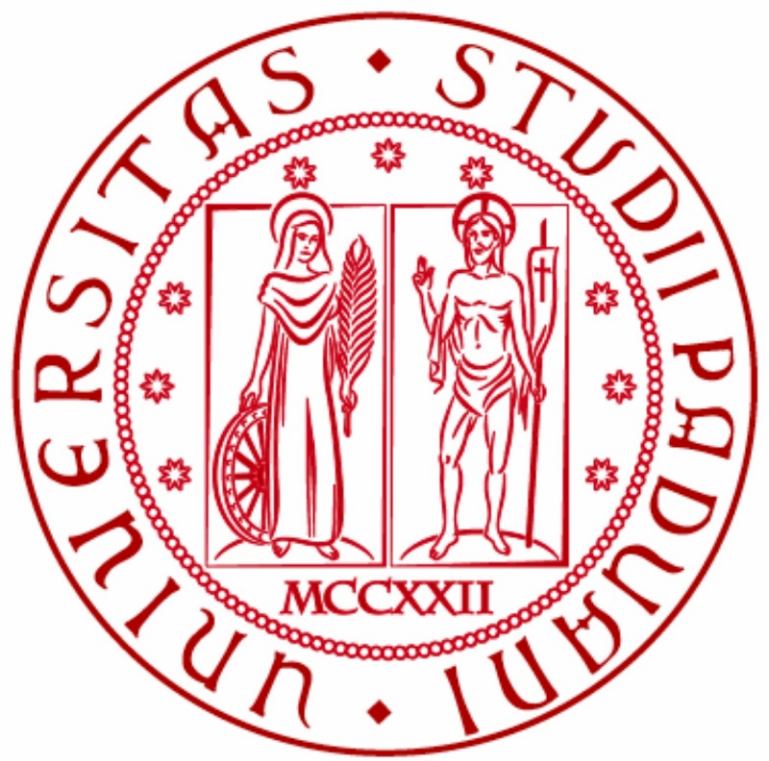

Relazione

Programmazione Concorrente e Distribuita

Prima parte

Simone Campagna 1005922

1 Introduzione

1.1 Abstract

Il progetto si prefigge di risolvere un puzzle commissionato tramite file di input. Tale dovrà prendere in input la descrizione dei tasselli disordinati componenti il puzzle, ordinarli e restituirne il puzzle per il suo intero.

1.2 Scopo del documento

Il documento ha lo scopo di descrivere l'architettura dell'applicazione, evidenziandone le scelte architetturali, descriverne l'algoritmo di ordinamento adottato per il progetto e l'eventuale installazione di tale. Inoltre in tale verranno trattati le modifiche apportate alla precedente architettura relativa alla prima parte del progetto didattico.

2 Architettura

L'architettura si compone di due package principali, rispettivamente *puzzleIO* e *puzzleObject*

2.1 Package

2.1.1 puzzleObject

PuzzleObject è il package ospitante l'architettura descrivente l'oggetto puzzle. In tale è possibile trovare una gerarchia che vede alla base una classe astratta *Puzzle* conferente le principali operazioni per l'oggetto, oltre che ai tasselli componenti tale. Derivate da essa troviamo poi *PuzzuleUnsolved* e *PuzzleSolved* classi relative ai puzzle nello stato disordinato ed ordinato.

2.1.2 puzzleIO

PuzzleIO è il package relativo alle operazioni di input ed output effettuate sui puzzle, potendone inizializzare il contenuto da un file di input o semplicemente scriverlo su un file di output descritto.

2.2 Classi

Come gà detto l'entità Puzzle è stata modellata con una gerarchia che vede alla base una classe astratta *Puzzle* e relative classi derivate costituenti lo stato del puzzle stesso. E' stata modellata come descritto dato che così

facendo si è potuto sfruttarne il polimorfismo nell'implementazione della funzione toString(), potendo rispettare le specifiche di consegna del progetto.

