**COMANDO**

Classe dove salviamo in modo astratto le informazioni da richiamare, rappresenta i comandi fondamentali di **EmotionalMaps** che sono rispettivamente **TipoComando tipo** e il relativo metodo get, **getTipo(),** e lo **String argomento** con il relativo metodo get, **getArgomento().**

**PUNTO\_DI\_INTERESSE**

Creazione della **classe puntoDiInteresse()** dove vengono istanziate le coordinate dei punti di interesse con le emozioni totali e le emozioni di utenti attivi.

Viene generato un nuovo punto di interesse *(con relativo nome e coordinata)*, e vengono instanziati a 0 i valori delle emozioni dei punti di interesse.

Creazione, dei relativi metodi **getEventiTotali()** e **getPuntoDiInteresse()**.

Creazione del metodo **resetStatistiche()** e crezione metodo **statisticheAttive()** dove crea una stringa che contiene una percentuale di tutte le emozioni attive.

Creazione del metodo **statisticheTotali()** fa la stessa cosa del precedente ma per tutti gli utenti, qualunque sia lo stato.

Creazione del metodo **aggiungiEvento()** che preleva l’evento e incrementa **eventiTotali**, controlla che il tipo di accesso sia uguale a **login** se è login allora aggiorna **eventiTotaliAttivi** quindi viene creato uno **switch** per gli stati emotivi che tramite il metodo ottiene lo stato emotivo dell’evento. In base allo stato emotivo viene incrementato il rispettivo contatore (totale) e viene eseguito un controllo per verificare lo stato di accesso, se è nuovamente **login,** viene incrementato il contatore per quello stato emotivo di utenti attivi.

Se lo stato emotivo non è tra quelli presenti viene stampato un messaggio WARNING specificando l’ID dell’evento.

Creazione del metodo **calcolaDistanza()** che controlla la distanza assoluta e dove viene richiamato il metodo della classe UTILITA che restituisce la distanza assoluta tra punto di interesse e le coordinate passate.

Creazione del metodo **percentualeTotale()** se

**if**(EventiTotali != 0 )

allora restituisce il valore in **int**, arrotondato.

**UTILITA**

Classe per la creazione di vari metodi di uso comune nel programma *(carica stringhe di file tramite buffer)*, esegue controlli fino a quando:

**if** (BufferedReader != null)

salva nell'ArrayList e manda un messaggio di ERROR.

Viene creato un metodo **boolean bisestile(),** che verifica se l’anno che viene passato come argomento è bisestile, se è tale torna TRUE.

Viene creato il metodo **suddividiStringhe()** dove viene passato come argomento l’ArrayList, in cui viene controllato se l’array è vuoto e crea una matrice salvando il risultato in posizione, prendendo la riga, fino a quando non riscontra la virgola .

Creazione del metodo **controllaArrayStringhe(),** dove viene passato l’array di tipo string, e controlla se è null o vuoto, se non lo è allora restituisce TRUE.

Creazione del metodo **aggiungiEstensione()** dove passa **<nomefile>** e controlla se finisce con la giusta estensione, se non è presente gliela aggiunge automaticamente in **“.txt”**.

Infine viene creato il metodo **distanza()** in cui passano le coordinate di due punti, e controlla se la variabile x del primo punto è maggiore del secondo, poi viene creata una variabile temporanea con i valore di x e vengono invertiti i valori, infine trova la distanza assoluta.

**MANAGER\_PDI**

Creazione della classe **ManagerPDI** e creazione della lista di tipo **<puntoDiInteresse>** con relativa creazione del metodo **ManagerPDI()** dove la lista viene instanziata come ArrayList di tipo **<puntoDiInteresse>** e viene aggiunta tramite metodi, il metodo, **aggiungiPDI()** dei nuovi punti di interesse i cui attributi sono il nome del punto, la sua latitudine e longitudine per tutti i punti. Creazione metodo **void elabora()** che ha come attributo l**’Evento e**, all’interno vengono istanziati:

* Indice
* IndicePDI
* distanza minima e distanza calcolata

Avviene una successiva scansione dell'ArrayList dove viene calcolata la distanza dal punto e ai vari punti di interesse e lo assegna al punto di interesse più vicino, poi lo aggiunge ad ArrayList.

Se invece l’indice è uguale a **-1** viene stampato un WARNING nell’elaborazione dell’evento.

Creazione del metodo **void resetStatistiche()** che resetta le statistiche di tutti i punti di interesse .

Infine la creazione del metodo **void stampaStatisticheTotali()** dove stampa la percentuale delle statistiche totali.

**EVENTI**

Creazione della classe per la rappresentazione degli eventi, serve a gestire in modo più semplice i dati riguardanti l’evento preso in considerazione.

Creazione della **classe evento()** con all’interno le relative variabili e creazione elemento di tipo **evento** inizializzato con tutti i campi dai valori dati in input.

