Informazioni sul documento

Nome documento	Relazione Prima Parte del progetto PuzzleSolver
Versione documento	v.1.0.0
Data redazione	2014-12-04
Redattore	Tesser Paolo

Sommario

Lo scopo del documento è quello di fornire una presentazione del prima parte del progetto PuzzleSolver da realizzare, descrivendo e motivando le scelte attuate in questa fase.

Indice

1 Organizzazione delle classi

La struttura dell'applicativo è composta da diversi package, con all'esterno di essi la classe **PuzzleSolver** responsabile dell'esecuzione del programma e dell'istanziazione delle classi in oggetti per risolvere il puzzle.
Anche se tutte le classi all'interno di questi package lavorano bene assieme e tra loro esistono alcune dipendenze, si è deciso di tenere un minimo si separazione logica per le diverse componenti.
Nei sotto capitoli seguenti verranno illustrati i package utilizzati e le classi presenti in essi, mostrandone anche il comportamento tra di loro.

1.1 Package Puzzle

Questo package contiene le classi che gestiscono le modalità con cui viene salvato un puzzle ricevuto in input da un file di testo.
Abbiamo una classe **Tile** che rappresenta un tassello del puzzle. In essa sono solo presenti il suo identificatore e quelli dei tasselli intorno.
E' stato fatto ciò per permettere, anche se non richiesto dalla specifica, che il puzzle possa essere in futuro formato da tasselli non di caratteri.
Seguendo questa logica anche la classe **Puzzle** è stata fatta astratta e contiene solo gli attributi che sono sempre presenti in un puzzle, cioè il numero di righe e di colonne che lo compongono e l'algoritmo che andrà a risolverlo.
La classe **PuzzleCharacter** è quella effettivamente utilizzata per il nostro problema. Viene formata estendendo quella astratta **Puzzle**.
Al suo interno possiede una classe interna **TileCharacter** che estende **Tile**. Si è scelto di farla interna perché le due classi hanno una relazione logica molto stretta. Un singolo oggetto del tipo TileCharacter non avrebbe motivo di esistere all'esterno di un puzzle di caratteri.

1.2 Package Solver

Questo package contiente le classi che gestiscono la risoluzione del puzzle.
E' stato deciso di slegarla dal contesto strutturale del **Puzzle** in quanto svolgono due compiti logicamente diversi. Questo permette una più facile manutenibilità del codice e una maggiore possibilità di riutilizzo.
La presenza dell'interfaccia **SolverStrategy** permette la possibile implementazione di altri algoritmi di risoluzione senza andare a toccare le modalità introdotte nella classe **SolverAlgStrategy**. Sul client **PuzzleSolver** basterà cambiare l'algoritmo che si vorrà utilizzare. Questo, grazie al tentato utilizzo del design pattern Stratey, spiegato nel capitolo **1.4**, sarà l'unico cambiamento da effettuare.

1.3 Package FileInputOutput

Questo package contiene le classi necessarie a fare leggere un file in input e a scrivere il contenuto di qualcosa in un file in output.
Si è deciso di utilizzare un'interfaccia **FileIO** contenente il solo metodo **readContent** e non anche **writeContent** perché il primo ha una firma molto più generica e classica per la lettura di un file, permettendone così una ridefinizione in possibili sottoclassi future che potrebbero implementare un altro sistema per la lettura.

