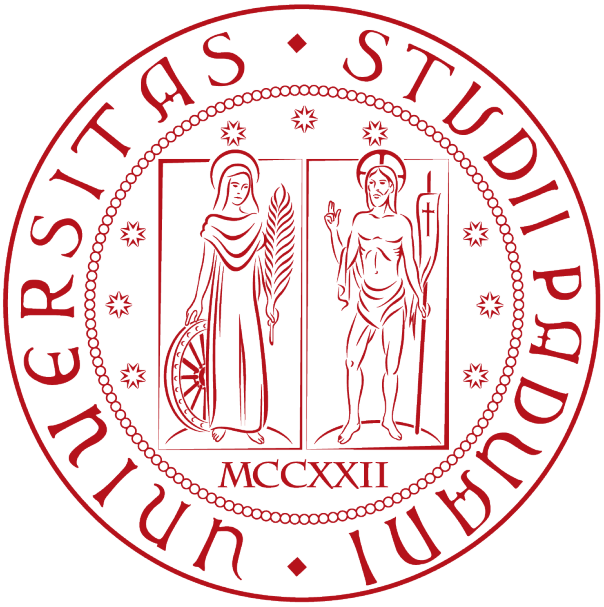

Risolutore di Puzzle

Parte 1

Stefano Munari 1031243

Sommario:
Relazione del progetto di Programmazione concorrente e distribuita AA 2014-15

Indice

1 Legenda e note importanti **1**

2 Scelte Progettuali di OOP **1**
2.1 `Text` 1
2.2 `InputText` 2
2.3 `OutputText` 2
2.4 `Puzzle` 2
2.4.1 `Piece` 2
2.5 `PuzzleSort` 3
2.6 `Eccezzioni` 3
2.7 `PuzzleSolver` 3

3 Gerarchia di classi **3**

4 Logica dell'algoritmo **4**

5 Test eseguiti **6**

1 Legenda e note importanti

I riferimenti al codice sorgente sono indicati nel modo seguente:
-`Classe`
-*`Metodo`*
-Interfaccia

Per brevitá nella seguente relazione il riferimento a metodi verrá fatto indicandone solamente il nome e il tipo di parametri formali. Se fosse necessario disambiguare la classe a cui appartiene il metodo(ad esempio nel caso di overriding) sará preposto il nome della classe al nome del metodo.
Si é scelto di utilizzare alcuni costrutti presenti solamente dalla versione 7 di Java, pertanto per compilare il sorgente é necessario disporre di una versione >= Java7.
In particolare é stato usato il costrutto try-with-resources perché piú conciso e semplice da utilizzare con gli Stream.
Comandi per compilazione ed esecuzione del progetto:

1. Per compilare il progetto lanciare il comando make da terminale.
2. Per eseguire il progetto scrivere la seguente riga da terminale dopo aver eseguito il punto precedente:
   sh puzzleSolver.bash nome_file_input nome_file_output

É necessario passare due parametri, come descritto al punto 2, per la corretta esecuzione del programma.

2 Scelte Progettuali di OOP

I metodi con visibilitá public sono tutti e soli i metodi che appartengono al contratto di chi utilizza la classe.
Il concetto appena espresso per i metodi vale anche per i campi dati delle classi.
Ho cercato di utilizzare degli identificatori significativi per entrambi in modo da rendere il codice di piú facile comprensione. Al fine di ottenere una relazione sintetica ho scelto di non descrivere nel dettaglio tutti i metodi ma solamente quelli per cui ho ritenuto necessaria una spiegazione.
Di seguito una sintetica spiegazione delle classi progettate:

2.1 `Text`

É una classe astratta, anche se non contiene alcun metodo astratto, perché fornisce un'intelaiatura di implementazione utilizzabile dalle classi derivate. Essa contiene un campo dati comune a quest'ultime. `Text` ha accessibilitá public perché cosí facendo é possibile fare polimorfismo sugli elementi della gerarchia definita nel package file, anche se il polimorfismo non viene utilizzato nel codice consegnato perché non necessario. Questa scelta progettuale rende la classe maggiormente predisposta ad utilizzi futuri nel caso si volessero inserire dei metodi su cui fare overriding. Infine il campo dati charsert é stato dichiarato final perché rappresenta un'informazione

immutabile durante tutta la durata del programma.

