Sebastiano Valle

Risolutore di puzzle – Parte 1
Programmazione concorrente e distribuita
Progetto A.A. 2014/2015

1 Programmazione a oggetti

1.1 Incapsulamento

Tutti i campi dati delle classi nel progetto sono privati, ovvero accessibili solamente da metodi della classe stessa, così da isolare completamente il lato utente dalla struttura interna degli oggetti. Non sono state dichiarate costanti pubbliche poiché non è stato ritenuto necessario.

Al contrario, non tutti i metodi sono stati dichiarati pubblici e (di conseguenza) disponibili all'utente. Questo è dovuto al fatto che alcune porzioni di codice sono state isolate per modularizzare al meglio il comportamento dei metodi, riuscendo in questo modo a leggere ed analizzare in modo più chiaro alcune parti significative (e.g. orderLine(top : Dir, i : int) nella classe *Puzzle*).

1.2 Ereditarietà

In questo progetto è stato deciso di non creare classi derivate da altre classi, ma solamente da interfacce. Questo per due semplici motivi: non era necessaria alcuna ereditarietà di implementazione, ma erano necessarie delle ereditarietà di tipo, in grado di soddisfare alcuni contratti (vari a seconda dell'entità, ad esempio ordinamento e scrittura su un tipo *GruppoOrdinabile*).

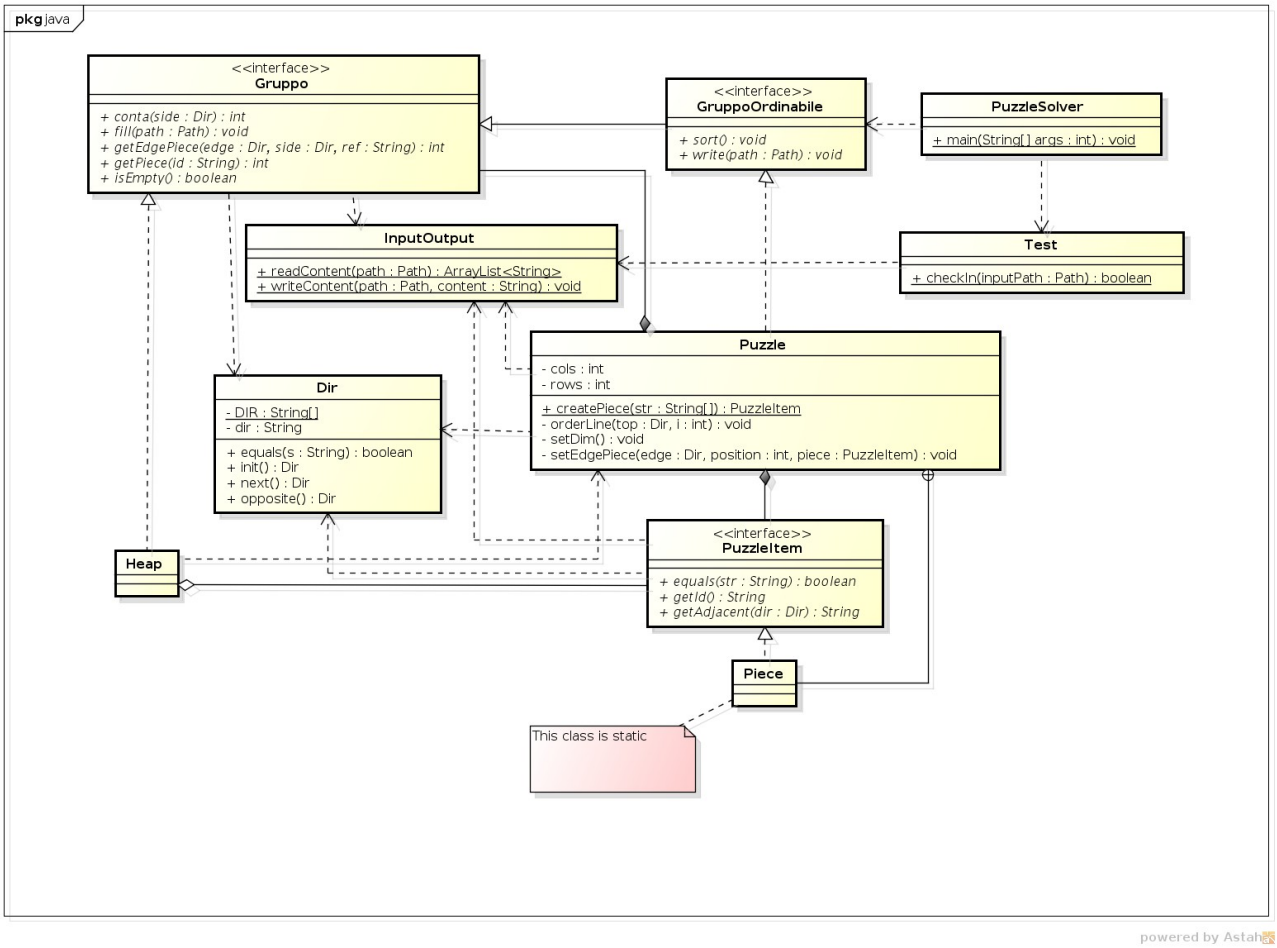
Un altro punto che è stato di orientamento verso la scelta dell'utilizzo delle interfacce per rappresentare tipi di oggetti è stato il fatto che queste garantiscono il maggior grado di disaccoppiamento tra le varie classi all'interno di un progetto. Di conseguenza, se nelle parti successive sarà necessario definire delle nuove classi che rappresentino un puzzle, un mucchio, un pezzo o altre entità, sarà sufficiente scrivere nuove classi che implementino i contratti offerti dalle varie interfacce *Gruppo*, *PuzzleItem*, ecc.