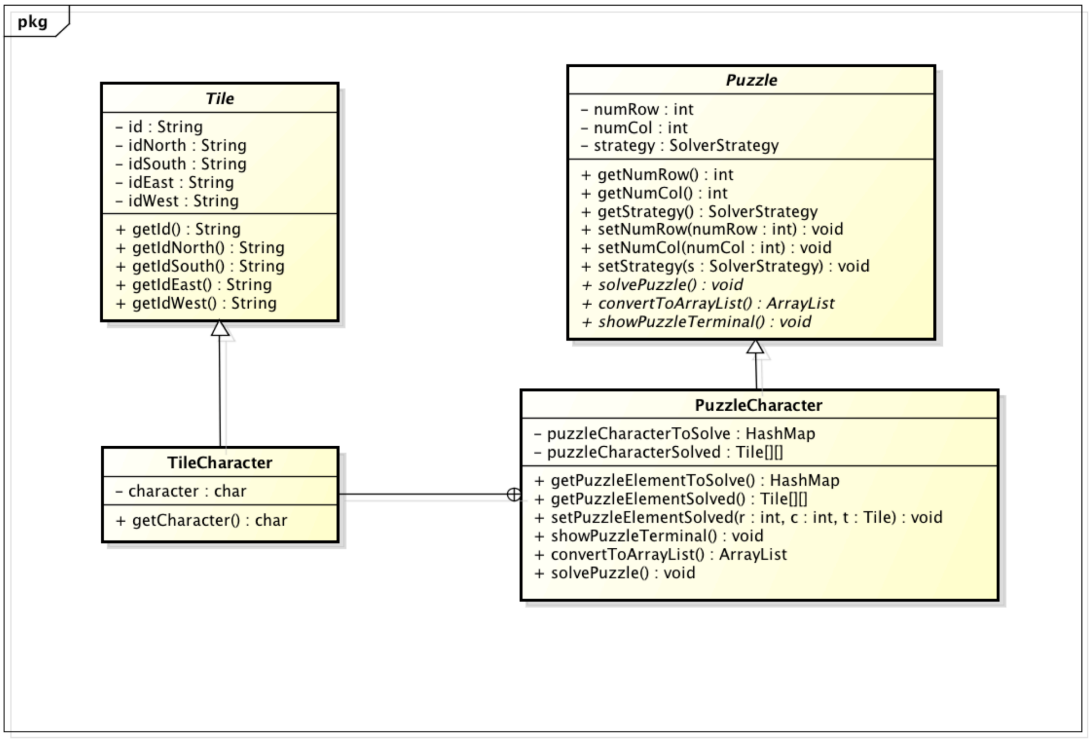

Figura 1: Package Puzzle

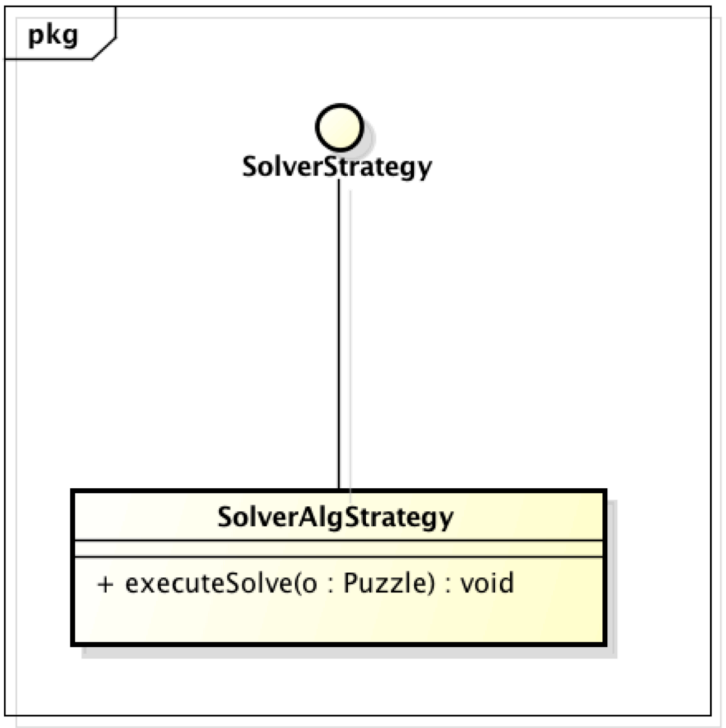

Figura 2: Package Solver

Essendo però **writeContent** più specifico per il problema del puzzle, si è deciso di tenerlo solo nella classe che implementa l'interfaccia.