2.2 `InputText`

Legge un determinato file processandone il testo e memorizzandone le righe in un campo dati di tipo LinkedList attraverso il metodo *`readContent(Path,String)`*. Si é scelto di usare il costrutto try-with-resources per gestire l'eccezione sul possibile errore di apertura del file, lo stream di input viene chiuso automaticamente alla fine del blocco try. Il metodo *`match(String,int)`* controlla che la stringa in posizione position di tutti gli oggetti di lines corrisponda all'espressione regolare passata come primo parametro e ritorna true se ció si verifica, false altrimenti.

2.3 `OutputText`

Memorizza come stato le opzioni di apertura di file, ovvero in che modalitá deve essere aperto il file. Ovviamente *`writeContent(Path, String)`* ci si aspetta che scriva nel file presente in uno specifico path il contenuto della stringa passata come parametro. Nel caso del programma viene passato al costruttore dell'oggetto un OpenOption[] tale che si possa creare il file nel caso non esistesse (altrimenti viene aperto) e aggiungere la string passata in coda al file (APPEND) ad ogni chiamata di *`writeContent(Path, String)`*. Anche in questo caso ho utilizzato il costrutto try-with-resources nello stesso modo descritto in `InputText`.

2.4 `Puzzle`

Incapsula per composizione `Piece`. Ogni `Puzzle` contiene (relazione HAS-A) un insieme di riferimenti ad oggetti `Piece`. Altri 2 campi dati rappresentano le dimensioni del puzzle, il comportamento dei metodi restanti di questa classe é facilmente deducibile dal loro nome.

2.4.1 `Piece`

É una classe interna a `Puzzle` perché logicamente correlata ad essa, inoltre `Puzzle` utilizza campi dati privati di `Piece` (vedi `Puzzle`::*`setDimensions()`*). `Piece` é static perché non utilizza alcun riferimento a `Puzzle`. `Piece` é pubblica perché puó essere utilizzata anche in contesti che non richiedano necessariamente `Puzzle`. Il fatto di avere accessibilitá public per `Piece` non inficia encapsulation ed information hiding perché il contratto di `Piece` mantiene comunque privati e non direttamente modificabili dall'esterno i suoi campi dati. NUMBER_OF_FIELDS ed EMPTY_ID sono delle costanti di classe per Piece, hanno entrambi accessibilitá public perché contengono informazioni che si vogliono utilizzare all'esterno della classe e non influiscono sul contratto privato di quest'ultima. In questo modo se volessi cambiare il corrispettivo valore nullo per i punti cardinali in qualsiasi parte del codice mi basterebbe modificare il valore di EMPTY_ID, lo stesso ragionamento vale per NUMBER_OF_FIELDS.
Utilizzare una classe per i punti cardinali mi sembrava una scelta eccessiva per il tipo di Pezzo da modellare e modellarli come dei semplici campi di tipo stringa una scelta

poco strutturata, per cui ho scelto di raggruparli in un HashMap.
Ho scelto HashMap come struttura dati per i seguenti motivi:

1. é piú descrittiva e semplice da leggere rispetto ad un array o ad una lista.
2. sono dei campi dati concettualmente diversi da Id e value, nel caso volessi modellare un puzzle 3d e avessi quindi altre 2 dimensioni con questa scelta basterebbe aggiungerle al costruttore cambiando anche NUMBER_OF_FIELDS in modo che non venga sollevata un eccezzione di tipo `InvalidParameterSizeException`.

2.5 `PuzzleSort`

É l'algoritmo di ordinamento del Puzzle. Ho scelto di utilizzare una classe a sé per l'algoritmo in modo da modularizzare il codice. `PuzzleSort` oltre a metodi public possiede diversi metodi private. Quest'ultimi sono dei metodi di utilitá per la classe, in particolare sono utilizzati dal metodo *`sort()`*. Ho scelto di fattorizzare *`sort()`* e alcuni passaggi in esso contenuti per rendere il codice modulare, comprensibile e conseguentemente mantenibile. Si rimanda la descrizione piú analitica della classe `PuzzleSort` al punto 4 del presente documento.

