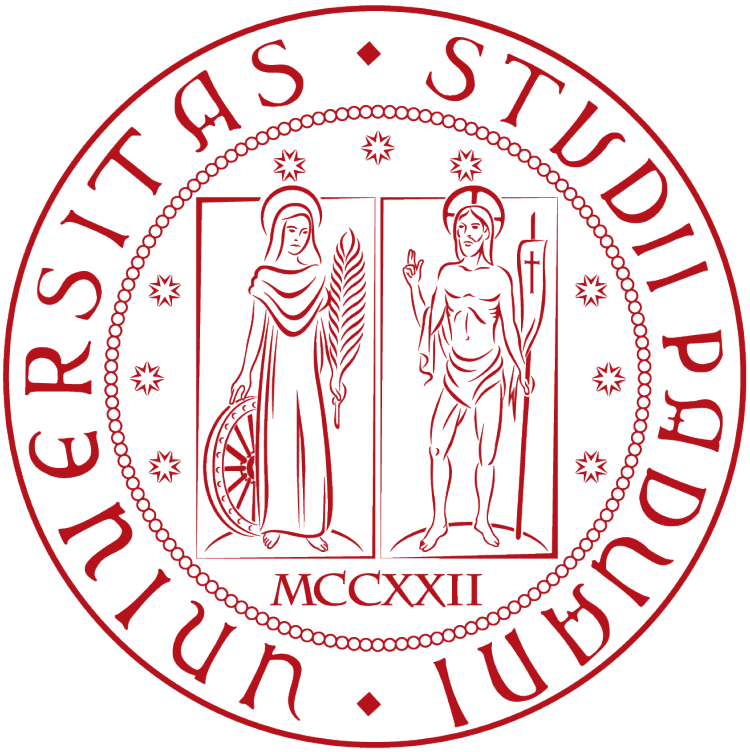

Informazioni sul documento

Nome documento	Relazione Terza Parte del progetto PuzzleSolver
Versione documento	v.3.0.0
Data redazione	2015-01-22
Redattore	Tesser Paolo

Sommario

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

Indice

1 Note introduttive

In questa terza parte, anche se la specifica non prevedeva come requisito la possibilità di avere più client che potessero eseguire la risoluzione di un puzzle sul server in maniera concorrente, si è deciso di implementare lo stesso tale funzionalità.
Il modo nel quale verrà effettuato ciò sarà illustrato nelle sezioni 2 e 3.

2 Cambiamenti e Aggiunte

In questa sezione verranno descritti i cambiamenti e le aggiunte apportate alla precedente versione per permettere al programma di essere distribuito tra un server e più client (come evidenziato nella sezione 1).
Non sono state effettuate particolare revisioni al codice sviluppato per soddisfare i requisiti della seconda parte.
É stata cancellata solo la classe **PuzzleSolver**, responsabile dell'esecuzione del programma, non più necessaria, a favore di due nuove classi: **PuzzleSolverServer** e **PuzzleSolverClient**, responsabili dell'esecuzione del programma sul server e di quello sul client. Nella sezione 3 verrà illustrato quale compito svolgono attraverso il loro main queste classi.

2.1 Aggiunte

2.1.1 Package objrem

Questo package contiene le classi che servono per permettere al client che lo desidera, di lavorare con il server per la risoluzione di un puzzle.
É presente un'interfaccia condivisa sia dal server che dal client: **ObjRem**, la quale estende la classe Remote del package rmi. Questa è l'unica cosa che sarà disponibile al client per lanciare i metodi che consentiranno la codifica del puzzle nel server.
Nel server verrà implementata l'interfaccia precedente dalla classe **ObjRemImpl**, che darà una definizione concreta dei metodi e aggiungerà dei membri. Di seguito viene spiegata la loro funzione.