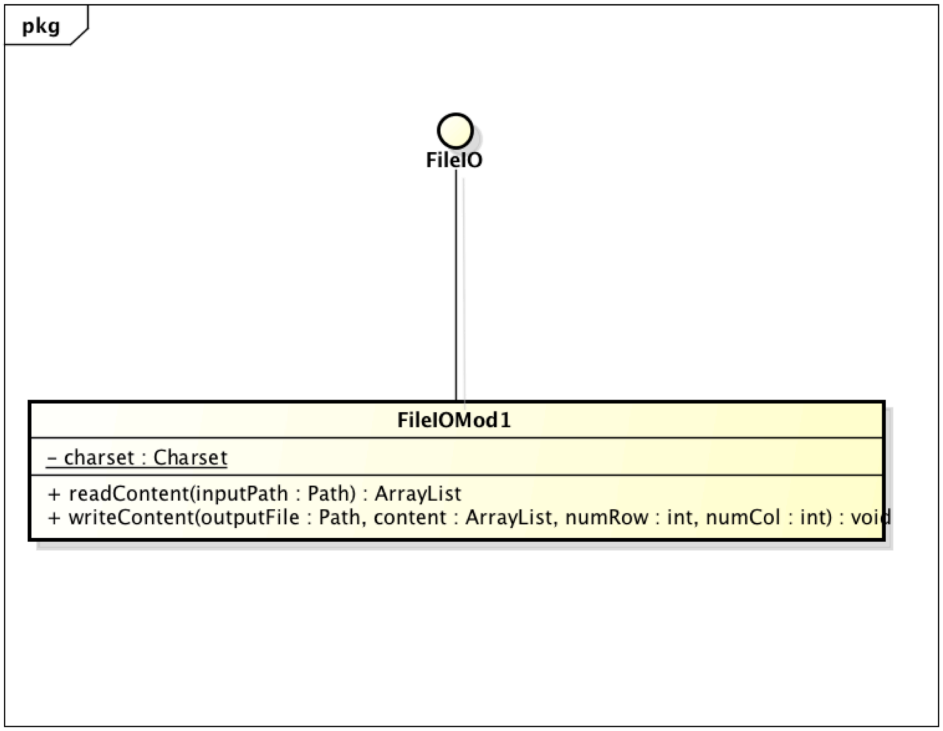

Figura 3: Package FileInputOutput

1.4 Design Pattern: Strategy

In questa sezione verrà esposto come si è cercato di implementare il desing pattern strategy tra **Solver** e **Puzzle**.
Si è scelto di utilizzarlo in quanto si possono avere molte implementazioni dell'algoritmo (alcune più efficienti di altre) e per garantire che la scelta di uno o di un altro non necessiti la ricodifica di una parte della classe **Puzzle**.
Seguendo la notazione espressa nel libro Design Patterns - Elementi per il riuso di software a oggetti di Gamma, Helm, Johnson, Vlissides viene illustrato il suo uso nel caso concreto del progetto.
Partecipanti:

- **Strategy** (SolverStrategy) : dichiara un'interfaccia comune a tutti gli algoritmi supportati, nel nostro caso per ora solo uno. Context usa questa interfaccia per invocare l'algoritmo definito da un ConcreteStrategy;
- **ConcreteStrategy** (SolverAlgStrategy) : implementa l'algoritmo usando l'interfaccia Strategy;
- **Context** (Puzzle) : Contiene un riferimento a un oggetto Strategy. In particolare viene utilizzato il campo dati 'strategy'. Viene configurato con un oggetto ConcreteStrategy;

Collaborazioni:

- Strategy e Context collaborano per implementare l'algoritmo voluto. Context passa se stesso come parametro ai metodi di Strategy. Questo consente a Strategy di invocare i metodi appropriati su Context;
- Context inoltra le richieste dai propri client verso Strategy.