2.6 `Eccezzioni`

Le classi di eccezzioni hanno tutte accessibilitá public in quanto la loro gestione é demandata a coloro che utilizzano dei metodi che contengono nella loro firma (method signature) delle dichiarazioni di tali eccezzioni. Un caso particolare é `InvalidIndexPosition` che risulta essere un'eccezzione interna di `InputText` gestita appunto internamente, per questo motivo ha accessibilitá package.

2.7 `PuzzleSolver`

Contiene il metodo statico main, entry point del programma. Al suo interno vengono istanziati ed usati i vari oggetti necessari all'esecuzione del programma. Sono gestite le eccezzioni che rappresentano un formato di input o un numero di parametri errato (vedi sezione test 1.1). Si é scelto di marcare tutte le variabili locali come final in quanto best-practice di programmazione Java.

3 Gerarchia di classi

Come si puó notare non sono state utilizzate interfacce.
Non sono state ritenute necessarie per questo di programmazione orientata agli oggetti (non ancora parallela e concorrente). Tuttavia ci si rende conto che nelle fasi successive del progetto saranno indispensabili per una corretta implementazione del paradigma concorrente.
Si é scelto di inserire le classi in package diversi per avere un sistema maggiormente modularizzato.
Sono state progettate 2 gerarchie di classi:

1. `Text` classe base astratta da cui ereditano `OutputText` ed `InputText`.

2. `InputException` da cui eredita `InvalidIndexPosition`.

In entrambi i casi la superclasse rappresenta un insieme piú generale di stati (campi dati) e comportamenti (metodi) che la sottoclasse estende. Ogni sottoclasse ha una relazione del tipo IS-A nei confronti della superclasse, ciò significa che per ogni oggetto della sottoclasse vale il principio di sostituzione di Liskov.

4 Logica dell'algoritmo

> "Pluralitas non est ponenda sine necessitate"
> — Guglielmo di Ockham

Una qualsiasi matrice é un'astrazione per un insieme contiguo di elementi: per questo motivo ho scelto di utilizzare come struttura dati che memorizza lo stato del puzzle una semplice lista.
Ho usato ArrayList per `Puzzle` e `PuzzleSort` in quanto necessitavo di una struttura dati espandibile dinamicamente e che mi permettese di avere accesso ad un elemento qualsiasi in tempo costante. L'ho preferita ad un array perché altrimenti avrei dovuto gestire i riferimenti ad esso nelle varie parti del codice condividendo tra loro gli stati dei vari oggetti. Quest'ultima sarebbe stata una scelta meno alla Java, dove gli oggetti sono passati molto spesso per valore a differenza del C++, e soprattutto meno pulita in vista di una futura versione concorrente dell'algoritmo. Infatti una delle regole fondamentali per fare concorrenza é avere il minor numero possibile di stati condivisi ed evitare così proattivamente il rischio di lasciare il programma in uno stato inconsistente. Inoltre un semplice array é una struttura meno flessibile rispetto ad ArrayList, implicherebbe righe di codice in piú per essere gestito e quindi diminuirebbe la leggibilitá del codice o perlomeno la renderebbe piú faticosa/difficile.
Infine ho preferito non utilizzare strutture derivate da Set per mantenere il codice semplice senza aver bisogno di assegnare delle chiavi aggiuntive ad ogni elemento per creare un ordine totale sull'insieme dei pezzi.
Tutta la logica dell'algoritmo é stata ideata basandosi sul principio KISS e pensando alla sua futura parallelizzazione.
Ho scelto di progettare un algoritmo semplice per 2 motivi:

1. l'algoritmo dovrá molto probabilemente subire delle modifiche in futuro per essere parallelizzato, in questo modo é piú facile fare manutenzione del codice.

2. non é richiesto nella specifica di progetto che il codice sia particolarmente efficiente quanto piuttoso che sia semplice ed efficace.