Creazione metodo **getAccesso()** facendo riferimento alla classe **enum** statistiche

Creazione del metodo **getID()** che restituisce l'id dell'utente, e creazione del metodo **getX()** e **getY()** che restituiscono le relative coordinate.

Crezione di **getData()** che restituisce la data, e creazione di **getStatoEmotivo()**, facendo riferimento alla classe **enumerativa StatoEmotivo.**

**MANAGER\_EVENTI**

Viene creata una classe **ManagerEventi()** e viene creato un HASHMAP *(implementato nelle*

*api di java),* la cui mappa è una collezione di oggetti il cui scopo principale è rendere veloce ed efficiente le operazioni di inserimento e ricerca elementi. La mappa memorizza coppie *(chiave* e *valore),* come *chiave* ha delle stringhe mentre come *valore* ha un ArrayList chiamato <**listaEventi>.**

Creazione nuovo elemento di **ManagerEventi** dove al suo interno viene istanziato <**listaEventi>** . Crezione metodo **void aggiungiEventi()** di attributo **<nomefile>**, con l’ArrayList settato a null e anche la stringa.

**Creazione di array di stringhe** dove tramite il richiamo di un metodo della classe UTILITA viene caricata la stringa nel file, dove viene verificato se l’array di stringhe non e null e contestualmente viene controllato tramite il metodo **boolean** **controllaArrayStringhe()** che controlla effettivamente se l’array e il suo contenuto sono formalmente corretti. Successivamente tutti gli elementi dell’array vengono convertiti in tipo **evento**

**if**( **evento** != null)

viene controllata la data in **<listaCorrente>** se è = a null oppure se e != dalla data dall’**evento**, in quel caso data **<listaCorrtente**> prende il valore della data dell’**evento**. Imposta **<listaCorrente>** con **dataEvento**, controlla se = null, se è cosi viene instazia **<listaCorrente>** come ArrayList di tipo **evento** e viene associata alla chiave **dataCorrente***(string)* il valore di **<listaCorrente>** che se è = null viene aggiunto l’**evento** all’ArrayList di **<listaCorrente>**, nel caso invece è **evento** adessere = a null, viene comunque aggiunto.

Nel caso però in cui l’array stringhe è = null, oppure il metodo **controllaArrayStringhe()** restitusce FALSE, viene stampato a schermo un WARNING indicando che non è stato possibile aggiungere l'evento alla lista.

Nel caso invece venga riscontrata un eccezione di tipo **IOException** viene stampato il medesimo WARNING.

Creazione di un nuovo elemento di tipo **resetEventi()** che istanzia nuovamente la **<listaEventi>**

come HASHMAP e viene creato il metodo **converti()** che è un array di tipo **evento** che ha come **input** un array di stringhe chiamato **datiGrezzi[]**.

Viene creato un ArrayList di tipo **evento** chiamato **<listaConvertiti>** e vengono salvati in una matrice di stringhe chiamata **datiSuddivisi[][]**, i datiGrezzi vengono quindi separati tramite il metodo **suddiviStringhe()** e infine viene istanziato un indice uguale a 1.

**if**(datiSuddivisi == null)

return null;

Vengono poi salvati i **datiSuddivisi** in un array di tipo **dato[]** e viene creato un **boolean corretto impostato a** TRUE. Richiama registrazione **login** ed **emozione** dai rispettivi **enumerativi** e li imposta a null, ed effettua un controllo dove se la prima posizione dell’array è “IN” viene impostata come stato registrazione IN , altrimenti OUT.

Se non viene verificato nessuno dei due casi imposta **corretto** aFALSE.

Effettua poi un secondo controllo per la seconda posizione, se è uguale a **login** viene impostato **login** uguale a stato utente LOGIN, se **logout** imposta stato utente a LOGOUT.

Se invece non è nessuno dei due allora **corretto** viene impostato a FALSE.

Viene effettuato un controllo per la correttezza sintattica per la 3 posizione altrimenti MHANZ. Creazione di uno **switch** con i valori in 7 posizione per ogni **emozione** consentita nell**'enumerativo** viene impostata l’emozione appropriata.

Se il valore non dovesse esserci, allora **corretto** viene impostato a FALSE.

Se i **dati[i]** sono **corretti** viene aggiunto all’ArrayList l’**evento** altimenti se **corretto** = FALSE stampa un WARNING indicando l’indice.

In ultimo ritorna la lista come array.

Viene creato il metodo **ricerca()** con argomenti **data1** e **data 2** viene istanziato un ArrayList chiamato **<eventi>** eviene effettuato un controllo se le date sono nel formato corretto.

Controlla poi se **data1 < data2**, se lo è la ricerca viene saltata e ritorna null.

Recupero eventi compresi tra le date e vengono salvati sull’ArrayList di tipo <lista>

**if**(lista != null)

vengono aggiunti ad <**eventi>** tutti gli eventi contenuti nell HASHMAP che hanno quella data e restituisce gli eventi trovati.