Membri:

- **NUMCLIENT**: membro statico intero che serve a tenere il conto di quanti client hanno fatto richiesta al server di un identificativo. Questo garantisce che ogni richiesta sia associata ad un valore diverso e non ci sia conflitto tra client diversi;
- **clients**: membro di tipo HashMap<String, Puzzle>. Serve per immagazzinare il puzzle che ciascun client vuole risolvere, associandolo all'identificativo che viene associato al client al momento dell'avvio del programma;
- **output**: membro di tipo HashMap<String, ArrayList<String> >. Serve per immagazzinare il puzzle risolto nel formato richiesto dal client, cioè ArrayList<String>, per scriverlo poi sul file di output;
- **solveThread**: membro di tipo HashMap<String, SolveThread>. Serve per immagazzinare la lista dei thread avviati per la risoluzione dei diversi puzzle. Questo ci consente in seguito di usarlo per fare delle chiamate del metodo **join()** sul thread lanciato, per mettere in attesa un altro thread fintanto che l'operazione non è completata;

- **concreteThread**: membro di tipo HashMap<String, ConcreteThread>. Serve per immagazzinare la lista dei thread avviati per la conversione dei diversi puzzle. Questo ci consente in seguito di usarlo per fare delle chiamate del metodo **join()** sul thread lanciato, per mettere in attesa un altro thread fintanto che l'operazione non è completata.

Metodi:

- **getNewIdClient()**: ritorna un identificativo univoco di tipo stringa, formato dalla parola client più il numero del prossimo client disponibile tramite l'accesso al membro **NUMCLIENT**;
- **setClientPuzzle(String, ArrayList<String>)**: riceve un ArrayList dal client, rappresentativa del puzzle che vuole risolvere. A partire da essa crea un nuovo puzzle e lo mappa nel membro **clients**, dandogli come chiave l'identificativo del client ricevuto in input e come valore quello del puzzle appena creato;
- **solve(String, String)**: imposta la tipologia di ordinamento del puzzle in base al valore ricevuto dal client. Crea e lancia un thread di tipo SolveThread che andrà a risolvere il puzzle del client che lo ha richiesto. Inoltre, prima di eseguire il thread, lo aggiunge nel membro **solveThread**;
- **convert(String)**: crea un thread del tipo ConvertThread e lo aggiunge al membro **concreteThread**, ma lo avvia solo dopo che il thread lanciato per la risoluzione del puzzle ha terminato la sua esecuzione;
- **getOutput(String)**: restituisce un ArrayList di stringhe che servono al client per scrivere su file il puzzle nella forma richiesta dalla specifica;
- **getPuzzleCol(String)**: restituisce il numero di colonne presenti nel puzzle che il client ha richiesto di risolvere;
- **getPuzzleRow(String)**: restituisce il numero di righe presenti nel puzzle che il client ha richiesto di risolvere.

Nota: tutti i metodi elencati sono **synchronized** per garantire che non ci sia interferenza tra i diversi client possibili.
Il primo parametro formale dei metodi elencati (qualora ci fosse) è sempre la stringa identificativa del client, ricevuta all'inizio dalla chiamata del metodo **getNewIdClient()**.

