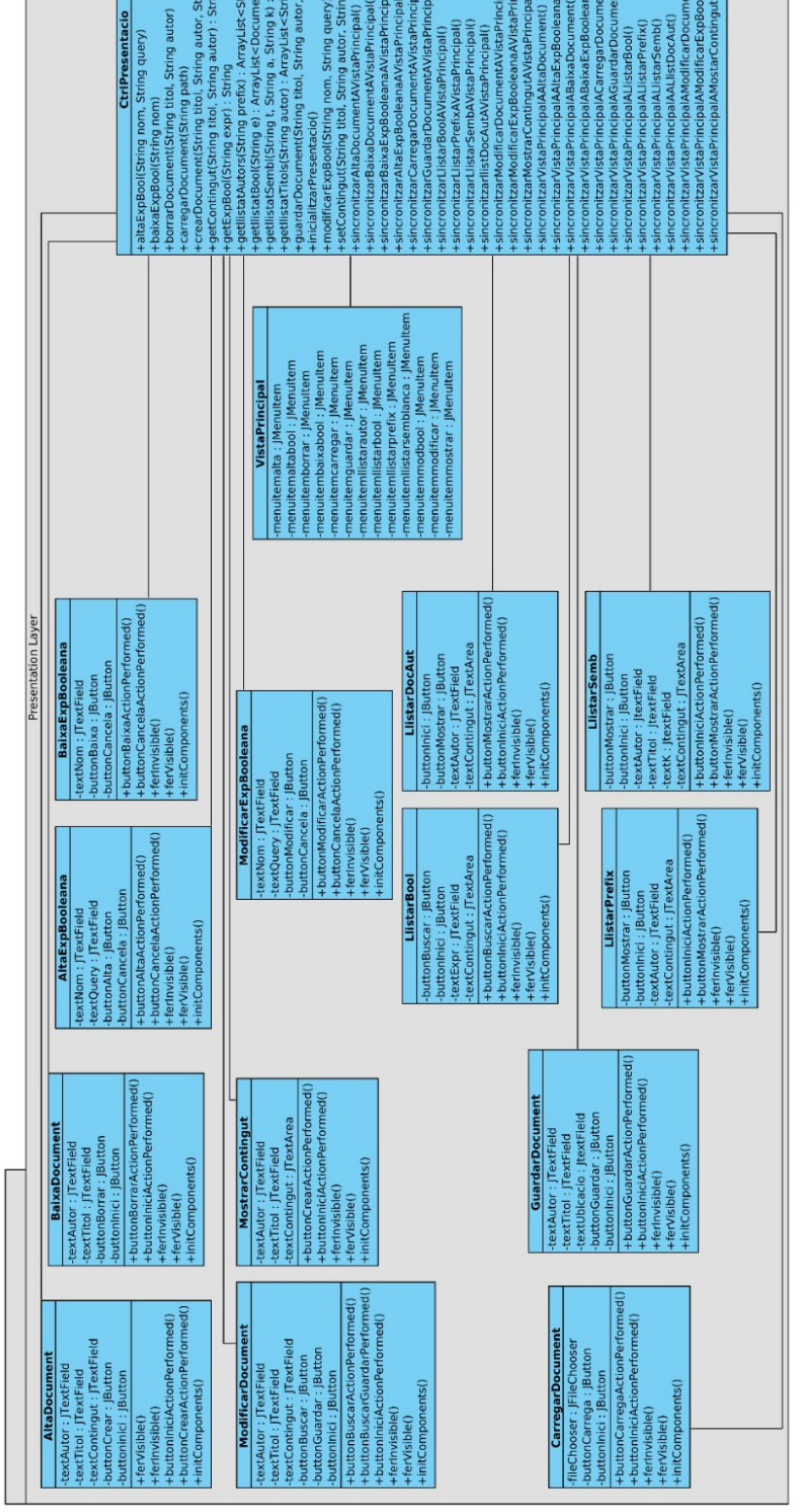
Diagrama UML: (Imatge disponible a "WholeApp.png" en alta resolució per fer zoom)