Creazione del metodo **contieneEventoId()** con argomento **l’ID** e una lista, e verifica che nella lista sia presente **l’ID** dell’evento specificato con ID.

Viene creato il metodo **controllaData()**, viene passato come argomento la data.

Se la lunghezza dovesse essere maggiore di 8, o se il formato della data non dovesse essere di soli numeri, o ancora se l’anno di inizio fosse inferiore al 1970, o per ultimo, il formato della data non dovesse essere valido, allora restituisce FALSE, altrimenti TRUE.

Viene creato un metodo **boolean primaDiData()** che controlla che **data1** preceda **data2**.

Viene poi creato un metodo **boolean** chiamato **stessaData()** che controlla che **data1** e **data2** facciano riferimento alla stessa data.

Creazione di un metodo **boolean dopoDiData()** che controlla che controlla che **data1** succeda **data2.**

Creazione finale del metodo **creaDataSuccessiva()** con argomento uno **String data,** dove viene generata una data successiva a quella data in input.

Se quella data in input non è **corretta** restituisce null.

**MANAGER\_COMANDI**

Viene creata una classe **ManagerComandi()** per la gestione dei comandi di **EmotionalMaps** e vengono dichiarate le variabili **mpdi** di tipo **ManagerPDI**, **me** di tipo **ManagerEventi** e **listaComandi** dell'array **Comando[]**.

Viene Creato un nuovo **ManagerComandi** istanziando le variabili prima citate,

**me** = new **ManagerEventi();**

**mpdi** = new **ManagerPDI();**

**listaComandi** = null;

Creazione di un nuovo **ManagerComandi** inizializzando il proprio **ManagerEventi** con uno già

esistente, creazione successiva del metodo **void caricaComandi()**, che carica i comandi contenuti

nel file **<nomefile>**. La funzione prevede come input una stringa contente il path da cui caricare i comandi per l'esecuzione del programma. Nel caso in cui avvenga un errore irrecuperabile durante

la lettura del file viene stampato a schermo **l'ERROR** e successivamente viene chiuso in maniera forzata il programma.

Poi viene creato il metodo **esegui()**, che esegue i comandi contenuti in **<listaComandi>**.

Nel caso non riconosca il comando, stampa a schermo un **WARNING** con l'indice del comando non riconosciuto.

Viene creato poi il metodo **converti()** di tipo **comando[]** che restituisce un array di oggetti comando

convertiti dalle stringhe contenute in input, quindi nel caso la stringa non possa essere convertita verrà ignorata e la funzione continuerà a procedere.

Viene creato il metodo **ottieniArgomento()** che restituisce l'argomento passato al comando contenuto in stringa, la cui funzione prevede in input un comando conforme in quanto non esegue

alcun controllo.

Infine, creazione del metodo **boolean formatoComandoCorretto(),** che controlla che la stringa passata in input sia conforme alle specifiche riguardanti i comandi accettati dal programma. Questo

controllo viene effettuato tramite **Reg.exe***(che controlla la struttura lessicale della stringa)*.

L'ultimo metodo ad essere creato è il metodo **creaMappa(),** che genera una mappa emozionale degli eventi che sono stati caricati e che sono avvenuti nell'intervallo di tempo contenuti in argomento.

**EMOTIONAL\_MAPS**

Viene creata la classe **EmotionalMaps()** che gestisce il flusso di funzionamento del programma,

questa è la classe principale.

Viene istanziato **mc** di tipo **ManagerComandi.**

Nel **main** viene creato il metodo **controllaArgomenti()**, che ha la funzione di controllare che i parametri forniti dalla linea di comando siano **corretti**, i controlli eseguiti sono rispettivamente:

* numero degli argomenti
* estensione del file
* esistenza del file fornito

Se l'estensione del file dovesse mancare, gli viene aggiunta mediante il metodo **aggiungiEstensione()** della classe UTILITA.

Per ultimi vengono creati i metodi **importa()**, che carica i comandi contenuti in **<nomefile>** nel **ManagerComandi**.

*N.B. (Tutti i comandi non conformi vengono ignorati lasciando proseguire il funzionamento del programma).*

Infine viene creato il metodo **void esegui(),** che esegue serialmente i comandi caricati nel **ManagerComandi.**

**STATO\_EMOTIVO**

Stato gestito da sistema

(NEUTRO, FELICE, TRISTE, ARRABBIATO, SORPRESO)

**STATO\_REGISTRAZIONE**

Stato registrazione degli utenti gestiti da sistema

(IN/OUT)

**STATO\_UTENTE**

Stato di accesso utente gestito da sistema

(LOGIN/LOGOUT)

**TIPO\_COMANDO**

Tipologia di comando gestito da sistema

(IMPORTA/CREA\_MAPPA)