Excellent

Descripció EDA i implementació

Rubén Aciego
Jofre Costa
Mariona Jaramillo
Francesc Pifarré

Cela (i fills CelaNum, CelaText, CelaData, CelaRef)

Per a tal d'implementar la classe Cela no ha fet falta usar estructures de dades ni algorismes gaire complicats. Primer de tot, s'ha implementat la classe Cela com a abstracta, i les seves 4 filles CelaNum, CelaText, CelaData i CelaRef són les que han hagut d'implementar certs mètodes. La classe pare Cela inclou com a atributs una String per a l'input de l'usuari i una enumeració TipusCela per indicar de manera més fàcil amb quin tipus estem tractant. Les filles, a part de tenir aquests atributs, en tenen algun més per guardar el verdader valor de l'input de l'usuari: un double per a CelaNum, una altra String per a CelaText, un LocalDate per a CelaData i una Cela per CelaRef.

Pel que fa als mètodes, no hi ha molt a comentar ja que la gran majoria d'ells són getters i setters. Potser sí que fa falta mencionar, però, el mètode abstracte copy() i el compare(Cela):

- **copy()**: el mètode permet fer una Deep Copy de la Cela en qüestió (S'ha tingut en compte quins tipus són immutables)
- **compare(Cela):** fent ús dels mètodes compare() i compareTo() ja implementats pels diferents tipus de Java, aquest mètode permet comparar (i per tant ordenar) diferents obejctes Cela entre ells.

MatriuCeles

Per a d'implementar-la, hem fet ús de 3 atributs: dos *int* per guardar el número de files i columnes i un estructura de dades bastant curiosa per a implementar la matriu d'objectes Cela en sí. Aquesta estructura de dades és una ConcurrentSkipListMap de ConcurrentSkipListMaps d'objectes Cela (ambdós indexades per Integer). Al principi s'havia pensat d'usar TreeMap en comptes de ConcurrentSkipListMap, ja que els membres de l'equip era la que coneixíem més, però les següents raons van fer que la balança es decantés cap a l'altra banda:

- Degut a la seva implementació més complicada, TreeMap no permet l'addició (i amb penes i treballs l'eliminació) d'elements a l'estructura mentre s'itera sobre els elements d'aquesta (es llança ConcurrentModificationException). En canvi, usant ConcurrentSkipListMap no es té aquest problema, cosa que fa que la implementació dels mètodes sigui més fàcil i no s'hagi d'estar creant nous objectes sovint.
- Com el seu nom indica, les operacions principals de l'estructura es poden executar de forma concurrent per diversos threads, fet que podria ser útil en futures fases de desenvolupament de l'aplicació.

- L'estructura ConcurrentSkipListMap ofereix la mateixa funcionalitat que TreeMap (es podria dir que son dues implementacions diferents amb un únic objectiu o propòsit). Per aquest motiu, els elements que emmagatzemen estan ordenats segons la seva respectiva clau. Les operacions principals de l'estructura són, també, logarítmiques en mitjana en terme de complexitat. La complexitat de les mateixes operacions és moderadament millor en TreeMap (son sempre logarítmiques) però, en el nostre cas, la complexitat d'aquestes en ConcurrentSkipListMap ja és suficientment.

Pel que fa als mètodes, sobre la constructora i els getters i setters no hi ha molt a dir: com a molt s'ha de comentar que algunes fan ús de les operacions principals de ConcurrentSkipListMap, però adaptades al fet que n'estem usant una de dues dimensions (cal mencionar que la matriu que estem implementant està indexada primer per columnes i després per files, ja que per algunes operacions que s'havien de fer amb aquesta era molt més còmode).
Hi ha uns quants mètodes, però, que sí que paga la pena mencionar ja que són un pèl més complexos, però tots usen la mateixa idea. Els mètodes en qüestió són eliminaFila, eliminaColumna, getBloc, getEntrades i getEntradesColumna, i la idea que usen és la següent: mitjançant operacions pròpies de ConcurrentSkipListMap que permeten obtenir un subgrup d'entrades (tailMap, subMap…), s'itera (amb iteradors o ForEach) sobre aquests subgrups per tal d'obtenir totes les entrades que s'estan buscant i sense haver de mirar entrades innecessàries.