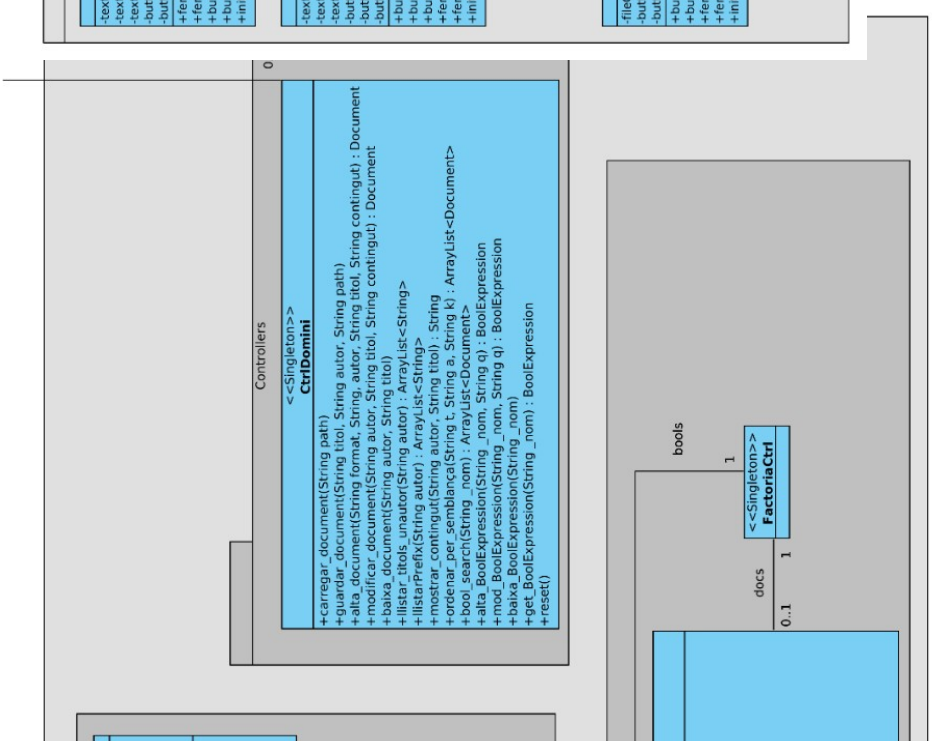
1.1 PresentationLayer (Capa de Presentació):

El propòsit d'aquesta capa és organitzar les diferents vistes de l'aplicació, i controlar la seva interacció amb l'usuari.

Emprant una estructura de desacoblament mitjançant un controlador general de la capa, aconseguim separar la dependencia de cada vista amb la capa de domini (el que seria el sistema funcional) respecte la seva interfície.