1.3 Polimorfismo

Il polimorfismo permette di identificare comportamenti comuni in diverse classi.

Nel progetto, ciò è stato possibile per le classi *Puzzle* e *Heap*: *Puzzle* implementa GruppoOrdinato, la quale è un'interfaccia che estende *Gruppo*, in quanto anch'essa offre il riempimento, il conteggio dei pezzi al suo interno, il controllo *isEmpty*() (**false** se vi sono alcuni elementi dentro il gruppo) e di ottenere alcuni pezzi in base all'id del pezzo ricercato o a quello di un pezzo vicino.

2 Organizzazione delle classi



2.1 PuzzleSolver

Questa classe racchiude la procedura principale del programma, il main(): dopo aver validato l'input procede al riempimento, l'ordinamento e la scrittura del puzzle sul file di output. A tal fine, utilizza un *GruppoOrdinabile* così che oltre a riempire il gruppo, possa anche ordinarlo e stamparlo su un file.

2.2 Test

Questa classe realizza vari controlli per il main; verrà descritta in dettaglio più avanti nella sezione 4. Con il metodo *checkIn*(inputPath : Path) si può verificare se l'input rispetta le pre-condizioni.

(Nota: nel diagramma UML non è raffigurata la classe interna Pair, siccome questa era utile solo ai fini di test più accurati ma non per la validazione dell'input, così come i metodi diversi da *checkIn*(inputPath : Path)).

2.3 Gruppo

Questa interfaccia rappresenta un gruppo di elementi disposti in ordine casuale.

A tal fine, l'utente può

- *conta*(side : Dir) : contare tutti gli elementi che non hanno pezzi adiacenti sul lato *side*
- *fill*(path : Path) : riempire un gruppo prelevando le informazioni da un file di input *path*

- *getEdgePiece*(edge : Dir, side : Dir, ref : String) : ottenere un pezzo che non ha pezzi adiacenti sul lato *edge* e che sul lato *side* ha un pezzo con id *ref*
- *getPiece*(id : String) : ottenere un pezzo presente nel gruppo che ha il proprio id uguale a *id*
- *isEmpty*() : sapere se il gruppo è vuoto (**true**) oppure no (**false**)

2.4 GruppoOrdinabile

Questa interfaccia estende *Gruppo* poiché anch'essa rappresenta un gruppo, ma con ulteriori funzionalità offerte che un semplice *Gruppo* non poteva offrire:

- sort() : ordinare il gruppo
- write(path : Path) : stampare il contenuto del gruppo in un file path

Questa interfaccia rappresenta il tipo del puzzle che viene utilizzato nella procedura principale del risolutore di puzzle; è stato deciso di utilizzare un'interfaccia e non una classe astratta nel caso in cui il comportamento (e dunque l'implementazione) specifico dovessero cambiare (così come per gruppo).

2.5 InputOuput

Questa classe offre dei metodi statici (poiché non dipendono dalle varie istanze di questa classe) per la lettura e la scrittura su file.

2.6 Dir

Questa classe serve a rappresentare i punti cardinali; è stato scelto di rappresentarli perché spesso nell'ordinamento serviva conoscere il riferimento a nord/sud/ovest/est e il punto cardinale successivo (*next*()), opposto (*opposite*()) o uno di riferimento iniziale per partire con l'ordinamento (*init*()).

2.7 PuzzleItem

Questa interfaccia rappresenta i pezzi che compongono il puzzle; ciò è stato fatto così da permettere la facile modifica dell'implementazione *Piece* dei pezzi se dovesse servire, pur che continui ad essere offerta l'implementazione dei contratti per avere l'id di un pezzo (*getId*()), conoscere l'id del pezzo adiacente sul lato dir (*getAdjacent*(Dir dir)) e verificare se due riferimenti a pezzi del puzzle corrispondono in realtà ad un riferimento allo stesso pezzo.

Grazie a questa interfaccia si istanziano i tasselli con cui riempire *Heap* e la matrice ordinata dentro *Puzzle*, classe che offre un factory method pubblico per la creazione di PuzzleItem di classe Piece.

2.8 Piece

Questa classe fornisce un'implementazione di PuzzleItem ed è definita internamente a Puzzle così che eventuali pezzi possano essere istanziati solamente passando per questa. Questa classe presenta una ridefinizione del metodo *toString*(), con la quale si restituisce solamente il carattere del tassello.