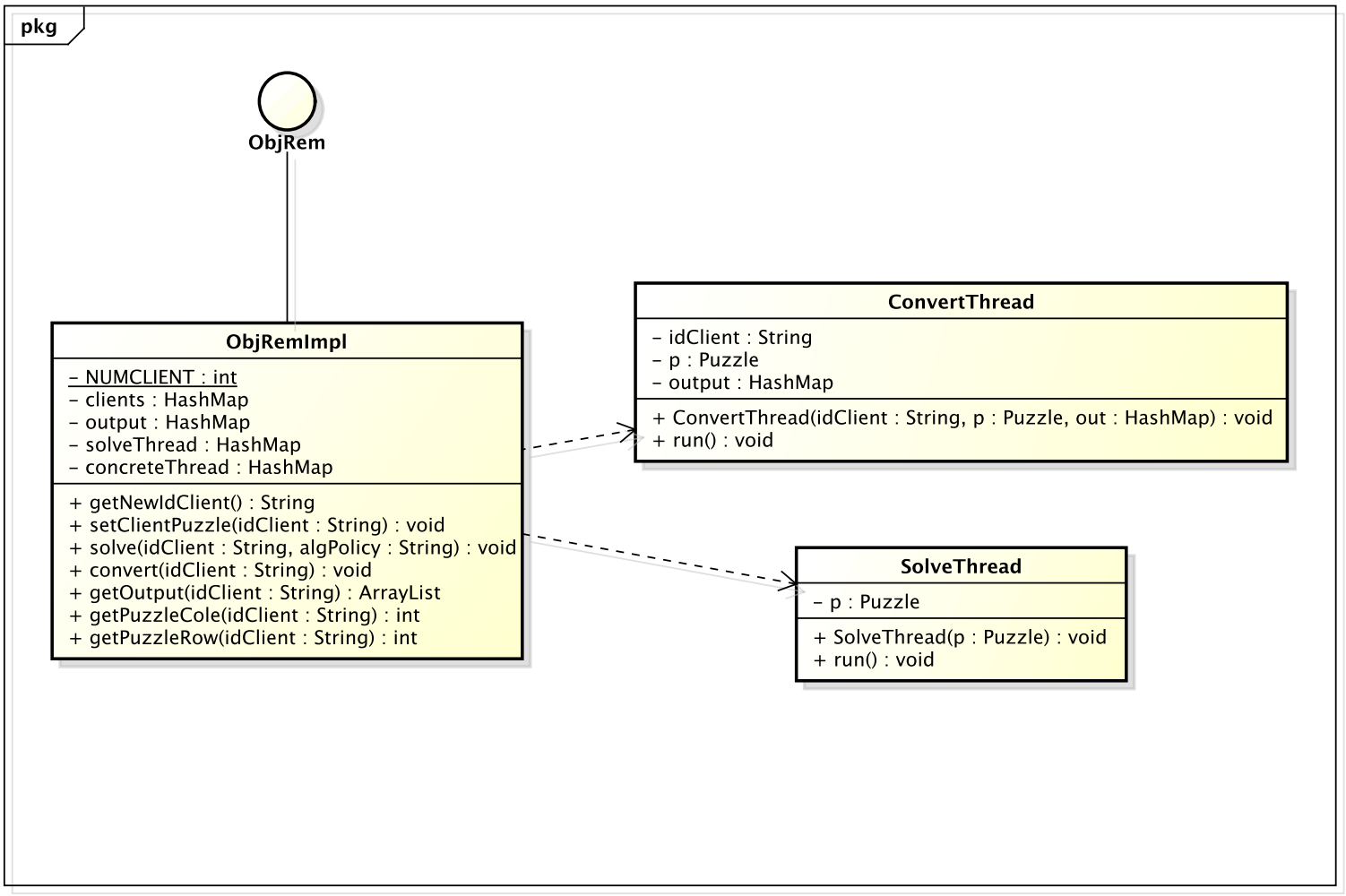

Figura 1: Diagramma delle classi - Package objrem

3 Logica di comunicazione client-server

Per la comunicazione tra i diversi client e il server il programma adotta la libreria RMI che offre un livello di astrazione più elevato rispetto al meccanismo di comunicazione tramite socket.
Il suo scopo è quello di rendere trasparenti al programmatore quasi tutti i dettagli della comunicazione su rete.
Per implementare il meccanismo offerto da RMI, ci serviamo del package **objrem** citato nella sezione precedente e dei main delle classi **PuzzleSolverClient** e **PuzzleSolverServer**.
La logica di comunicazione viene descritta separando la parte del client da quella del server, indicando per ciascuno degli attori le azioni che andranno a compiere.

Main Thread Server:

1. crea un oggetto del tipo **ObjRemImpl**;
2. pubblicizza l'oggetto appena creato sul server che ha il nome del parametro passato in input durante l'avvio del programma;
3. stampa la conferma che è pronto e si mette in attesa che qualche client effettui delle chiamate sull'oggetto remoto condiviso.