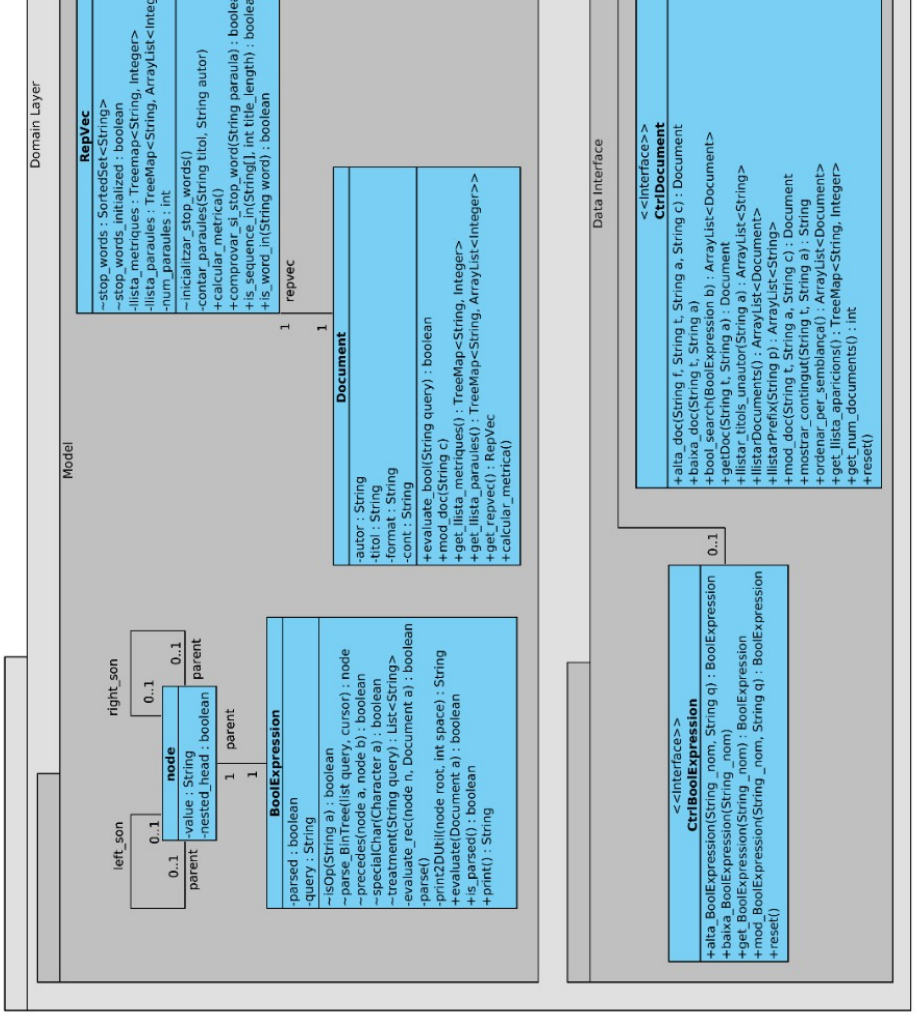


1.2 DomainLayer (Capa de Domini):



Aquesta és la capa on es conce
lògica de l'aplicació. Es presenta
controlador de capa però aquest
ser estès a varis controladors
diferents funcionalitats.

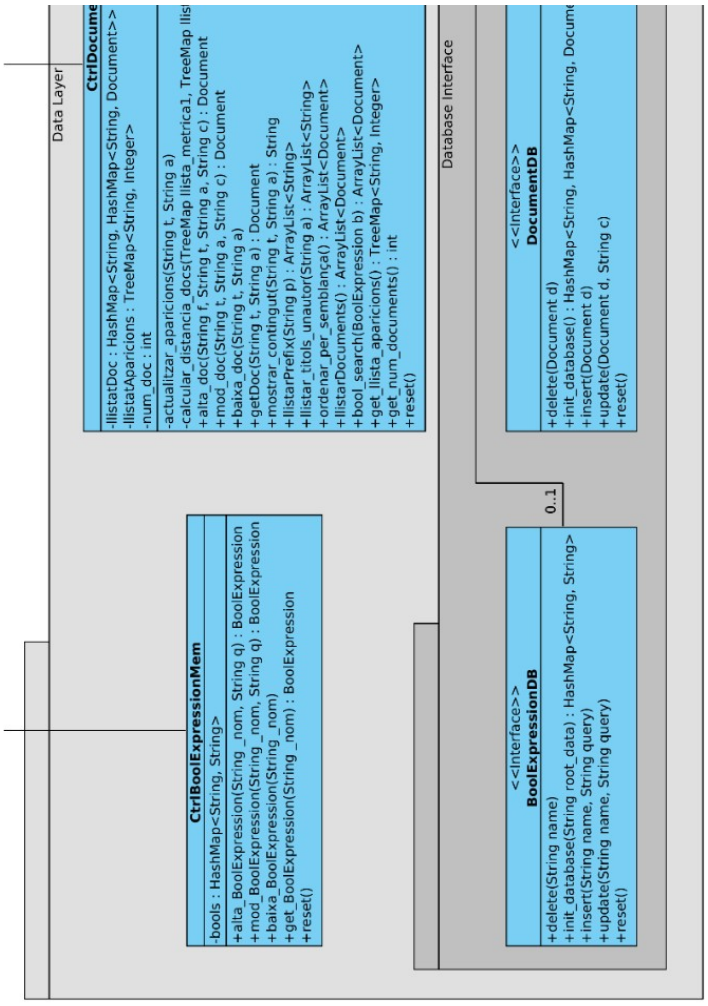
La FactoriaCtrl permet a qu
d'aquestes accedir a les instànci
objectes guardats a m
proporcionant una interficie
desvincular la memoria respe
model.



