RELAZIONE PRIMO PROGETTO INTERMEDIO
Matteo Biviano – Corso B – Matricola 543933

1 L'interfaccia DataCounter<E>

L'interfaccia DataCounter<E> rappresenta una collezione modificabile di oggetti di tipo E, a cui è associato un valore numerico (rappresentativo del numero di volte in cui l'oggetto occorre nell'insieme).
È stato deciso di strutturare la possibile implementazione dell'interfaccia come funzione totale "$f: E \rightarrow N$" che associa al dominio di oggetti E l'insieme dei naturali N, che rappresenta il codominio. Nello specifico, questa funzione non è parziale, poiché non è definita solo sugli elementi presenti nella collezione, ma su tutti gli elementi presenti nel dominio, di conseguenza è totale: *f* assume valore *n*, pari al numero di occorrenze dell'elemento nella collezione, oppure 0 se questo elemento non è presente nella collezione.

1.1 Metodi di DataCounter<E>

L'interfaccia presenta i seguenti metodi:
- **incCount(E data)**: è il metodo che impone di considerare la classe *modificabile* poiché permette di modificarne lo stato interno. Prende come argomento un oggetto *data* di tipo *E* e, se *data* appartiene alla collezione, ne incrementa il valore associato, altrimenti aggiunge *data* alla collezione, con valore associato pari a 1. La precondizione di questo metodo è che *data* non sia nullo. Nel caso in cui *incCount* venga chiamato su un oggetto nullo, si è scelto di lanciare un'eccezione *unchecked* (disponibile in Java) di tipo *NullPointerException*, in linea con uno stile di programmazione difensiva;
- **getSize()**: restituisce il numero di elementi presenti nella collezione. Non richiede precondizioni e, in quanto *osservatore*, non modifica lo stato interno;
- **getCount(E data)**: è un osservatore che prende come argomento un oggetto *data* di tipo *E* e ne restituisce il numero di occorrenze. La precondizione richiesta è che *data* non sia nullo, mentre, in caso contrario il programma prevede il lancio di un'eccezione *unchecked* (disponibile in Java) di tipo *NullPointerException*;
- **getIterator()**: questo *osservatore* restituisce un iteratore per la collezione che permetta di scansionare coppie ordinate senza la possibilità di eliminarne alcun elemento.

2 Le classi HashDataCounter<E> e TreeDataCounter<E>

Le classi *HashDataCounter<E>* e *TreeDataCounter<E>* rappresentano due possibili implementazioni dell'interfaccia *DataCounter<E>* basate sull'uso di due strutture di supporto, rispettivamente *HashTable<E, Integer>* e *TreeMap<E, Integer>*. Le due classi risultano similari, l'unica differenza dal punto di vista funzionale, è rappresentata dalla scelta del tipo *E* della classe *HashDataCounter*, il quale deve necessariamente implementare i metodi *equals* e *hashCode*, al fine di memorizzare e recuperare con successo oggetti da un *hashtable*.
La caratteristica particolare delle due implementazioni risiede nel metodo *getIterator*. In questo metodo viene istanziato un *TreeSet* contenente coppie di tipo *PairDataCounter<E>*, che

verranno ordinate in modo decrescente, rispetto la frequenza delle chiavi. Nel caso in cui ci siano chiavi con stessa frequenza esse verranno ordinate in modo lessicografico.
Il *TreeSet* permette l'ordinamento dei suoi elementi attraverso il metodo *compareTo*, che è implementato nella classe *PairDataCounter<E>*. L'ordinamento è possibile poiché *PairDataCounter* implementa *Comparable<PairDataCounter<E>>*

2.1 Funzione d'astrazione – Invariante di rappresentazione

La funzione di astrazione è identica per entrambe le classi, ed è così definita:

$$\text{“}f: E \to N\text{”, tale che: } f(k) = \begin{cases} datacount.get(k) & se\ k \in datacount \\ 0 & altrimenti \end{cases}$$

L'invariante di rappresentazione è anch'essa uguale per entrambe le classi:
$datacount\ != null\ \&\&\ for\ all\ key \in datacount.keySet() => datacount.get(key) > 0$

2.2 Test delle Implementazioni

La classe *TesterDataCounter* presenta codice di controllo che ha il compito di verificare il corretto funzionamento delle classi implementate. *TesterDataCounter* si avvale della classe *UtilsDataCounter* per quanto concerne i due metodi atti a svolgere i test:

- **testerException(E data)** che controlla il corretto funzionamento dei metodi;
- **testerFile(E data)** che applica l'implementazione scelta al file *Test0.txt*

3 La classe WordFrequencyCount

La classe *WordFrequencyCount* applica l'implementazione fornita, determinando la frequenza delle parole in un documento di testo. Avvalendosi della classe *UtilsWordFrequencyCount*, permette all'utente di scegliere tra alcuni testcase quello da sottoporre al programma e quale implementazione della classe *DataCounter* testare.
È stato utilizzato un oggetto *BufferedReader* per leggere un testo riga per riga. Sono stati istanziati due oggetti *String* che in sede di lettura permetteranno il conteggio del numero di caratteri presenti nel testo, con e senza spazi. Il testo viene normalizzato attraverso la rimozione della punteggiatura – avvalendosi del pattern *\\p{Punct}* – e tramite la conversione dei caratteri maiuscoli in minuscoli – utilizzando il metodo *toLowerCase*. In seguito, ogni riga viene passata come parametro ad un oggetto *StringTokenizer,* che si occupa di suddividerla in singole parole, le quali verranno inserite all'interno di un oggetto *DataCounter* e la loro quantità verrà memorizzata nella variabile *n_words*.

4 Note finali

I testcase, a cui le classi *TesterDataCounter* e *WordFrequencyCount* fanno riferimento, sono stati creati in codifica *UTF-8 (senza BOM)* per permettere il corretto funzionamento del programma proposto nei sistemi UNIX. Eventuali altre codifiche potrebbero non garantire che i risultati ottenuti siano identici a quelli attesi (es. la stringa *più* potrebbe essere riportata in tabella come *più*[1], con conseguente alterazione dei conteggi effettuati dal programma).
Dopo la compilazione di *TesterDataCounter* o di *WordFrequencyCount*, l'esecuzione del codice risulta guidata, perciò non risultano necessarie ulteriori note.