2.2.1 Puzzle

La classe astratta fornisce l'intelaiatura per gli oggetti di tipo puzzle mettendo a disposizione di seguenti metodi:

- **int rows()**: restitusce il numero attuale di righe
- **int columns()**: restituisce il numero di colonne attuali
- **void setColumns(int c)**: setta il numero di colonne pari a c;
- **void setRows(int r)**: setta il numero di righe pari a r;
- **void addTile(PuzzleTile t)**: aggiunge la pedina t (*tile*) al puzzle attuale;
- **PuzzleTile tile(int r, int c)**: restituisce la pedina posizionata alla riga r e colonna c;
- **ArrayList<PuzzleTile> tiles()**: restituisce la lista di tasselli del puzzle;
- **String toString()**: metodo astratto ridefinito nelle sottoclassi conferente il layout di stampa dal progetto richiesto;
- **void solve(Puzzle puzzletoSolve)**: metodo astratto ridefinito nella sola sottoclasse PuzzleSolved relativo alla risoluzione del puzzle.

2.2.2 PuzzleUnsolved

La classe rappresenta un puzzle non ancora risolto inizializzato da file di input. In tale infatti, essendo semplicemente un'entità di comodo, è stato solamente definito il metodo *String toString()* della classe base, mentre non è stato implementato il metodo *solve(Puzzle)*; tale circostanza comunque non vieta la creazione di oggetti di tipo *PuzzleSolve()* ma può creare problemi nel momento in cui si richiami il metodo clone su un oggetto di questo tipo: tale esecuzione comporterà il sollevamento di un'eccezione di tipo *AbstractMethodError* dal programma non gestita data l'attuale implementazione ed uso di tale.

2.2.3 PuzzleSolved

PuzzleSolved invece rappresenta quello che sarà il puzzle ordinato tramite l'algoritmo di ordinamento solve. Essa implementa l'interfaccia *Solvable* e dunque implementa a sua volta il metodo *solve(Puzzle)*. Di seguito i metodi di classe:

- **String toString()**: come per la precedente anche *PuzzleSolved* implementa la stampa però formattandola per colonne rappresentati il puzzle;
- **PuzzleTile tile(int r, int c)**: rappresenta un overloading del metodo di classe base. Il perchè dell'implementazione si ritrova nell'implementazione di stampa dato che per PuzzleSolved sarà necessario stampare le pedine relative all'arraylist proprio e non quello di classe base;
- **void solve(Puzzle p)**: tale è il metodo relativo al riordinamento del puzzle p passato come parametro;
- **void solveFirstColumn(Puzzle p)**: è il metodo che permette l'ordinamento della prima colonna del puzzle;
- **void solveRemaining(Puzzle p)**: tale metodo permette l'ordinamento dei tasselli rimanenti con la pre-condizione che almeno la prima colonna sia ordinata.

2.2.4 PuzzleThread

Tale classe rappresenta il thread richiesto per la risoluzione del puzzle e pertanto la classe estende la classe *Thread*.
In essa troviamo un costruttore accettante come unico parametro la riga sulla quale il thread si dovrà concentrare. Pertanto, essendo la classe creata all'unico scopo di risolvere una riga del puzzle, non avrà altri metodi se non la run relativa al thread risolvente la riga del puzzle per il quale il thread è stato creato.

2.2.5 Solveable

Interfaccia rappresentante oggetti di tipo risolvibile. Unico metodo in essa iscritto è *void solve(Puzzle puzzleToResolve)*.

2.2.6 PuzzleTile

Entità rappresentante il singolo tassello di puzzle. Essa viene inizializzata con un costruttore a 6 parametri, relativi rispettivamente all'id del tassello, contenuto del tassello e confini a partire dall'alto sino a terminare al confine sinistro rappresentanti gli id dei tasselli ad esso adiacenti.

In esso gli unici metodi descritti sono quelli restituenti i valori di classe e cioè:

- **String id()**: restituente l'id del tassello;
- **String top()**: restituente il confine nord;
- **String right()**: restituente il confine est;
- **String bottom()**: restituente il confine sud;
- **String left()**: restituente il confine ovest;
- **String character()**: restituente il valore letterario inscritto nel tassello.