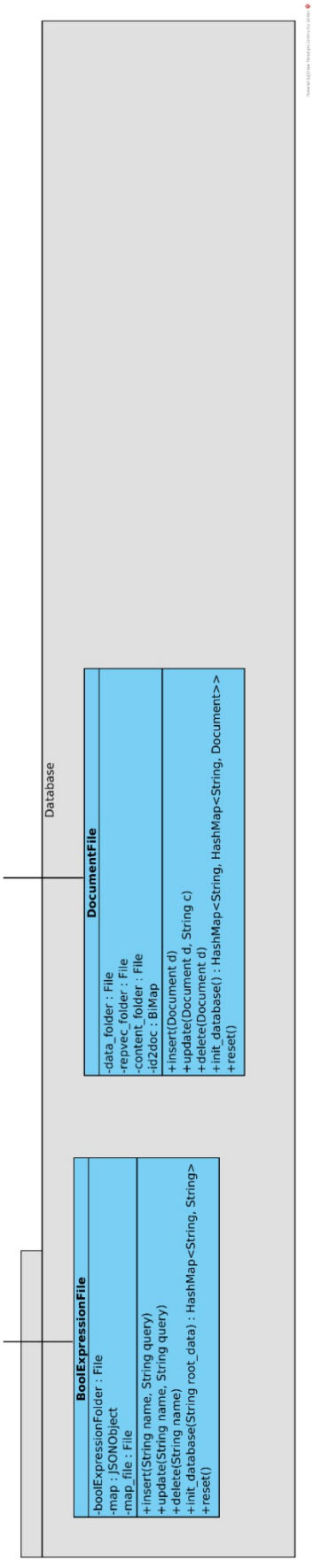
1.3 DataLayer (Capa de Ges
Persistencia):

La funcionalitat d'aquesta capa és, per una banda, actuar de contenidor dels objectes de la capa de domini que són guardats a memoria principal (classes CtrlDocumentMem i CtrlBoolExpressionMem, cada una implementant per separat les interfícies de la anterior capa per afegir un nou nivell de desacoblament).

Aquestes, al seu torn, estan en sincronització amb la memòria auxiliar (el mecanisme de persistencia utilitzat, ja sigui base de dades o fitxers), mitjançant interficies per, novament, desvincular el mètode de persistencia respecte el control de memoria principal.

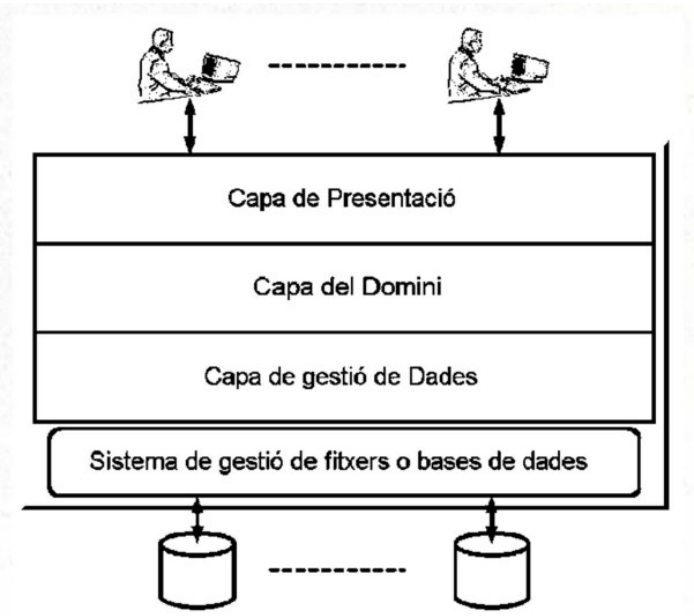


FactoriaDB permet a les classes anteriors entrar en contacte amb la base de dades de cada un dels objectes. D'aquesta manera, desacoblem el contacte, però necessitem d'una package que actuï com a base de dades per se.