Full

Es tracta d'una classe que hereda de MatriuCeles. Com ja s'ha descrit MatriuCeles i Full no se'n diferencia gaire, no hi ha molt a comentar respecte l'anterior. Els mètodes principals de Full (buidaBloc, copiaBloc i mouBloc) usen la mateixa idea que les de MatriuCeles: s'itera sobre un grup reduït d'entrades gràcies a l'ús d'operacions pròpies de ConcurrentSkipListMap.
Potser la principal diferència amb la classe pare és l'atribut celaResultat, que es tracta d'una Cela especial que hi haurà per cada Full i, per referenciar-la, s'han d'usar els índexs (-1, -1). Per a implementar aquesta idea es podia haver afegit una entrada a la posició (-1, -1) de la matriu, però es va decidir que usant la primera manera era més coherent i llegible.

EntradaMatriuCeles

Res a mencionar d'aquesta classe, ja que només està composta per getters i setters (està pensada per ser com un "struct" de C o C++).

Controlador Full

Es tracta d'una classe que actua de intermediari entre el Full i les Operacions que s'han de fer sobre el mateix i les Cel·les que conté. Rep l'operació ja parsejada del Controlador Domini i l'executa. En cas que l'operació sigui sobre una MatriuCeles la delega a Operador i en cas que sigui directament sobre full crida els mètodes de full adients.

Té com a atributs el Full que controla i l'Operador a que delegar les Operacions sobre Matrius Cela. Per saber a qui ha de delegar l'operació, analitza el ResultatParserFull que li passa el Controlador Domini al critar el metode executaOperacio().

Al tractar-se d'un operador, no té getters ni setters i més enllà del mètode executaOperació nomes conté algun mètode privat que es crida des d'executaOperació segregat d'aquest per a facilitar la lectura del codi.

Document

La classe Document actua com un vector d'elements de la classe Full amb un nom i una data de modificació. Les seves operacions permeten afegir fulls, obtenir un full en un índex concret i eliminar un full en un índex concret. Quan l'índex del full no sigui vàlid (menor a 0 o major igual que el nombre de fulls) els mètodes llencen excepcions d'índex de full. L'operació d'afegir full té un cost constant amortitzat, l'obtenció d'un full té un cost constant i l'eliminació d'un full té un cost lineal en el nombre de fulls.

Operador

Classe amb estructura de singleton que executa les operacions sobre MatriuCeles que li delega un Controlador Full. Conte un mètode per cada una de les Operacions implementades, que rep com a paràmetre una MatriuCeles sobre la qual aplicar la Operació i retorna com a resultat un altra fruit d'aplicar l'operació a la MatriuCeles d'entrada. Algun dels mètodes associats a operacions a executar reben també paràmetres extra d'entrada per complementar la informació necessaria per implementa la seva funcionalitat (Ex.: string a cercar, xifra a la qual truncar un número…).

Els mètodes, exceptuant alguns que serien cerca, ordena i algunes de les operacions estadístiques d'executa operació estadística, iteren sobre totes les cel·les de MatriuCeles i,

segons si el tipus és el correcte per aplicar l'operació, s'apliquen i deixen enregistrat en el paràmetre InputUsuari de la Cel·la que aquesta operació ha estat executada a mode d'històric. Si el tipus no és l'adequat deixen la cel·la sense modificar i es retorna en el mateix estat en que havia arribat.
El cost de tots aquests mètodes que iteren sobre els elements de la MatriuCeles és, per tant, lineal sobre el nombre d'elements no buits de la MatriuCeles.
S'han implementat també alguns mètodes privats per ajudar a la lectura del codi dels mètodes públics. El seu codi podria trobar-se en els mètodes públics pero per facilitar la lectura i enteniment s'han separat en privats.
S'utilitza com a llibreria per facilitar càlculs matemàtics de operacions aritmètiques i estadístiques el paquet java.lang.Math. Per a al cerca s'utilitza també la llibreria json-lib per poder expressar tota la informació dels resultats de la cerca en format JSON.
El format de retorn de l'operació cerca és un JSON amb un atribut "ocurrencies" que conté un enter amb el nombre d'ocurrències de l'string cercat i, per a cada cel·la textual del bloc on s'opera, un atribut "fila:columna" que alhora és un objecte JSON amb dos atributs, "ocurrencies" amb el nombre d'ocurrències en aquella cel·la en particular i "indexs", un vector d'enters que conté les posicions de les ocurrències a la cadena de text de la cel·la.