Piece è definita come statica poiché è una classe interna che non ha alcun riferimento a nessun campo dati del proprio Outer.this *Puzzle*.

2.9 Heap

Questa classe, implementazione di *Gruppo*, rappresenta un gruppo possibilmente non ordinato di *PuzzleItem* del puzzle da risolvere; implementa tutti i contratti offerti da *Gruppo* e per come è stata pensata non si prevede che questi pezzi possano essere ordinati, in modo che questa rappresenti un insieme di *PuzzleItem* ad accesso casuale. Nel diagramma UML è indicata una relazione di aggregazione verso *PuzzleItem*, siccome questa contiene degli elementi di quella classe ma non può istanziarli direttamente: ciò è possibile solamente utilizzando il factory method di *Puzzle*.

2.10 Puzzle

Questa classe rappresenta l'implementazione di *GruppoOrdinabile*, ovvero un gruppo su cui è possibile richiedere le operazioni di ordinamento e di stampa oltre a tutte quelle offerte dal tipo *Gruppo*.

Al suo interno vi è un riferimento a un *Heap* dove tenere i *PuzzleItem* che non sono ancora stati ordinati dentro una matrice di PuzzleItem, il cui numero *rows* di righe e *cols* di colonne sarà determinato grazie al metodo *setDim*(), che conta gli elementi sul bordo sinistro e sul bordo superiore nel mucchio.

Puzzle ha anche due metodi private *setEdgePiece*(edge : Dir, position : int, piece : PuzzleItem) e *orderLine*(top : Dir, i : int) che sono stati pensati per l'algoritmo di ordinamento; per questo motivo saranno descritti più approfonditamente nella sezione 3.

Puzzle, come detto in precedenza, offre un factory method per la costruzione di *PuzzleItem* definiti al suo interno; questa scelta è utile nel caso in cui si dovesse variare l'implementazione dei tasselli del puzzle, in quanto basterebbe variare l'unica riga all'interno del factory method.

3 Algoritmo di ricostruzione del puzzle

Per ricostruire il puzzle, è stato pensato un algoritmo che potesse essere sufficientemente parallelizzabile in un secondo momento.

Data un mucchio contenente tutti i tasselli di un puzzle e una tabella con *rows* righe e *cols* colonne, l'algoritmo procede nel seguente modo:

1. Individuazione del lato *edge*, ovvero il lato più lungo tra quello superiore ed il sinistro
2. Riempimento del lato *edge* in modo sequenziale, partendo dall'angolo superiore sinistro
3. Riempimento (possibilmente parallelizzabile) di ogni linea (colonna o riga), partendo dai tasselli definiti sul lato *edge* e procedendo, per ogni riga, dall'alto verso il basso o da sinistra a destra a seconda che *edge* corrisponda rispettivamente il lato superiore o quello sinistro

Il passo 2 dell'algoritmo è svolto tramite il metodo *setEdgePiece*(edge : Dir, position : int, piece : PuzzleItem) di *Puzzle* richiamata per un numero di volta pari alla lunghezza del lato *edge*, mentre nel passo 3 dell'algoritmo è richiamato per ogni linea da riempire il metodo *orderLine*(top : Dir, i : int) di Puzzle.

4 Test

Sono stati eseguiti due diversi tipi di test per il programma, uno per la validazione dell'input e uno per testare la sua correttezza.

4.1 Validazione input

Questo test, realizzato invocando il metodo statico *checkIn*(inputPath : Path), verifica che l'input sia ben-formato, ovvero composto da righe aventi sei stringhe separate da un carattere di tabulazione.

4.2 Correttezza

Questo test per essere eseguito deve essere decommentato nella procedura *main*() di *PuzzleSolver*.

Dato un file di input testuale scritto su una sola riga senza tabulazione (questo perché nessun tassello può contenere tabulazione e ritorno a capo come requisito), il metodo *build*(inputPath : Path) di Test genera un file di input del tipo descritto nel punto 3.6 della specifica del progetto.

Questo file viene successivamente passato ad un metodo *shuffle*(path : Path) che scambia casualmente le righe che rappresentano i tasselli, così da non compiere assunzioni aggiuntive sulla struttura del file contenente i tasselli.

Con l'input appena mescolato, verrà riempito un GruppoOrdinabile per poi essere ordinato e stampato.

Se il file stampato non è corretto, una funzione checkOut(outputPath : Path) di Test segnalerà che vi sono errori e verrà visualizzata la scritta “ERRORE” nella console; altrimenti, se il file stampato è corretto, verrà visualizzata sulla console la scritta “OK”.

In tutti i casi, il programma ha sempre funzionato correttamente.

5 Note

5.1 Avvio del programma

Il programma è avviabile semplicemente con il comando

```
./PuzzleSolver nome_file_input nome_file_output
```

Con questo comando è prevista la rimozione dei file *.class grazie al compito clean nel makefile.

5.2 Java 7

Come richiesto da specifica del progetto, è stata usata la versione 7 di Java.

Questo ha permesso di poter utilizzare nella classe *InputOutput* il costrutto *try-with-resources*.