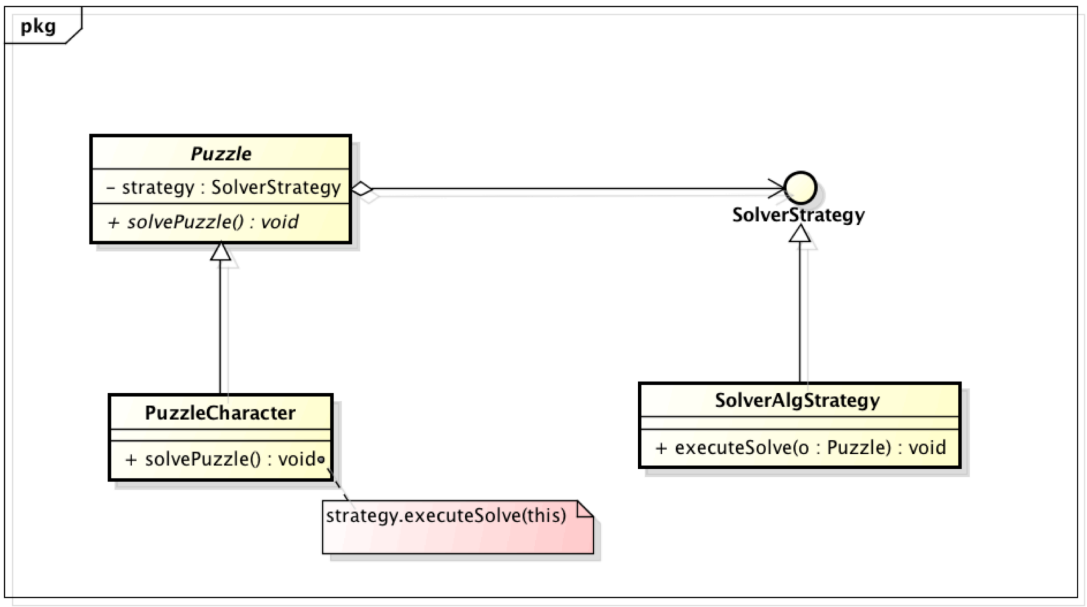

Figura 4: Design Pattern Strategy

2 Principi di OOP

In questa capitolo vengono descritte le scelte effettuate per implementare i principi della programmazione ad oggeti, in particolare quelli di incapsulamento e di information hiding.

2.1 Principio di modularità

E' stato deciso di dividere la struttura dell'applicativo in diversi package.
Questo permette una migliore manutenzione del codice in quanto in ogni package sono contenute delle classi logicamente correlate e niente altro che possa essere di interferenza con il loro funzionamento.
Si è deciso di rendere inoltre il codice il più possibile estendibile e ciò è stato realizzato tramite l'utilizzo di interfacce e di classi astratte.
Maggiori dettagli su come è stato realizzato il tutto sono illustrati nel capitolo precedente, dedicato apposta alle scelte architetturali.

2.2 Principio di information hiding

Si è cercato di realizzare questo principio esponendo il meno possibile i dati. Sono stati dichiarati quindi 'privati' tutti i membri delle diverse classi e associati dei metodi getter/setter (non per tutti i campi è stato dichiarato il metodo setter, ma solo su alcuni che ne richiedevano l'accesso in scrittura al di fuori della classe stessa).
TO DO

2.3 Principio di incapsulamento

Per avere un buon incapsulamento si è mantenuto all'interno della classe lo stato dell'oggetto e il suo comportamento. I metodi setter/getter citati precedentemente sono inoltre tutti firmati **final** per non consentire delle future ridefinizioni da sottoclassi che ne possano compromettere il naturale funzionamento.
TO DO

2.4 Principio di sostituizione

Date le classi e le interfacce presenti illustrate precedentemente abbiamo che TileCharacter è sottotipo di Tile e PuzzleCharacter è sottotipo di Puzzle. Questa struttura ha permesso, in particolare nel client **PuzzleSolver**, di scrivere codice basato sulla specifica dei supertipi e solo in pochi casi effettuare dei downcast, sempre testati preventivamente tramite l'operatore instanceof, per invocare dei metodi non presenti nelle classi base.

3 Algoritmo di risoluzione

L'algoritmo scelto per risolvere il puzzle è di tipo sequenziale, come richiesto dalla specifica di progetto.
Per arrivare alla soluzione vengono utilizzati due strutture dati come membri della classe PuzzleCharacter.
La prima struttura è la collezione HashMap, nella quale vengono salvati in ordine casuale i tasselli ricevuti in input dal file di testo. Come chiave inserisco l'id del tassello mentre come valore uso l'intero pezzo (Tile).
La seconda struttura dati è un array bidimensionale di oggetti Tile. In essa andranno memorizzati i vari pezzi in ordine corretto, questo contenitore infatti rappresenterà il puzzle nell'ordine corretto alla fine dell'algoritmo.
Di seguito vengono esposte le seguenze che vengo eseguite, correlate da dei grafici che mostrano come esso agisca sulle strutture dati utilizzate.