`PuzzleSort` prende come input un puzzle attraverso il metodo *`PuzzleSort(Puzzle)`* e lo ordina con il metodo *`sort()`*:

`PuzzleSort(Puzzle)` Crea lo stato di un oggetto di tipo `PuzzleSort`. result contiene l'insieme dei pezzi del puzzle. result puó essere un insieme non ordinato se

sort() non é ancora stata invocata oppure un insieme ordinato dopo l'esecuzione di *sort()*. columns é una struttura di supporto che contiene la lista delle colonne del puzzle durante l'esecuzione di sort, al di fuori di sort invece contiene una lista vuota. mentre puzzle_size contiene la dimensione del puzzle, ho preferito inserire anche questo campo dati perché durante l'esecuzione dell'algoritmo non avrei altrimenti alcun riferimento alla dimensione reale del puzzle risultante fino alla fine di *sortColumns()*. Questa informazione potrebbe essermi utile nella fase di parallelizzazione.

`sort()` Se l'insieme da ordinare ha un numero di elementi <=1 allora é giá ordinato, altrimenti viene eseguito il metodo *sort()* che si divide in 3 parti corrispondenti alla chiamata di 3 metodi con tipo di ritorno void:

1. corrisponde a *initializeColumns(ArrayList<Piece>)*
   costruisce attraverso una serie di chiamate di metodi annidate la prima riga ordinata del puzzle eliminandola dall'insieme result e salvandola in columns.

2. corrisponde a *sortColumns()*
   partendo dalla prima riga ordinata costruisce le colonne ordinate, alla fine dell'esecuzione di questo metodo columns é un insieme ordinato che corrisponde al puzzle risolto mentre result é vuoto in quanto non esiste piú nessun elemento da ordinare.

3. corrisponde a *mergeColumns()*
   columns viene copiato per righe in result che contiene il puzzle risolto ordinato per righe. Successivamente viene eseguito clear su columns che diventa quindi una lista vuota.

Idea per l'algoritmo nella sua versione concorrente:
l'algoritmo lavorerá parallelamente sulla creazione di colonne ordinate a partire da un insieme non ordinato di pezzi, questo insieme sará condiviso tra i vari thread che gestiscono le diverse colonne. Ogni thread lavorerá di copia sugli elementi dell'insieme senza modificarne nessuno in modo da evitare interferenze con gli altri thread.

Maggiori dettagli sul comportamento dei singoli metodi si possono trovare direttamente nel codice in forma di commento.

5 Test eseguiti

Sono stati eseguiti test di verifica sui seguenti tipi di input:

1. **Input non valido:**
   Ognuno dei seguenti casi viene gestito da una opportuna eccezione.

   1.1. **formato errato:**
   Comprende input che non corrispondono ai requisiti descritti al punto 6 della specifica di progetto. Fanno parte di questa categoria di eccezioni sia `InputException` (caso di valore del carattere non valido) sia `InvalidParameterSizeException` (numero di parametri non valido). Quest'ultima si riferisce a `Piece` anziché ad `InputText` in quanto é un informazione legata alla classe che voglio controllare ogni volta prima di costruire un oggetto di quel tipo. Cosí facendo aumento la robustezza e l'affidabilitá di `Piece`. `InputText` non é strettamente legata a `Piece` perché si occupa di processare un file di testo generico formattandolo secondo specifiche regole ma il suo stato non descrive necessariamente campi dati di `Piece`.

2. **Input valido:**
   Il programma si comporta come descritto nella specifica di progetto eseguendo e producendo un output nel formato richiesto.

   2.1. **input vuoto:**
   Un puzzle vuoto é un caso di puzzle degenere con dimensione 0x0. Ho considerato l'input vuoto come un input valido per il programma. Infatti il puzzle vuoto é giá risolto, tutti i suoi pezzi sono banalmente ordinati perché ha 0 pezzi da ordinare. Il programma produce sempre un output che rispetta il formato richiesto, in questo caso la prima riga e la terza riga sono vuote in quanto il testo é vuoto, mentre la quinta riga del file di output contiene le dimensioni 0x0.

   2.2. **input non vuoto:**
   Un puzzle non vuoto viene ordinato e stampato come descritto nella specifica di progetto.

3. **Input in un formato valido ma puzzle non risolvibile:**
   Si è scelto di non gestire questo particolare tipo di input escludendolo nella PRE-CONDIZIONE del costruttore di `PuzzleSort`. Il programma, in particolare l'algoritmo di risoluzione, assume di ricevere in input `Puzzle` che siano sempre risolvibili.