Mattia Sorgato, matricola 1004404

Relazione di Progetto di Programmazione Concorrente e Distribuita

Anno didattico 2014/2015

Principi di Programmazione ad Oggetti

All'interno del progetto didattico da me svolto, ho prestato molta attenzione a far sì che ogni campo utilizzato all'interno della classe rimanesse non accessibile (soprattutto non modificabile) dall'esterno della classe stessa. Il principio dell'incapsulamento è visibile maggiormente nella classe "Piece", che rappresenta un pezzo di un Puzzle.
In questa classe, i campi riguardanti gli identificativi proprio e dei pezzi adiacenti, sono nascosti all'esterno e sono marchiati **final**, cosicché non possano venir modificati da azioni esterne alla classe. Sono comunque presenti dei metodi di confronto tra due **Piece** che permettono di effettuare controlli sugli **ID**, nascondendo però l'architettura sottostante.
Il Parser che effettua il controllo dell'input e la classe che rappresenta l'algoritmo di ordinamento sono entrambe composte da un'interfaccia che espone i metodi base di ognuna e una classe che implementa l'interfaccia. Ciò è motivato da cause di modifiche future o di riadattamenti successivi, in quanto il progetto sarà modificato nelle prossime consegne, e avendo un'interfaccia comune, bisognerà solamente ridefinire gli **override** dei metodi usati per chiamate polimorfe.
Questa scelta è stata presa anche in un'ottica di gestione della concorrenza. Infatti, con le prossime specifiche e l'introduzione della concorrenza, l'algoritmo di ordinamento è già stato pensato per ordinare il puzzle in maniera parallela, su più linee, così da rendere il programma più concorrente possibile.
La suddivisione in diversi *package,* invece, è dovuta ad una maggior divisione dei vari aspetti del progetto, così da aumentare l'indipendenza delle varie funzionalità.

Organizzazione

Package "puzzlesolver"

La dicitura *"puzzlesolver"* identifica il package generale del progetto.
In questo package è posta la classe **PuzzleSolver,** contenente il main da eseguire. Questa classe contiene i metodi di lettura e scrittura da file necessari per far dialogare il main con i file contenenti i puzzle e per sapere dove scrivere i risultati dell'esecuzione. Inoltre è presente il metodo *puzzleStringBuilder(Piece[][])* che prepara la stringa da stampare in output.

Package "puzzlesolver.parser"

Questo package contiene l'interfaccia **IParser** e la sua implementazione **PuzzleParser**. Il Parser espone le funzionalità di controllo dell'input utente, esaminando la sintassi di ogni stringa ricavata dal file di input. Il metodo *parsePuzzle(String),* che nella versione corrente richiama *parseLine(String)* sequenzialmente su ogni stringa, controlla che l'input non presenti errori sintattici, quali:

- un pezzo non può contenere più di un carattere, e non può essere il carattere di tabulazione o caporiga;
- la dimensione delle informazioni contenute nella stringa sono solo quelle date dalla specifica (ID, carattere del pezzo e ID dei pezzi adiacenti);
- nessuna parte della stringa in input può essere vuota.

Per effettuare l'ultimo controllo, nella classe **PuzzleParser** ho inserito un metodo statico *inputNotEmpty(String[])* che controlla che appunto non vi siano campi vuoti.
Inoltre, il Parser espone il metodo *idCheck(List<Piece<)* che viene utilizzato per controllare che gli ID dei vari pezzi siano legali, ovvero che non esistano pezzi con lo stesso ID, o con lo stesso ID nella stessa posizione cardinale.

Package "puzzlesolver.piece"

In questo package si trova la classe **Piece**, che rappresenta un pezzo del Puzzle. Questa classe è composta da un campo **final** (una volta creato un pezzo, non voglio che sia modificabile) per ogni informazione in esso contenuta, ovvero il proprio ID, il carattere contenuto nel pezzo e gli ID dei pezzi adiacenti.
Oltre ai due costruttori della classe, vi sono dei metodi per ottenere le informazioni contenute nel pezzo e dei metodi di confronto tra ID dei pezzi.

Package "puzzlesolver.sort"

Sono contenuti qui l'interfaccia **ISort** e la sua implementazione **SequentSort**, che implementa l'algoritmo di ordinamento.
I metodi esposti dall'interfaccia sono *sortLine(Piece, List<Piece>)* e *sortPuzzle(List<Piece>).* Ho scelto di dividere in due l'ordinamento per cercare di, nella prossima parte del progetto, rendere parallelizzabile l'ordinamento di più righe contemporaneamente. Ora, il metodo *sortPuzzle* richiama l'ordinamento riga per riga.
Nella classe **SequentSort** sono presenti due metodi ausiliari utilizzati nell'ordinamento, ovvero *removePiece(String, List<Piece>)* e il metodo statico *compareSize(int, Piece[][])*. Il primo metodo ricerca all'interno della lista di Piece un pezzo con ID voluto e lo ritorna, togliendolo dalla lista; il secondo, invece, serve per comparare la grandezza iniziale del puzzle con quella del puzzle ordinato.