L'operació ordena de bloc aplica el següent algorisme: primer obté un vector d'EntradaMatriuCeles format pels elements de la columna del bloc respecte la que s'ordena. Acte seguit ordena (en ordre ascendent o descendent segons el criteri triat) els elements d'aquest vector fixant-se exclusivament en la cel·la dins de cada EntradaMatriuCeles. L'ordre en el que s'ordenen les cel·les de tipus diferents és el marcat per l'enumeració Cela.TipusCela, que és el següent: numèrica < textual < datada. Els tipus de cel·la igual s'ordenen segons el seu propi criteri (el marcat pel seu valor). L'ordenació de les dades té un cost O(nlogn) on n és el nombre de files (amb entrada no buida) en la columna respecte la que ordenem. Un cop tenim les dades ordenades, com pot ser que no totes les cel·les de la columna tinguéssin valor, el que fem és recórrer totes les files i marcar les que tenien valor (i per tant hem ordenat) en l'ordre corresponent, les files sense valor s'afegeixen en ordre ascendent a les ordenades. Finalment es crea la MatriuCeles resultat amb les mateixes cel·les que la inicial i mantenint la posició de les columnes de cadascuna però permutant les cel·les segons el nou ordre descrit.
La resta d'operacions que realitza l'operador tenen un cost lineal en el nombre de cel·les no buides del bloc d'entrada (doncs hem d'operar cada cel·la) multiplicat pel cost innherent de

l'operació que estem fent. Per exemple les operacions aritmètiques tenen cost constant, les cerques i reemplaçaments tenen cost lineal en la longitud del text i les operacions sobre dates tenen cost constant.

DocumentParser i DocumentConverter

Aquestes classes són les encarregades de les conversions d'instàncies de la classe Document als formats suportats per desar les dades (CSV i JSON) i viceversa.

El format de les dades en el cas de CSV és el següent:

- Primera línia: "dataModificació", "numFulls"
- Per a cada full del document una línia: "fulli", "numFiles", "numCols" on i fa referència a l'i-èssim full
- Després de cada línia on s'especifica un full, s'especifiquen les cel·les que pertanyen a aquest full donat de la següent forma:
  "fila:col", "inputUsuari", "type", "value" on type és el tipus de cel·la (enumeració Cela.TipusCela) i value és el valor depenent en el tipus de cel·la

El format de les dades en el cas de JSON és el següent:

- JSON del document (el global): {"dataModificacio": dataStr, "numFulls": num, "full0": JSONfull0, …, "fulln": JSONfulln } on JSONfulli és un objecte JSON del full i-èssim
- JSON de cada full: { "numFiles": numF, "numCols": numC, "i:j": JSONcelai_j } on JSONcelai_j és un objecte JSON de la cel·la i:j
- JSON de cada cel·la: { "inputUsuari": str, "type": tipus, "value": value } on tipus és del tipus enumeració Cela.TipusCela i value és el valor depenent del tipus

La conversió i el parsing de les dades és bastant senzilla gràcies al format triat i les llibreries utilitzades, una idea que cal esmentar però és el cas de les cel·les referencials a l'hora de fer el parsing. El problema ve donat que potser just quan estem fent parsing d'una cel·la referència determinada, la cel·la a la qual fa referència encara no s'ha afegit al full que s'està creant. Per solucionar aquest problema el que fem és, cada cop que trobem una cel·la referencial, la guardem en un vector per afegir-la al final. Un cop hem llegit totes les cel·les del full actual és quan afegim les cel·les referencials que havíem guardat, de manera que ara

sí tindrem les cel·les a les que fan referència dins el full (doncs no es permeten referències a referències).

Parser

El Parser s'encarrega de transformar el vector d'strings codificats provinents de la capa de presentació/vista en instàncies de les classes ResultatParserFull/Document que contenen dades útils per poder operar amb comoditat dins la capa de domini. A continuació es detalla el sistema de codificació que hem fet servir.

Si el primer string codificat del vector comença per OPERACIO_DOCUMENT, aleshores estem davant d'una operació relacionada amb l'entorn del document i retornarà una instància de la classe ResultatParserDocument. Aleshores, el primer element de l'array és sempre no buit i conté la informació general, amb diversos camps separats entre comes. El primer camp és sempre OPERACIO_DOCUMENT. En funció del segon camp (tipus d'operació) es parseja d'una manera o d'una altra:

- si tipusOperació és CREA_DOCUMENT o CARREGA_DOCUMENT, el segon element de l'array conté el nom del document creat o del document carregat.
- si tipusOperació és ELIMINA_FULL, s'afegeix un camp extra al final que és el idFull a eliminar.
- si tipusOperació és AFEGEIX_FULL, DESA_DOCUMENT O TANCA_DOCUMENT, no hi ha camps extres.

