# Relazione progetto PCD – parte 1

Gabriele Pozzan
matricola 1051239

dicembre 2014

## Indice



Il programma ha come scopo la risoluzione di un puzzle a partire da un input formato dall'insieme dei suoi pezzi in disordine e dalle indicazioni su ciò che rappresentano (il carattere) e sui loro pezzi adiacenti nella direzione dei quattro punti cardinali.

## 1 - Classi

Per la risoluzione del problema sono state utilizzate una gerarchia di classi (*Fragment*, *PuzzlePiece* e *SetOfPieces*) che rappresenta i tipi di dato presenti e una classe (*Puzzle*) contentente la logica per la risoluzione.

Il puzzle del problema è visto come un insieme di frammenti (*Fragment*) i quali sono caratterizzati dall'avere dei vicini (altri frammenti i quali possono essere tessere o insiemi di tessere) e dall'essere unibili a vicenda. La classe *Fragment* è astratta: i riferimenti ad essa vengono utilizzati polimorficamente nella soluzione del problema, i metodi *merge, findNeighbor* e *print* sono concretizzati nelle sottoclassi, tutti gli altri metodi sono invece concreti (si è scelto di utilizzare una classe astratta piuttosto di un' interfaccia per poter sfruttare questa eredità di implementazione oltre che di tipo) e sono presenti due campi dati (*idSet* che contiene il nome del gruppo di appartenenza del frammento e *puzzleRef*, riferimento al puzzle che permette di invocare alcuni metodi della classe *Puzzle*).

I frammenti si specializzano in tessere (*PuzzlePiece*) e insiemi di tessere (*SetOfPieces*):

- I *PuzzlePiece* sono i dati di partenza del problema, strutturati sulla forma dei dati in input: sono caratterizzati da id, carattere, l'insieme degli id dei *PuzzlePiece* vicini e dei riferimenti ad essi (le tessere sul bordo del puzzle sono collegate a un *PuzzlePiece* di default chiamato *boundary* che rappresenta l'esterno). Hanno inoltre un id del gruppo di tessere di appartenenza (*idSet*), se non ancora parte di un gruppo tale id sarà uguale all'id della tessera.
- I *SetOfPieces* vengono formati durante la risoluzione del problema: sono caratterizzati dal loro id (*idSet*), da una lista di *PuzzlePiece* rappresentante tutte le tessere appartenenti all'insieme e da una lista chiamata *boundaryPieces* che accoglie tutte le

tessere presenti sul bordo dell'insieme (ovvero le tessere cui manca qualche collegamento coi vicini). Il campo dati *first* è destinato a contenere la prima tessera del puzzle (quella posta in alto a sinistra) ed è di utilità alla stampa finale.

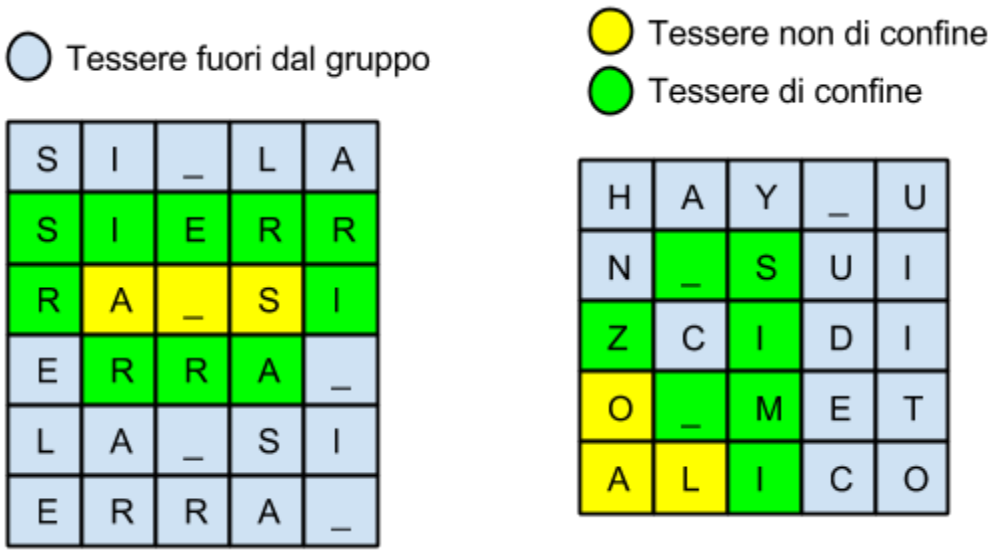


Figura 1 - Esempio di gruppi di tessere

*Nota: le tessere fuori dal gruppo potrebbero essere parte di un altro gruppo o ancora non essere state caricate dal file di input, sono mostrate come già sistemate nel puzzle per chiarezza di esposizione*

La classe *Puzzle* racchiude la logica di risoluzione del problema. Ogni *Puzzle* dispone di una lista (ArrayList) di frammenti detta *fragmentList* e di un indice (HashMap) di frammenti detto *fragmentIndex*. In *fragmentList* si avranno in ogni momento i pezzi del puzzle ai diversi stadi di collegamento, prima tessere e gruppi di tessere, poi solo gruppi, fino ad arrivare ad un solo gruppo. In questa lista una tessera non potrà essere presente più volte (ad esempio come singola tessera e come parte di un gruppo presente). L'indice *fragmentIndex* funziona diversamente: esso ha lo scopo di rendere disponibili immediatamente i vari pezzi del puzzle a partire dal loro id, quindi una tessera verrà rimossa da esso solo quando completamente collegata (facendo cadere la necessità di reperirla a partire dell'id per collegarla ad altri frammenti) e quindi non più di confine, un insieme di tessere solo quando inglobato in un altro insieme.