1. 4 Persistence:

Aquest és el paquet que imita la instancia d'una base de dades on els nostres documents son guardats. S'ha optat en aquest projecte per utilitzar una base de dades basada en un sistema de fitxers. Aquesta és una decisió que després pot ser modificada gràcies al desacoblament de funcionalitats. Les classes d'aquesta capa han per la d'implementar les interfícies anteriorment definides correcta comunicació entre la memoria principal i auxiliar. Aquesta estructura segueix el model de 3 capes i amb aquest paquet repsetentem la estructura inferior (Sistema de gestió de fitxers o base de dades) del diagrama següent:



2. Descripció detallada de totes les classes rellevants

La composició concreta de cada classe es pot observar en el Javadoc proporcionat, aquest document farà una mera síntesi de les classes i es centra més en les estructures i algoritmes de les mateixes.

Document:

Representa el significat conceptual d'un document, i serveix per manipular el contingut del propi document i dur a terme les diverses cerques que proporciona la aplicació. Títol i autor que conformen la clau externa, i per tant, definiran inequivocament un sol document.

Aquesta classe presenta una optimització respecte el contingut del Document. Es coneix que el contingut d'un Document és la part més llarga (i, per tant, més costos de guardar a memoria). Per un document ja guardat en una sessió anterior de l'aplicació, no es carregarà el seu contingut a memoria si aquest no és consultat de manera explícita mitjançant l'operació de mostrar contingut. Tota la informació necessària per les búsquedes de Documents no es troba en aquesta classe (no depèn del objecte contingut), fent possible la descrita optimització

L'explicació de les estructures de dades utilitzades per fer la cerca està més desenvolupada al punt 3.

BoolExpression:

Aquesta classe encapsula la informació i els mètodes relacionats amb les expressions booleanes del sistema. Aquestes expressions booleanes son conjuncions '&' i disjuncions '|' d'expressions d'altres expressions booleanes parentitzades amb '(' , ')' i de booleans literals (són les unitats atòmiques de les nostres expressions, i son avaluades directament). Aquests literals podran ser sets '{' , '}' (successions NO ordenades de paraules separades per espais les quals es busquen en un Document i totes han d'aparèixer per fer el literal

cert), o seqüències ‘ “ ‘ , ‘ “ ‘ (successions ordenades de paraules separades per espais les quals es busquen en un Document i totes han d'aparèixer en el ordre descrit per fer cert el literal). Més informació de com es representa aquesta estructura pot ser trobada a la descripció d’estructures de dades

RepVec:

Conté tota la informació necessària per representar un document sense necessitat de tenir el seu contingut a memoria. Disposa d’una estructura mapejada que vincula cada paraula que apareix al document amb una llista d’aparicions indexades. Permet resoldre totes les búsquedes de documents i es carregada a memòria per tot Document de l’aplicació.

3. Descripció de les estructures de dades i dels algoritmes

3.1 Estructures de dades de RepVec.java

llista_paraules (TreeMap<String,Integer>): un *map* on s’emmagatzemen totes les paraules del contigut del document i del títol separades i si alguna es repeteix només surt una vegada i el valor de l’entrada equival a les vegades que apareix. El treemap té com a cost O(logN) per funcions d’inserció, esborrar i consulta.

llista_metriques (TreeMap<String,Double>): un *map* amb totes les paraules que no coincideixen amb cap *stop word* i la mètrica tf-idf de cadascuna. El treemap té com a cost O(logN) per funcions d’inserció ,esborrar i consulta.

stop_words (SortedSet<String>): un set amb totes les *stop words* ordenades alfabèticament. El SortedSet té com a cost O(logN) per funcions d’inserció, esborrar i consulta.

3.2 Estructures de dades de CtrlDocumentMem.java

llistatDoc (HashMap<String,HashMap<String,Document>>): un *map* on la clau del mapa extern és l’autor del document i el valor és un mapa “intern” on la clau és el títol del document i valor és l’objecte Document. S’ha decidit escollir aquesta estructura de dades ja que la complexitat en temps de cerca és O(1). No necessitem tenir els elements ordenats.