Package "puzzlesolver.tests"

In questo package sono presenti le classi di collaudo del Parser e dell'algoritmo di ordinamento, con variabili ad hoc per testare il comportamento delle due classi.

Algoritmo di ordinamento

L'idea alla base dell'algoritmo sequenziale è molto semplice. Per la sua ideazione e codifica, mi sono ispirato al metodo comune di risoluzione di un puzzle, ovvero partendo dai bordi e successivamente riempiendo l'interno. La strategia utilizzata per l'ordinamento si divide in diversi passi:

1. Trovare il pezzo che identifica il bordo in alto a sinistra;
2. Grazie a questo pezzo, completare la prima linea del puzzle cercando ogni volta il pezzo che ha come proprio ID quello che sta ad EST di quello precedente, fino ad arrivare al bordo EST;
3. Si passa all'ordinamento della riga successiva cercando il primo elemento, il quale avrà l'ID corrispondente al SUD del primo elemento della riga precedente;
4. Ripetere il punto 3 fino ad arrivare all'ultima riga del Puzzle.

Se l'algoritmo di ordinamento procede senza intoppi, alla fine della sua esecuzione verrà ritornata una matrice contenente i pezzi ordinati del puzzle. Se questo non dovesse succedere, possono verificarsi diversi casi. Infatti, al termine dell'ordinamento di ogni riga, viene controllato che non manchino pezzi alla riga (sappiamo infatti da specifica che il file in input deve essere un puzzle rettangolare completo). Questo controllo viene effettuato per ogni riga e, al riscontro di un problema, l'algoritmo si interrompe e ritorna **null** al chiamante, che gestirà questa mancanza segnalando un errore.

Test effettuati

Il programma è stato testato in parti e nell'insieme. Nel package "puzzlesolver.tests" sono presenti vari test delle parti, dove si inseriscono input corretti ed errati per verificare la validità degli output.
Nella classe **WrongParserTest** ho provato a richiamare il Parser con input errati quali:

- stringhe con più di un carattere nel pezzo
- puzzle contenente una stringa con carattere di caporiga come carattere del pezzo
- stringa contenente il carattere di tabulazione come carattere del pezzo
- stringa con minor numero di argomenti del dovuto

Nella cartella "bin", nel package "puzzlesolver.tests", sono presenti invece diversi file .txt contenenti errori voluti nella costruzione del puzzle e la loro esecuzione mostra nello standard output gli errori riscontrati nel loro parsing da file.

Note

Utilizzo Java7

All'interno del progetto didattico ho utilizzato un costrutto proprio di Java7, il "**try-with-resources**". Questo costrutto è particolarmente utile per gestire le operazioni di input/output da file, in quanto consente di creare e gestire una variabile da utilizzare nel corpo del **try** direttamente come argomento del **try** stesso, rendendo così la sintassi più ridotta e leggibile. Inoltre, questo costrutto permette di chiudere ogni connessione con i file automaticamente, senza bisogno di richiamare metodi aggiuntivi.
Oltre a questo, ho utilizzato la classe **StringBuilder**, che mi è stata molto utile per "impacchettare" l'output prima di scriverlo su file. Infatti, questa classe è pensata per costruire una stringa aggiungendo pezzi in coda, per poi esportare la stringa completa.

Bash

Nel root del mio progetto ho creato un file *bash* chiamato **Test.sh.** Questo bash esegue prima di tutto una pulizia dei compilati e una successiva ricompilazione, chiamando il **make** incluso nella *root* del progetto. Successivamente, invoca il comando di esecuzione dell'eseguibile del *main* di **PuzzleSolver** con degli argomenti di prova, prendendo in input il file "bin/puzzlesolver/tests/base.txt" e crea l'output nella stessa cartella del file di input, chiamando l'output "risultato".

Nota: Se viene eseguito il comando *bash,* o anche tramite il **make***,* i compilati vengono posizionati ordinatamente nella cartella **bin.**

Se non si volesse utilizzare il *bash* per eseguire il progetto, una volta compilati i file necessari (utilizzando il compilatore di Java7), occorre posizionarsi nella cartella dei compilati (che nel caso del *bash* è **bin**) e chiamare il comando

java -cp . puzzlesolver.PuzzleSolver path_input path_output