I dati del problema (gerarchia) e il meccanismo di soluzione (classe *Puzzle*) sono logicamente correlati, si è pensato dunque racchiuderli in uno stesso package chiamato *engine*.
Le classi *Fragment*, *PuzzlePiece* e *SetOfPieces* hanno accesso ristretto al package, mentre la classe *Puzzle* è pubblica. Questo per nascondere i dettagli di rappresentazione del problema all'esterno del package (information hiding).

## 2 - Algoritmo di risoluzione

L'algoritmo di risoluzione è ispirato a un possibile metodo di risoluzione di un puzzle fisico (schematico e semplificato, non tiene conto dell'importanza del colpo d'occhio sulle tessere e sui pezzi già formati di puzzle, ecc.):

1. Situazione di partenza: le tessere sono tutte dentro alla scatola
2. Estraggo una tessera e controllo se posso collegarla a una tessera o a un gruppo già presente sul tavolo
    - Sì : unisco e vado al punto 3
    - No : metto la tessera sul tavolo e vado al punto 3
3. Le tessere nella scatola sono finite?
    - Sì : vado al punto 4
    - No : torno al punto 2
4. Unisco tra loro le tessere e i gruppi presenti sul tavolo e ho finito.

Il metodo *solve* della classe *Puzzle* opera seguendo l'agoritmo illustrato sopra: ogni riga del file di input contiene i dati per creare un *PuzzlePiece*; dopo la creazione si verifica se nel *fragmentIndex* non sia presente uno dei vicini della nuova tessera: in caso positivo si collega e si procede a nuova creazione, in caso negativo la tessera viene aggiunta alla lista di frammenti. Si è pensato di cominciare a collegare i pezzi già durante il caricamento dei dati soprattutto in vista della futura versione concorrente dell'algoritmo (con l'idea che mentre qualcuno si occupa di caricare i dati, altri possono già cominciare a sistemarli).

Una volta creati tutti i *PuzzlePiece* si procede a unire tra loro i frammenti salvati in *fragmentList* secondo un ciclo diviso in tre fasi:

1. **Get** : viene recuperato il primo frammento salvato in *fragmentList*
2. **Find** : viene trovato un vicino di tale frammento
3. **Merge** : i due frammenti vengono uniti

Ad ogni iterazione di questo ciclo la dimensione di *fragmentList* diminuirà di uno (per l'unione dei due frammenti recuperati nelle prime due fasi).

In tutte e due le fasi dell'algoritmo (1-caricamento dati + primi collegamenti e 2-unione dei frammenti) viene sfruttato il polimorfismo dei riferimenti di tipo *Fragment*: in base al tipo dinamico sia del chiamante (sia per il metodo *findNeighbor* che per il metodo *merge*) sia del parametro attuale (solo per *merge*) vegono invocate le concretizzazioni di questi metodi astratti di *PuzzlePiece* e di *SetOfPieces* di volta in volta.

