PuzzleSolver - Relazione Programmazione Concorrente e Distribuita

Tobia Tesan - #1051819

14 dicembre 2014

Indice

Sommario

La presente relazione dettaglia e motiva le scelte progettuali dell'allegato programma `PuzzleSolver`, che realizza le specifiche della parte 1 del progetto di Programmazione Concorrente e Distribuita per l'A.A. 2014/2015 [2].

1 Vista d'insieme

Il progetto consta di 11 classi principali, di cui 3 interfacce, 1 classe astratta e 3 eccezioni, organizzate in due subpackage a seconda della funzionalità (I/O e logica) nel modo seguente:

```
puzzlesolver/
|-- core
|   |-- BasicPuzzlePiece.java
|   |-- BasicPuzzlePieceTest.java
|   |-- BFSHashmapPuzzle.java
|   |-- BFSHashmapPuzzleTest.java
|   |-- HashmapPuzzle.java
|   |-- IPuzzle.java
|   |-- IPuzzlePiece.java
|   |-- MissingPiecesException.java
|   `-- PuzzleNotSolvedException.java
|-- io
|   |-- IPuzzlePrinter.java
|   |-- MalformedFileException.java
|   |-- PlaintextPuzzlePrinter.java
|   `-- PuzzleFileParser.java
`-- PuzzleSolver.java
```

Ad esse si aggiungono due piccole classi annidate di utilità, il main e alcuni unit test.

Il perno principale del progetto sono le interfacce PUZZLE e IPUZZLEPIECE: le classi che le implementano rappresentano rispettivamente un singolo pezzo di puzzle e una classe contenitrice.

2 Dettaglio delle classi

2.1 `puzzlesolver.core`

2.1.1 IPuzzlePiece

IPUZZLEPIECE prescrive dei metodi di ispezione dell'ID del pezzo, dell'ID dei suoi vicini e alcuni semplici helper per ottenere informazioni sintetiche come `isNWCorner()`.

Per natura del task è fatta l'assunzione che gli ID del pezzo e dei vicini non possiano cambiare e dunque l'interfaccia non prescrive metodi di inserimento e modifica di tali informazioni.

L'interfaccia prescrive tuttavia dei metodi `getNorth()` ... `getEast()` e `getNorth()` ... `getEast()` che permettono di leggere e modificare riferimenti ai vicini man mano che il puzzle viene risolto.

Ogni IPUZZLEPIECE può essere visto come nodo di un grafo di grado massimo (uscente) 4 e inizialmente 0 in cui i riferimenti ai vicini sono archi.

Per esempio, lo stato finale desiderato per i quattro pezzi del puzzle 2×2 "Ciao" è quello in figura 1.

2.1.2 IPuzzle

L'interfaccia IPUZZLE espone i metodi `addPiece`, `solve`, `getSolution`, `Iterator`, `getRows` e `getCols`.

Il flusso di programma che si intende realizzare è:

1. Istanziazione

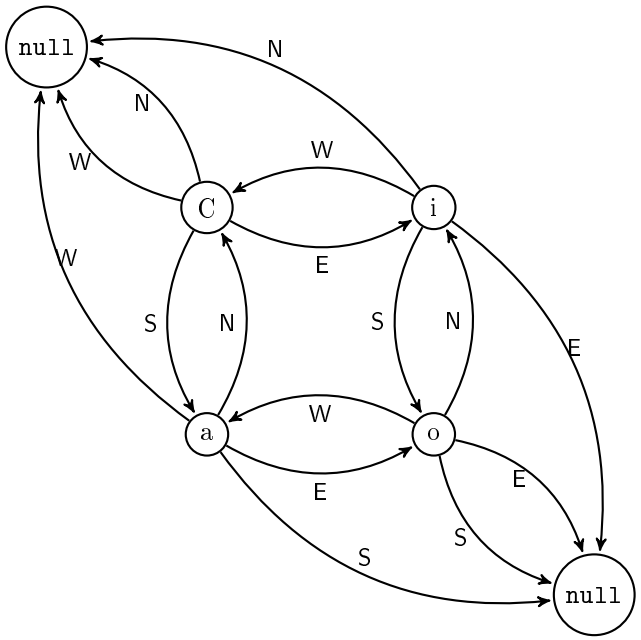

Figura 1: Un puzzle 2×2

2. Aggiunta dei pezzi con `addPiece`
3. Calcolo di una soluzione con `solve`
4. Output di una soluzione con `getSolution`, `getIterator`, `getRows`, `getCols`.