2.2.7 PuzzleReader

PuzzleReader è la classe responsabile della lettura da file del puzzle. Per far ciò la classe associa ad un *BufferedReader* uno stream di input (*InputStream-Reader*) costruito da un *FileInputStream*. Essa legge il file di input, creando riga per riga i tasselli componenti il puzzle. E' chiaro che le dimensioni di colonna e riga sono facilmente ricavabili semplicemente controllandone i confini dal tassello descritti. Alla classe è passato un riferimento con il quale può settare eventuali valori del puzzle non ancora ordinato.

2.2.8 PuzzleWriter

Anche PuzzleWriter è la classe responsabile per la scrittura dei puzzle su file. In maniera più specifica per far ciò essa costruisce un buffer di lettura (*BufferedWriter*) associato ad uno stream costruito anch'esso con uno stream rappresentante il file di destinazione. Fatto ciò essa richiama il metodo toString() della classe *Puzzle* che per polimorfismo farà si che venga stampato il puzzle con le sue impostazioni di layout specifiche.

2.2.9 PuzzleSolver

E' la classe principale, in cui è descritto il main. Essa inizializza due riferimenti polimorfi rispettivamente unsolved ed solved, e ne inizializza il contenuto, stampandone infine lo stato.

3 Ordinamento

Per l'ordinamento si è ricorsi a due funzioni principali:

- *solveFirstColumn()*;
- *solveRemaining()*.

Il primo dei due, *solveFirstColumn()* non fa altro che ordinare la prima colonna del puzzle. Per farlo seleziona i tasselli aventi come pre-condizione quella di avere il confine sinistro pari a "VUOTO", marcatore relativo al confine non rappresentato da nessun tassello. Successivamente confronta l'eventuale tassello selezionato con un marcatore definente il tassello superiore sotto il quale incastrare il nuovo. Tale marcatore verrà sovrascritto ad ogni iterazione così da avere sempre come limite superiore il tassello appena inserito.

Fatto ciò, il metodo *solve(Puzzle)* richiama *solveRemaining()* funzione per l'ordinamento dei tasselli restanti. A differenza della prima consegna, in questa vi è la necessità di creare semplicemente tanti thread concorrenti quante son le righe del puzzle; dunque il metodo non farà altro che richiamare la *start()* del thread appena creato per la riga corrente.
Tale non farà altro che eseguire la *run()* del thread, la quale a differenza della precedente, si concentra semplicemente sulle colonne senza iterare per le righe dato che l'unica riga presso il quale far riferimento è definita come campo privato della classe.

4 Concorrenza

Vengono ora discusse le problematiche relative alla concorrenza:

- Deadlock;
- Starvation;
- Interferenza sui dati;

Data l'implementazione del progetto PuzzleSolver, dobbiamo dire che in questa seconda consegna situazioni di deadlock sono impossibili dato che non è stato usato nessun costrutto di lock.

Inoltre per l'interferenza sui dati anche qui possiamo dire che l'implementazione dell'algoritmo di ordinamento ha permesso di tener bene contraddistinte le aree di lavoro dei rispettivi thread, e dato che la sorgente dei dati, in questo caso il puzzle da risolvere, non muta nel tempo è stato possibile omettere eventuali lock sui dati o metodi.

Per lo starvation invece possiamo dire che i thread vengono creati tutti quanti con la stessa priorità e non essendoci come in precedenza detto lock sui dati, condizioni di attese attive sono pressoché impossibili.

5 Compilazione ed Esecuzione

Per la compilazione è stato predisposto un make file eseguibile semplicemente con il comando make all'interno della root della consegna, il quale compilerà tutti i file necessari per il funzionamento. Per lanciare il programma sarà necessario eseguire l'istruzione *sh puzzlesolver.sh input output* sostituendo ad input il nome del file ed output il nome presso il cui si vuole stampare i risultati.