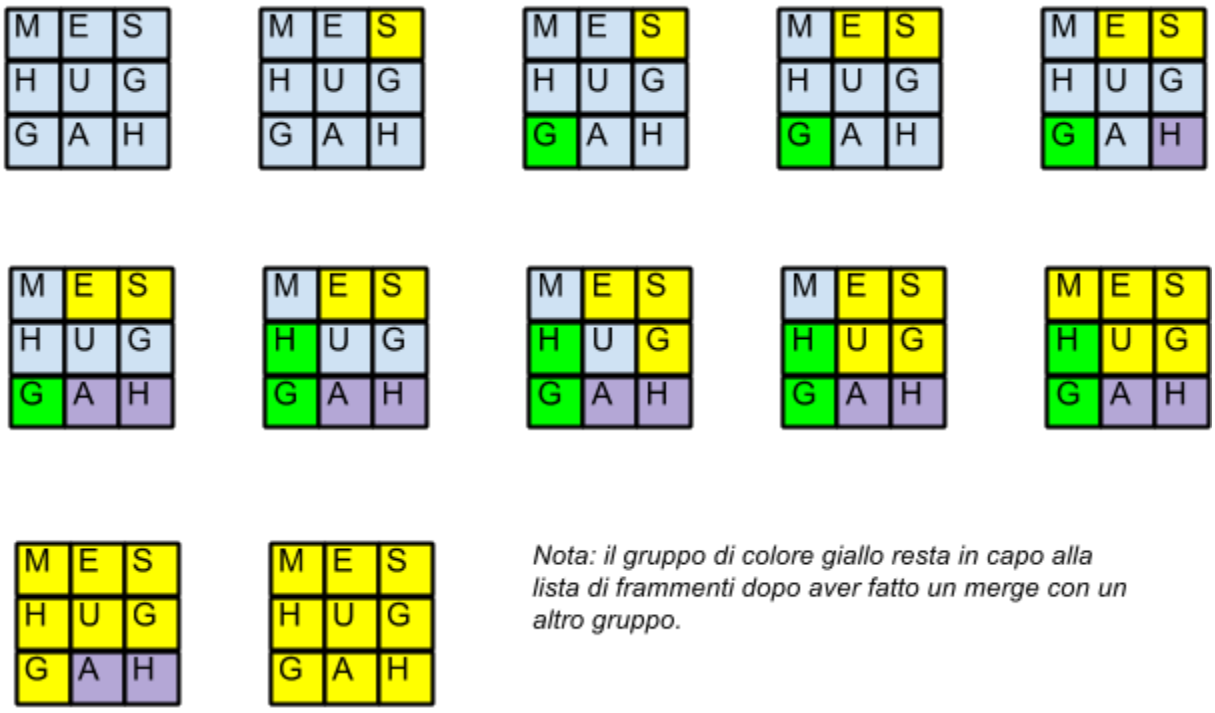


Figura 2 - Esempio di risoluzione di un puzzle

*Le tessere azzurre sono i dati ancora non recuperati dal file di input, gli altri colori identificano i diversi gruppi di tessere.*

Il polimorfismo viene anche sfruttato per la stampa: nel caso limite in cui il puzzle sia formato da una sola tessera il metodo verrà invocato il metodo *print* della classe *PuzzlePiece*, altrimenti verrà invocata la concretizzazione della classe *SetOfPieces* la quale provvederà in primo luogo (tramite il metodo privato *createMatrix*) a sistemare i dati in una lista di stringhe e poi a formare l'output secondo le specifiche.

### 3 - Test di correttezza e controllo degli errori sul file di input

Il file di input potrebbe presentare tre tipologie di errori:

- Qualche tessera ha più di un carattere, cioè dopo il primo carattere di tabulazione nel file di input trovo una stringa di lunghezza maggiore a uno: questo errore viene individuato controllando la lunghezza della stringa trovata in quel particolare punto di ogni riga
- Il file di input contiene qualche riga vuota: in questo caso caso il metodo *solve* della classe *Puzzle* tenterà di estrarre le varie parti della riga dividendole in un array, per una riga vuota questo causerà un'eccezione di tipo *IndexOutOfBoundsException* la quale viene gestita nello stesso metodo
- Uno o più id (sia nella parte di riga riguardante quello del pezzo, sia nella parte riguardante quelli dei vicini) sono sbagliati e non hanno match nel file: questo può causare due tipi di errori in base a quale id del file di input sia sbagliato, nel primo caso il ciclo while controllato dalla dimensione di *fragmentList* diventerebbe un loop infinito perché non verrebbero trovati *neighbors* del primo frammento, si controlla

dunque alla fine di ogni ciclo che la dimensione di *fragmentList* non sia maggiore o uguale a quella dell'inizio; nel secondo caso il numero di frammenti diminuirebbe linearmente ma alcuni pezzi rimarrebbero comunque "di confine", per evitare questo problema si controlla nel metodo *createMatrix* (che sistema le tessere del frammento per la stampa) che non ci siano ancora *boundaryPieces*.

Questi errori vengono gestiti invocando il metodo *error* della classe *Puzzle* il quale semplicemente stampa su video e file un messaggio esplicativo terminando l'esecuzione.

Per i test sono stati generati una serie di puzzle che prendessero in cosiderazione diverse casistiche:

- *Casi limite*: sono stati testati puzzle di un solo elemento, di una sola riga e di una sola colonna
- *Casi normali*: sono stati testati puzzle con uguale numero di righe e di colonne, più colonne che righe e più righe che colonne
- *Casi errati*: sono stati testati puzzle i cui input presentavano errori sui caratteri dei pezzi, righe vuote ed errori sugli id

I vari test hanno dato i risultati attesi.

Nota: In base alla disposizione dei pezzi nel file di input e alla posizione di un errore nell'id di un pezzo confinante (quindi un id_nord, id_sud, ecc.) l'esecuzione del programma potrebbe comunque andare a buon fine: questo quando il pezzo con il campo sbagliato in una particolare direzione venisse connesso al vicino relativo come parametro (e non come invocatore) del metodo *connect* il quale sistemerebbe i riferimenti ai vicini (*neighborsRef*) di entrambi i pezzi in modo corretto permettendo quindi al metodo *solve* e al metodo di stampa di terminare senza errori. Si è pensato di mantenere questo effetto per evitare di appesantire il programma con ulteriori controlli (bisognerebbe per ogni metodo *connect* controllare che effettivamente il vicino a cui si vuole connettere il pezzo invocante abbia questo stesso pezzo come vicino nella direzione opposta a quella della connessione).