1. **Ricerco il primo elemento del puzzle (quello in alto a sinistra).**
   Per fare ciò scorro una sola volta la tavola hash per cercare il tassello che ha id nord e id ovest uguale alla stringa VUOTO.
   Una volta trovato salvo il pezzo nella prima posizione dell'array bidimensionale.

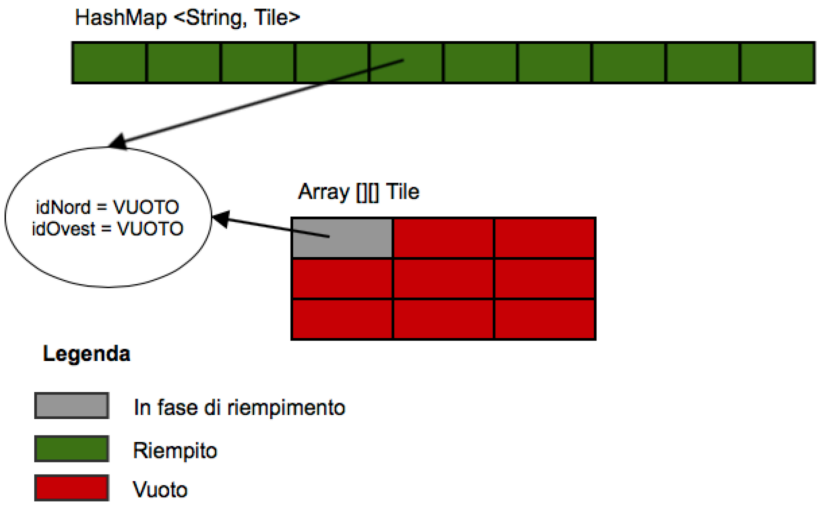

Figura 5: Algoritmo Step 1 - Ricerca del primo elemento

2. **Ordino la colonna più a sinistra (quella con i tasselli aventi id ovest uguale alla stringa VUOTO).**
   Prendo l'elemento trovato al passo successivo, mi estraggo l'id sud di esso e tramite i metodi della tavola hash mi ricavo il tassello reale a cui corrisponde.

Una volta estratto lo inserisco nella posizione corretta.
Continuo così da quello appena trovato fino a quando non trovo tutti quelli sottostanti.

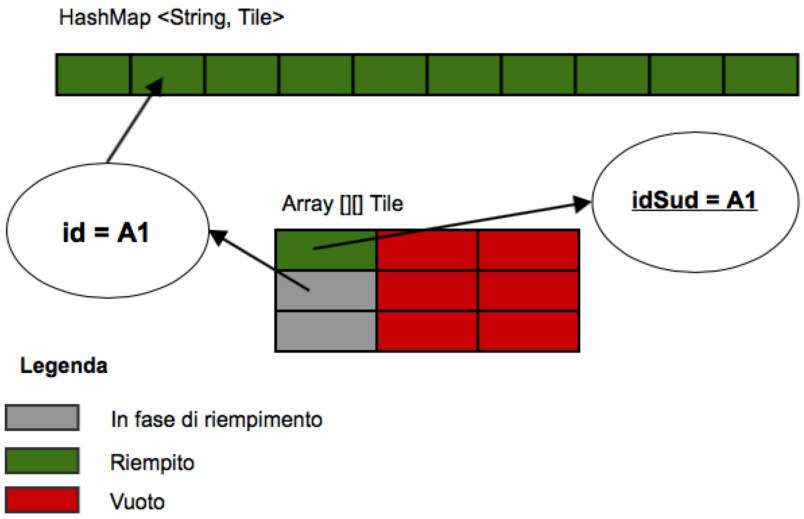

Figura 6: Algoritmo Step 2 - Ricerca della colonna più a sinistra

3. **Ordino tutte le righe.**
   Dopo aver ricavato tutta la prima colonna, eseguo, secondo lo stesso principio, anche la risoluzione per le righe, procedendo però sta volta con la ricerca del pezzo successivo a destra tramite l'id est.