A questo scopo, i metodi di cui al quarto punto sollevano una RuntimeException chiamata PuzzleNotSolvedException - l'invocazione di tali metodi prima dell'avvenuto calcolo di una soluzione costituisce un *errore logico* da parte del programmatore. Il metodo `solve` può sollevare una MissingPiecesException se non è possibile calcolare una soluzione per mancanza di pezzi.

Non è prevista una eccezione da sollevare in caso di pezzi *in eccesso* o di errori logici più sottili nel puzzle, anche se non è nemmeno esclusa (ne è possibile escludere) una RuntimeException.

2.1.3 BasicPuzzlePiece

BasicPuzzlePiece è una triviale implementazione dell'interfaccia IPuzzlePiece costruita intorno a due array di 4 elementi. Si rimanda al codice per i dettagli.

2.1.4 HashmapPuzzle

HashmapPuzzle è una classe astratta che implementa IPuzzle.

Fornisce una Hashmap dove salvare dei IPuzzlePiece e recuperarli a costo $\approx O(1)$ nel caso medio e una serie di metodi di utilità.

```
1  public interface IPuzzlePiece {
2          String getId();
3          char getCharacter();
4          IPuzzlePiece getNorth();
5          IPuzzlePiece getSouth();
6          IPuzzlePiece getWest();
7          IPuzzlePiece getEast();
8          void setNorth(IPuzzlePiece p);
9          void setSouth(IPuzzlePiece p);
10         void setWest(IPuzzlePiece p);
11         void setEast(IPuzzlePiece p);
12         String getNorthId();
13         String getSouthId();
14         String getEastId();
15         String getWestId();
16         boolean isNRow();
17         boolean isWCol();
18         boolean isSRow();
19         boolean isECol();
20         boolean isNWCorner();
21         boolean isSWCorner();
22         boolean isSECorner();
23         boolean isNECorner();
24 }
```

Figura 2: IPuzzlePiece

Ha una classe annidata privata BasicPuzzleIterator di utilità che implementa l'interfaccia Iterator<Iterator<PuzzlePiece». Non è stato ritenuto necessario renderla standalone visto che è fortemente dipendente da BasicPuzzle, nè è stato ritenuto opportuno renderla pubblica.

Ha un solo metodo astratto: `solve`.

Con questo l'intenzione è di lasciare libertà nella scelta e nell'implementazione dell'algoritmo di soluzione/traversal.

2.1.5 BFSHashmapPuzzle

BFSHashmapPuzzle estende HashmapPuzzle. `solve()` è implementato attraverso un algoritmo di traversal breadth-first [1] in cui a ogni iterazione si impostano nel nodo i riferimenti agli oggetti corretti per gli ID.

L'algoritmo è stato scelto poichè BFS è dimostrato essere ottimale in termini di complessità e perchè appoggiandosi su una coda appare ingenuamente più semplice da parallelizzare - come suggerito opzionalmente al punto 5. delle specifiche di progetto [2].

2.1.6 MissingPiecesException

MissingPiecesException estende Exception indica che l'algoritmo di risoluzione si blocca perchè vi sono dei pezzi mancanti.

```
1 public interface IPuzzle {
2         public void addPiece(IPuzzlePiece p);
3         public void solve() throws MissingPiecesException;
4         public Iterator<Iterator<PuzzlePiece>> iterator();
5         public String getSolution() throws PuzzleNotSolvedException;
6         public int getRows() throws PuzzleNotSolvedException;
7         public int getCols() throws PuzzleNotSolvedException;
8 }
```

Figura 3: IPuzzle

2.1.7 PuzzleNotSolvedException

PuzzleNotSolvedException estende RuntimeException. Indica un errore da parte del programmatore nel tentare di accedere o iterare sulla soluzione senza prima avere invocato `solve()`, come discusso in 2.1.2.