Els costos de Separate Chaining són els que utilitza Java. Són els següents:

	Millor cas complexitat temps	Cas mitjà complexitat temps	Pitjor cas complexitat temps
Cerca	O(1)	O(1)	O(n)
Inserció	O(1)	O(1)	O(n)
Esborrar	O(1)	O(1)	O(n)
Complexitat espai	O(m+n)	O(m+n)	O(m+n)

On 'm' és la mida de la Hash Table i 'n' són el nombre de items inserits.

"num_doc" ens indica fàcilment quants documents tenim a memòria amb complexitat en temps O(1) ja que sinó hauríem de recórrer cada mapa de mapes per anar sumant quants documents hi ha en total amb cost *n* (sigui *n* el nombre d'autors).

"llistatDoc" i "num_doc" no s'han millorat des de la primera entrega ja que s'han considerat suficientment eficients.

llistaAparicions (TreeMap<String,Integer>): un *map* ordenat on el primer valor és una paraula que apareix en algun dels documents i la segona en quants documents surt. El treemap té com a cost O(logN) per funcions d'inserció, esborrar i consulta.

3.3 Algoritme de llistar documents per semblança

Per calcular la distància entre dos documents, hem fet servir la suma de tf-idf que farem servir per a calcular les mètriques de cada paraula, i el model d'espai vectorial que sabent aquestes mètriques farem servir per determinar la semblança entre els documents.

3.3.1 tf-idf

La mètrica tf-idf bàsicament ens serveix per indicar la freqüència d'una paraula en un conjunt de documents. Està format per dues parts, *term frequency* i *inverse term frequency.*
La primera part ens indica bàsicament quantes vegades apareix una paraula en un document entre quantes paraules hi ha en un document. Per calcular això necessitarem estructura amb totes les paraules del document i quantes vegades apareix, això està emmagatzemat a **llista_paraules**.

$$\mathrm{tf}(t,d) = \frac{f_{t,d}}{\sum_{t' \in d} f_{t',d}}$$

La segona part és la freqüència inversa d'un terme en el conjunt de documents i serveix per descartar els termes més fets servir com conjuncions o preposicions (i, a, o…) que tot i aparèixer moltes vegades no aporten cap informació. Ho calculem amb la següent fórmula,

$$\mathrm{idf}(t,D) = \log \frac{N}{|\{d \in D : t \in d\}|}$$

on N és el número de documents en el conjunt de documents i {d ∈ D : t ∈ d} el número de documents del conjunt on la paraula *t* apareix. Necessitarem doncs saber quants documents tenim (**num_doc** a CtrlDocumentMem), i en quants documents apareix una paraula (**llistaAparicions** a CtrlDocumentMem) que és un map amb cada paraula i el número de documents en que apareix.

Els valors del tf i idf els multipliquem i guardem aquest valor a **llista_metriques**, un map on el primer valor és la paraula i el segon la seva mètrica, n'hi ha un per a cada document.

$$\mathrm{tfidf}(t,d,D) = \mathrm{tf}(t,d) \cdot \mathrm{idf}(t,D)$$

Finalment, quan ja tenim totes les mètriques de totes les paraules de tots els documents (que varien a cada modificació, inserció o borrat de document) i el que farem serà comparar-los entre ells.

3.3.2 Model d'espai vectorial

El model d'espai vectorial és una manera d'expressar diferents documents en un espai multidimensional on els documents són cada punt i les seves dimensions són les paraules i la seva mètrica. Així, podem anar situant tots els documents en un espai i per veure quins s'apropen més o menys entre ells. Per calcular la distància farem servir la seguent fórmula:

$$\cos(d_j, q) = \frac{\mathbf{d_j} \cdot \mathbf{q}}{\|\mathbf{d_j}\| \, \|\mathbf{q}\|} = \frac{\sum_{i=1}^{N} w_{i,j} w_{i,q}}{\sqrt{\sum_{i=1}^{N} w_{i,j}^2} \sqrt{\sum_{i=1}^{N} w_{i,q}^2}}$$

on, el la part superior és el sumatori de cada una de les mètriques tf-idf dels dos documents que comparem multiplicades i la part inferior la norma del vector que forma cada un dels dos documents que comparem. Els valors de les distàncies els guardem en el map i posteriorment els ordenem per saber quins documents són més semblants a l'inicial.