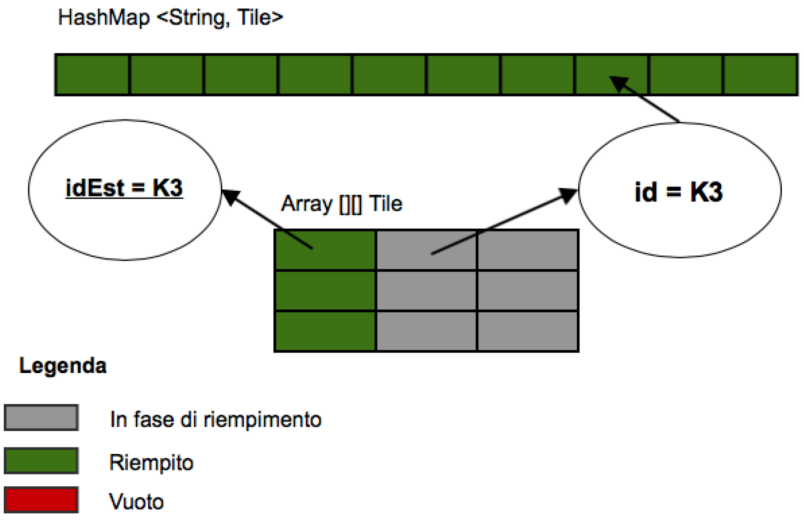

Figura 7: Algoritmo Step 3 - Ricerca delle righe

4 Controlli di correttezza

Partendo dalla pre condizione che il file che verrà usato per testare il funzionamento del programma sarà corretto, come specificato in classe dal proponente, si sono effettuati alcuni controlli per garantire che l'input sia conforme a certi requisiti di seguito elencati.

4.1 Controlli non bloccanti

Questi controlli servono per avvertire l'utilizzatore del programma che l'input specificato, pur non essendo conforme alle aspettative, è in grado di fare eseguire l'algoritmo di risoluzione e generare un qualche output che rispetti le richieste della specifica.

1. **Campo carattere non singolo** : se lo spazio riservato al carattere nella riga in input, contiene più di un carattere, viene preso in considerazione solo il primo trovato e ignorati i successivi, notificando l'accaduto sul terminale e proseguendo con l'esecuzione.

4.2 Controlli bloccanti

Questi controlli servono per bloccare l'esecuzione del programma qualora il file non sia conforme a certi requisiti. Questo perché un input scorretto pregiudicherebbe l'esecuzione di alcuni metodi della classe **Puzzle** generando dell'eccezioni non controllate.
Si è pensato quindi di creare delle eccezioni controllate che stampassero il messaggio d'errore personalizzato e bloccassero il flusso del main, saltando di fatto la chiamata

al metodo di risoluzione del puzzle.

1. **Presenza di 6 campi in ogni riga** : questo controllo serve per bloccare i file che contengono delle righe non aventi il numero esatto di campi richiesti. Quelli necessari sono 6: id, carattere, id nord, id est, id sud, id ovest e non ne può mancare nessuno di questi se si vuole risolvere il puzzle.
2. **Campi id non vuoti o formati da soli spazi** : questo controllo è stato previsto per bloccare i file che contengono id vuoti o formati solo da spazi in quanto poco conformi alle aspettative relative a un identificatore.

5 Note

5.1 Versione JVM

La versione di Java presente nella macchina utilizzata per l'implementazione del codice è quella 1.8.0_20.
Non sono stati usati però costrutti particolari di questa versione e per l'utilizzo di API si è sempre fatto riferimento alla documentazione ufficiale della 1.7.
Per essere sicuri di rispettare la specifica che richiedeva al massimo la versione 1.7, si è testata la compilazione e l'esecuzione dell'applicativo sulle macchine di laboratorio dopo avere eseguito da terminale le istruzioni presenti al seguente link: www.mashup-unipd.it.