



# 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.

# 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 *PuzzleUnsolved* 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 già 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()**: restituisce 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.

### 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.

### 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()*. 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 Solveable

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

#### 2.2.5 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 restituanti 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 iscritto nel tassello.

#### 2.2.6 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 (*InputStreamReader*) 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.

### 2.2.7 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 sì che venga stampato il puzzle con le sue impostazioni di layout specifiche.

### 2.2.8 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()* richiama *solveRemaining()* funzione per l'ordinamento dei tasselli restanti. Il principio di funzionamento basa la sua logica sul fatto che essendo la prima colonna già settata, basterà attaccare tanti tasselli quante son le colonne -1 aventi come confine sinistro il tassello della prima colonna precedentemente posizionata. Trovato il tassello si passa alla colonna successiva ed il ciclo si ripete, mantenendo come marcatori sinistri, i nuovi tasselli inseriti.

## 4 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 lanciando automaticamente poi il programma sui file descritti dal nome '*input.txt*' ed '*output.txt*'.

Nel qual caso non si voglia l'esecuzione automatica del file o si voglia cambiarne i file di input o output basterà eliminare le righe 2-3 del makefile, e rilanciare il comando `make`; fatto ciò sarà possibile lanciare `PuzzleSolver` con i relativi file di input ed output.