3.4 Estructures de dades de BoolExpresion.java

La classe BoolExpression representa la expressió booleana mitjançant un Abstract Syntax Tree (AST) amb una estructura de nodes recursiva. Aquesta definició permet respectar les

precedències. S'utilitza una funció de treatment que comprova que el input sigui parseable i un parser que converteix la llista de tokens extreta a un AST. Aquest és després avaluat en un recorregut recursiu.

3.5 Ús d'expressions booleanes

La anterior estructura permet avaluar les expressions booleanes i, en un futur, simplificar-les, ja que el nostre software "compren" aquestes expressió i en desglosa la seva estructura (que pot ser vista en la interfície d'usuari).

Per fer més eficient la búsqueda, s'aprofita que tota la informació necessària del Document es troba en el seu objecte RepVec (concretament en la llista de paraules) que conté per cada paraula una llista de índexs on ha sigut trobada la paraula. Agafant la longitud de la llista sabem el nombre de aparicions.

En la avaluació d'expressions booleanes existeixen dues casuístiques:

- Set:
  Un set consisteix en una expressió encapsulada per claus «{» i «}» que conté una serie NO ordenada de paraules vàlides separades per espais. Es busca trobar únicament si aquestes paraules apareixen al Document

- Seqüència:
  Un seqüència consisteix en una expressió encapsulada per claus «"» i «"» que conté una serie ordenada de paraules vàlides separades per espais. Es busca trobar si aquesta seqüència existeix en arxiu en qüestió.

Quant a l'avaluació de sets, l'algoritme que es presenta és el següent:

```
/**
 * Check if the word it's in llista_paraules
 * @param word Word to check
 * @return Returns true if word its in llista_paraules
 */
public boolean is_word_in(String word) {
    if(word.equals("")) return true;
    return (!word.equals("") && llista_paraules.get(word.toLowerCase()) != null && llista_paraules.get(word.toLowerCase()).size() != 0);
}
```

Sabem que un accés a un Treemap te cost log(n) sent “n” el nombre de paraules diferents en un Document. Per tant, si el set consta de “k” paraules, tindrem un cost total de k*log(n)

Per altra banda, en l'avaluació de seqüència, l’algoritme que es presenta és el següent:

```java
/**
 * Check if a literal sequence is in the Document
 * @pre The words on subquery are valid words, not empty spaces
 * @param subquery Array of words to check
 * @param title_length Lenght of the title of the document
 * @return returns whether the sequence is in or not
 */
public boolean is_sequence_in(String[] subquery, int title_length) {
    // Find sequence of words
    boolean result = false; // Better to set it initially false
    if(subquery.length == 0 || (subquery.length == 1 && subquery[0].equals(""))) return true;

    int[] pointers = new int[subquery.length]; // array of indexes initialized to 0 (ensured by java documentation)
    ArrayList<ArrayList<Integer>> aparicions = new ArrayList<ArrayList<Integer>>();
    for(int k = 0; k < subquery.length; ++k) {
        ArrayList<Integer> aparicions_per_paraula_k = llista_paraules.get(subquery[k].toLowerCase());
        if(aparicions_per_paraula_k == null || aparicions_per_paraula_k.size() == 0) {
            // stops when one of the words has exceeded the number of appearances
            return false; // If that word doesn't appear, no way there is a seq. with that word
        }
        if(subquery.length == 1) return true; // It is the equivalent of a sequence of one word, if we reached this point it means it exists
        aparicions.add(aparicions_per_paraula_k); // array[0...subquery.length-1][0...num_aparicions-1]
    }

    while(pointers[0] < aparicions.get(0).size() && !result) {
        // While there is still instances of the first word to see
        int app_of_prev_word = aparicions.get(0).get(pointers[0]);
        int i = 1; // Begin checking next word
        while(i < subquery.length) {
            int app_of_next_word = aparicions.get(i).get(pointers[i]);

            while(pointers[i] < aparicions.get(i).size() && app_of_next_word <= app_of_prev_word) {
                app_of_next_word = aparicions.get(i).get(pointers[i]);
                pointers[i]++;
            }
            if(pointers[i] == aparicions.get(i).size() && (app_of_next_word <= app_of_prev_word || app_of_next_word - app_of_prev_word > 1)) {
                //System.out.println("One of the words indexes has been exceeded: " + Integer.toString(i));
                return false; // stops when one of the words has exceeded the number of appearances
            }
            // If this line is reached, means we have found an instance of the next word posterior to the previous word
            if(app_of_next_word-app_of_prev_word == 1) { // If they are consecutive, jump no next word
                i++;
                if(i == subquery.length) result = true; // That means we found the sequence!!!! :)
                app_of_prev_word = app_of_next_word; // get ready for next iteration
            }
            else break; // We need to increment the first index, no matter how far we got
        }
        pointers[0]++;
    }
```