2.2 puzzlesolver.io

2.2.1 IPuzzlePrinter

IPuzzlePrinter è un'interfaccia che definisce un singolo metodo, `print(Puzzle p) throws IOException, PuzzleNotSolvedException`, e si applica nell'intenzione a tutte quelle classi che stampano in *qualche modo* un puzzle - su file, in console, in un file grafico (e.g. SVG)

2.2.2 PlaintextPuzzlePrinter

PlaintextPuzzlePrinter implementa IPuzzlePrinter e stampa su file di testo come prescritto nelle specifiche del progetto alla sezione 3 [2].

2.2.3 PuzzleFileParser

PuzzleFileParser è una classe che fornisce un metodo statico di utilità `parseFile` che legge file nel formato prescritto nella sezione 3 delle specifiche di progetto. Si è fatto in modo che il metodo ritorni una lista di semplici struct "alla C". Si è preferito infatti mantenerlo completamente disaccoppiato dalla logica del puzzle e dalla sua rappresentazione interna al programma. Utilizzare una struct "alla C" è stata una scelta *particolarmente sofferta*, ma le alternative - es. una "vera" classe ad hoc o un array con indici numerico sono apparse eccessivamente complesse o scomode per il programmatore, in particolare perchè è facile confondere l'ordine dei punti cardinali.

2.2.4 MalformedFileException

MalformedFileException estende Exception viene sollevata quando il parser incontra un file con sintassi invalida.

3 Ragioni progettuali

Come visto nella disanima classe per classe, si è cercato di tenere le interfacce (ovvero i contratti, i cosa-fa) come perno centrale della progettazione per ottenere il massimo disaccoppiamento tra componenti e favorire il riuso del codice. Similmente, encapsulation e information hiding sono garantiti massimizzando l'interazione tramite metodi - specificati nell'interfaccia e dove necessario propri delle classi - ed evitando di esporre dettagli implementativi o attributi delle classi.

4 Test e collaudo

Nelle prime fasi di scrittura si sono approntati dei semplici unit test (`BasicPuzzlePieceTest` e `BFSHashmapPuzzleTest`) per fare da "parapetto" contro errori triviali e modifiche involontarie nel comportamento del codice. Tali unit test non hanno la pretesa di essere particolarmente curati o ben strutturati, ma solo "buoni abbastanza" per ottenere lo scopo summenzionato. Si sono inoltre utilizzati dei file di esempio - alcuni dei quali malformati o altrimenti inaccettabili (presenti nella cartella `samples`) unitamente al semplice script (`quicktest.sh`) per effettuare il collaudo finale.

Non è stato dato molto peso a errori non triviali nell'input (e.g. grafi di collegamenti non planari e dunque impossibili da ricostruire o collegamenti non corrisposti dal vicino); sono considerati undefined behaviour e evitarli è responsabilità dell'utente.

La correttezza del programma non è stata dimostrata formalmente - tuttavia, il solo algoritmo non triviale utilizzato finora (BFS) è già dimostrato in letteratura [1] e si ritiene di poter quindi approcciare il programma con una sufficiente confidenza.

5 Compilazione e utilizzo

```
$ cd parte1/
$ make
$ ./PuzzleSolver input.txt output.txt
```

oppure

```
$ cd parte1/
$ make
$ java -jar PuzzleSolver.jar input.txt output.txt
```

Opzionalmente:

```
$ make javadoc
```

Riferimenti bibliografici

[1] T.H. Cormen, C.E. Leiserson, R.L. Rivest, and C. Stein. *Introduction To Algorithms*. MIT Press, 2001.

[2] Silvia Crafa. Risolutore di puzzle - parte 1. http://www.math.unipd.it/~crafa/prog3/specifica_parte1_v2.pdf.