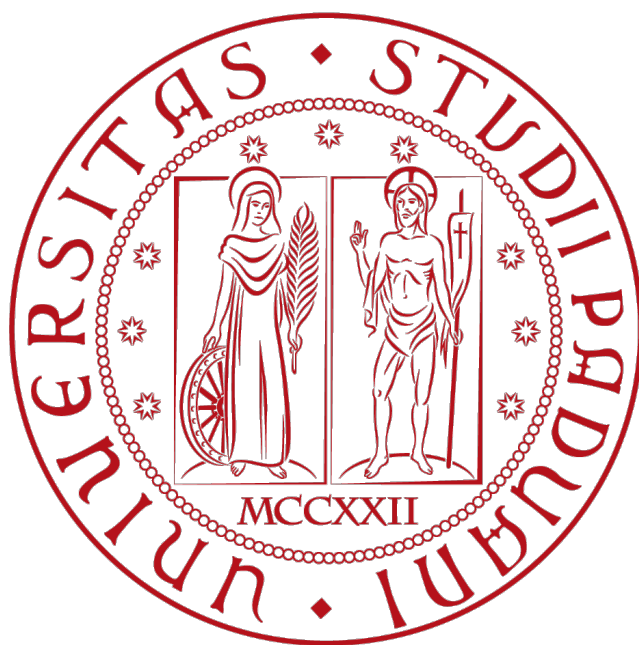


# Programmazione Concorrente e Distribuita

## Seconda Parte

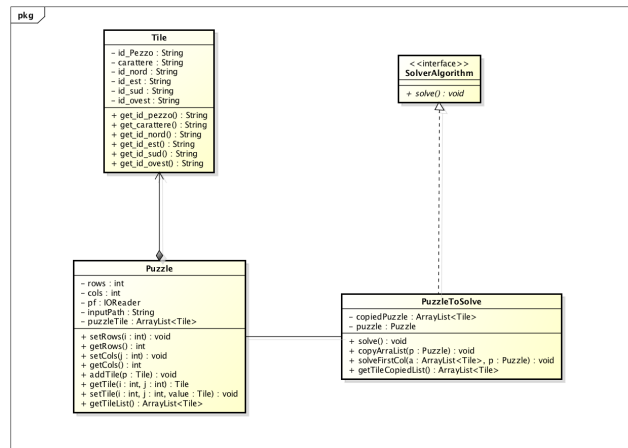
Suierica Bogdan Ionut 1008089

June 10, 2015



# 1 Cambiamenti e aggiunte

In questa sezione verranno descritti i cambiamenti apportati alla versione precedente del programma in modo tale che la logica dell'algoritmo di risoluzione del puzzle sia concorrente. I cambiamenti effettuati sono minimi, in particolare è stato cambiato il metodo *solve()* dell'interfaccia *AlgorithmSolver* e l'implementazione di tale metodo.



La classe **PuzzleToSolve** implementa l'interfaccia **SolverAlgorithm**. Oltre ad avere il compito di salvarsi il puzzle disordinato e restituirlo ordinato attraverso il metodo realizzato "solve()", fornito dall'interfaccia **SolverAlgorithm**, deve costruire l'oggetto "Puzzle" da cui verranno copiati tutti gli elementi del puzzle.

Essendo disponibile l'oggetto "Puzzle" è stato tolto il parametro alla funzione **solve()** dell'interfaccia **SolverAlgorithm** che viene comunque utilizzata per permettere una futura estensibilità del codice. È quindi possibile inserire un differente algoritmo di risoluzione, presentando quindi alternative diverse allo sviluppatore e/o utilizzatore essendo necessaria la sola implementazione dell'interfaccia e del suo metodo **solve()**.

# 2 Algoritmo di risoluzione

L'algoritmo di risoluzione è di tipo concorrente come richiesto dalla specifica relativa alla seconda parte del progetto.

Ci sono più attività concorrenti che ricostruiscono il puzzle. I vari thread coinvolti hanno un carico di lavoro uniforme e non tengono occupata la CPU inutilmente. Non ci sono problemi di interferenza e di deadlock.

Le strutture dati utilizzate non cambiano rispetto alla prima parte del progetto e vengono utilizzate sempre 2 *ArrayList* di oggetti *Tile*: uno per il salvataggio degli elementi del puzzle disordinati da file e uno per il salvataggio degli elementi

del puzzle ordinati. Al momento del salvataggio degli elementi disordinati da file viene calcolato e memorizzato il numero delle righe e delle colonne in modo da poter visualizzare l'ArrayList come un array a 2 dimensioni.

La logica della risoluzione del puzzle è molto simile alla logica della prima parte del progetto:

- **Ordino la colonna più a sinistra:** Attraverso un ciclo for che scorre le righe del puzzle inizio con la ricerca del primo elemento del puzzle che avrà id\_nord e id\_ouest uguali a "0". Gli altri elementi li trovo cercando l'elemento con id\_ouest sempre uguale a "0" e id\_nord uguale al id dell'elemento precedentemente trovato cioè il primo elemento della riga precedente.
- **Ordino il resto del puzzle** Una volta completato il passo precedente, il main thread potrà lanciare l'avvio dei thread che si occupano dell'ordinamento del resto del puzzle partendo dalla seconda colonna in quanto la prima è stata già ordinata. Verranno avviati tanti task quante saranno il numero delle righe che compongono il puzzle. Per ogni riga, partendo dalla seconda colonna, si confronterà l'id\_ouest del elemento in questione con id dell'elemento a sinistra, nel primo caso con l'elemento della prima colonna. Solo quando tutti i task avranno finito l'ordinamento il main thread potrà continuare, effettuando tutte le operazioni, e terminare la sua esecuzione.

## 3 Gestione dei Thread

Di seguito verranno descritte le conseguenze che possono presentarsi con l'avvio dei thread necessari per la risoluzione del puzzle.

### 3.1 Numero Thread

Il numero di thread cambia in base al numero di righe del puzzle pertanto il numero minimo di thread attivi può esser uno.

### 3.2 Interferenze, Deadlock, Attesa attiva

- **Interferenze** Questo fenomeno non può mai presentarsi sull'oggetto condiviso in quanto ogni accesso da parte dei vari thread in esecuzione è effettuato in sola lettura;
- **Deadlock** Questo fenomeno non può mai presentarsi in quanto il codice è stato progettato senza lock;
- **Attesa attiva** Questo fenomeno non può mai presentarsi in quanto tutte le iterazione hanno condizioni di uscita date dalle precondizione del programma.

È stato scelto di non usare nessun costrutto di concorrenza per avere una soluzione quanto semplice.