Aquest algoritme ha estat fet considerant els següents factors:

- El contingut pot ser que estigui en memoria (ja que per optimitzar espai, el contingut no es carregat si no ha sigut consultat especificament prèviament). Tot i així. un recorregut sobre el Document sencer suposaria un cost “m” (sent el nombre de paraules de tot el Document) on “m >> n” (ja que moltes paraules en un Document acaben sent repetides). Un cost “m” suposaria un cost molt elevat

- Tenint en compte la llista amb els índexs on apareix cada paraula, podríem agafar aquesta llista i, per cada element seu, fer un recorregut sobre les altres k-1 llistes buscant que existeixi una seqüència consecutiva d'indexs. El cost d’un algoritme així, sense cap optimització, es planteja el pitjor cas com un producte:

  (n_1 * … * n_k) on n_i es el nombre de vegades que apareix la paraula i-èssima de la seqüència. aquestes es un cost exponencial si suposem que una paraula apareix “a” vegades de mitjana (cost a^k). Fent aquest algoritme bastant ineficient.

Per tant, l'algoritme escollit planteja una llista de punters (k punters, un per cada paraula de la seqüencia) els quals cada un d'ells apunta al element de la llista i-èssima que ha estat consultat per última vegada. cada un d'ells itera sobre la llista i a la que troba un índex de aparició de la paraula i-èssima que es superior al anterior, és a dir:

"aparicions_paraula_anterior(pointer[i-1]) < aparicions_paraula_actual(pointer[i])"

significa que hem trobat una instancia posterior de la següent paraula de la seqüència, fet que implica que potser existeix aquesta seqüència. Si la diferència entre els dos valors és 1, significa que de moment és certa la seqüència i procedim a avaluar la següent paraula ("i" respecte "i+1" amb el mateix procés). Si aquesta diferència és major que 1, s'ha trencat de la seqüència i hem de tornar a la llista de aparicions de la primera paraula (hem de començar de nou) per gràcies al vector de punters, no accediran a valors d'indexs prèviament ja consultats (punter[i] mai es redueix per cap "i", ja que no tindria sentit) aconseguint un cost, en el pitjor dels casos (n_1 + … + n_k) que comparant-ho amb la implementació exponencial equivaldria a un cost a*k (suposem que una paraula apareix "a" vegades de mitjana).

4. Llibreries externes utilitzades

Com a mesura per facilitar la transformació del document a un format propietari, s'empra la llibreria "json-simple" (https://code.google.com/archive/p/json-simple/) la qual permet representar objectes en estructura JSON. Aquestes estructures son després utilitzades per representar en fitxers els Documents i els seus components. Es fa ús de les següents classes i mètodes:

- JSONObject:

  Estructura implementada com un map de String a objecte que permet representar la imbricació dels objectes JSON

- JSONParser:

  Aquesta classe permet llegir un document del sistema i transformar-lo a un JSONObject que posteriorment és utilitzat per extreure la informació.