Si el primer string codificat del vector no comença per OPERACIO_DOCUMENT, aleshores estem davant d'una operació relacionada amb l'entorn de full i retornarà una instància de la classe ResultatParserFull. Aleshores, el primer element de l'array és sempre no buit i conté la informació general, amb diversos camps separats entre comes. En funció del primer camp (tipus d'operació) es parseja d'una manera o d'una altra. Tot i així, els primers 8 camps coincideixen per totes les operacions, aquests són: tipusOperació,idFull,filaOrigen,colOrigen,numFiles,numCols,filaDestí,colDestí

A més,

- si tipusOperació és OPERACIO_ARITMETICA, s'afegeix un camp extra al final que és el tipusOperacióAritmètica
- si tipusOperació és OPERACIO_ESTADISTICA, s'afegeix un camp extra al final que és el tipusOperacióEstadística

- si tipusOperació és CONVERSIÓ_UNITATS, s'afegeix un camp extra al final que és el tipusConversióUnitats
- si tipusOperació és ORDENA, s'afegeixen dos camps extres al final que són tipusCriteriOrdenació i columnaOrdenació
- si tipusOperació és TRUNCA_NÚMERO, s'afegeix un camp extra al final que són els dígitsTruncar
- si tipusOperació és OPERACIO_FULL, s'afegeix un camp extra al final que és el tipusOperacióFull i:
    - si tipusOperacióFull és CERCA_OCURRENCIES, el segon element de l'array conté l'stringCercada
    - si tipusOperacióFull és REEMPLACA, el segon element de l'array conté l'stringCercada i el tercer l'stringReemplaçadora
    - si tipusOperacióFull és MODIFICA_CELA, el segon element de l'array conté l'inputUsuari
    - si tipusOperacióFull és ELIMINA_FILA o ELIMINA_COLUMNA, s'afegeix un camp que és la filaColEliminar (a més, destacar que del segon al vuitè camp, en aquesta operació, són buits).

ControladorVista, WindowPrincipal i la resta de classes corresponents a vistes diferents a la principal

En tema algorísmic, no hi ha molt a comentar respecte a aquestes classes: el funcionament de ControladorVista és bàsicament de passar i rebre missatges de la capa de domini i fer les pertinents actualitzacions a la capa de presentació. Totes o gairebé totes les funcions que conté tenen aquest funcionament explicat: passen un missatge codificat de certa manera cap el domini i se li retorna la informació actualitzada del que ha demanat. Aquesta informació llavors es passa a WindowPrincipal per a què es faci visible a l'usuari.

Pel que fa a la WindowPrincipal , no hi ha molt de secret: per a cada botó o acció se li ha fet correspondre l'action listener corresponent, que fa que aparegui una de les moltes vistes de la capa de presentació per introduir la informació pertinent, per després passar aquesta informació cap a WindowPrincipal per a què faci les crides corresponents a la capa de domini.

La gran major part del codi de les vistes l'ocupa el disseny i configuració de la interfície. Per tal d'aconseguir que el codi ens donés el resultat visual que volíem, vam usar un plug-in que té IntelliJ per a Swing que t'autogenera el codi a mesura que un dissenya la interfície amb

l'eina visual. Llavors, un cop dissenyada la interfície i tot el codi ja s'ha generat, hem netejat i estructurat una mica aquest codi per tal de fer-lo més llegible, "bonic" i menys artificial. Notis que les classes amb herència no s'han pogut dissenyar gràficament ja que el plug-in no és prou potent com per detectar aquesta herència i generar el disseny visual o gràfic de la classe "Mare", i per tant els components addicionals de les classes hereves s'ha hagut de fer totalment mitjançant codi.

RowNumberTable

RowNumberTable és una classe que hereta de JTable i ha estat utilitzada per a presentar el contingut dels Fulls a la vista principal de l'aplicació. La classe no ha estat implementada per aquest projecte sino que ha estat obtinguda de la següent pàgina web: https://tips4java.wordpress.com/2008/11/18/row-number-table/

Els principals avantatges d'usar aquesta extensió de JTable enlloc de l'original és la possibilitat de visualitzar correctament les etiquetes de les files, així com la de les columnes i facilitar que en introduïr dades a la taula des de la vista aquesta modificació es faci efectiva en les cel·les corresponents de la capa domini al Full que s'està modificant.

TableCellListener

TableCellListener és una classe que implementa un listener sobre la taula. És la responsable de que es puguin modificar correctament cel·les en el full a partir d'introduïr el seu nou contingut directament en la taula. Aquesta classe no ha estat implementada per aquest projecte sinò que ha estat obtinguda de la següent pàgina web: http://www.camick.com/java/source/TableCellListener.java

Aquesta classe escolta per a canvis fets a les dades de la taula mitjançant *TableCellEditor* que registra el valor de cel·la quan s'inicia la modificació. Quan la modificació es finalitza, es comparen el valor introduït amb l'emmagatzemat previament i, de ser diferents, s'invoquen les accions necessaries per fer efectiu aquest canvi i propagar-lo a la capa de domini.