Main Client (presente un unico flusso):

1. crea un riferimento all'oggetto remoto pubblicizzato dal server, andandolo a prendere grazie al nome del server passato come uno dei tre parametri in input durante l'avvio del programma;
2. trasforma il file passato come primo parametro in un ArrayList di stringhe;
3. ricava un identificativo univoco dal server attraverso la chiamata del metodo **getNewIdPuzzle()** sul riferimento dell'oggetto remoto creato in precedenza;
4. attraverso l'invocazione del metodo **setClientPuzzle()** sul riferimento, imposta il sul server il puzzle che vuole andare a risolvere e lo associa al suo identificativo;
5. sempre attraverso il riferimento chiama il metodo **solve()** per risolvere il puzzle passato in precedenza;
6. chiama il metodo **convert()**, che esegue la conversione solo quando il metodo **solve()** ha terminato;
7. preleva il puzzle nel formato che gli serve e lo ottiene in locale dalla chiamata del metodo **getOutput()** sul riferimento all'oggetto remoto, il quale ritorna un ArrayList<String> che è **Serializable** di default;
8. scrive il contenuto precedentemente ottenuto dal server sul file di output passato come secondo parametro durante l'avvio del programma.

Nota: i metodi che vengono chiamati dal punto 2 a 7 vengono lanciati dal main del client, ma generano l'avvio di un thread sul server, non in conflitto però con il main thread dello stesso server.

Per cercare di mantenere un livello di concorrenza tra i diversi client sul server, vengono utilizzate altre due classi che estendo **Thread** e che vengono, concretizzate e avviate, nei metodi di **ObjRemImpl**.
SolveThread Server:

1. **Creato e avviato dal**: metodo **solve()**;
2. **Avviato quando**: viene chiamato il metodo;
3. **Scopo**: risolvere il puzzle passato come parametro al thread.

ConcreteThread Server:

1. **Creato e avviato dal**: metodo **concrete()**;
2. **Avviato quando**: il thread del tipo SolveThread, del client che richiede la conversione, ha terminato l'esecuzione;
3. **Scopo**: convertire il puzzle nel formato che servirà al client per la scrittura sul file di output.

Note: il numero di thread che viene avviato dipenderà da quanti client si connetteranno per risolvere il loro puzzle.
Si poteva implementare diversamente, magari attraverso un Thread Pool, ma non essendo richiesto dalla specifica si è deciso di limitarsi ad una cosa più semplice.
Inoltre potevano essere implementati anche dei metodi per fare un po' di pulizia sulle HashMap varie, quando il client ha terminato la sua esecuzione.

4 Robustezza

Nel programma la robustezza è stata gestita in maniera approssimata.
Di seguito però viene illustrato ciò che è stato fatto, utile per ampliare la gestione di eventi che possono verificarsi nella rete o in caso di caduta o del server o del client.
In particolare non sono stati previsti metodi per recuperare i dati persi.

4.1 Lato client

Se cade il client, il server non dovrà fare nulla, ma il client dovrà salvarsi in qualche modo il suo identificativo e il punto nel quale è riuscito a compiere l'ultima istruzione in maniera completa sul server prima di cadere.
Un approccio minimo viene effettuato tramite l'uso di una variabile statica nella classe **PuzzleSolverClient** denominata **STEP**.

4.2 Lato server

Se cade il server, dovranno essere presenti dei meccanismi per salvare i dati elaborati completamente per i diversi client.
Una volta salvati, il client, con la stessa variabile **STEP** citata in precedenza, potrà sapere fino a che punto il server ha effettuato il lavoro richiesto e se vuole potrà ripartire da la, una volta che il server sia tornato attivo.
In questa versione del programma, se il server cade, viene solo stampato su terminale il punto dal quale è caduto.

5 Note sulla compilazione

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: http://www.studenti.math.unipd.it .

5.2 Compilazione

Dalla root principale consegnata è possibile avviare il comando per la compilazione sia dei file necessari al server sia quelli necessari al client attraverso l'istruzione **make**.
Se si vuole lanciare il programma, testandolo con degli input definiti dal fornitore per provare l'applicativo, è possibile eseguire sempre da root i seguenti comandi:

1. **make loadserver**: il quale come prima cosa avvierà il **registro rmi** e poi lancerà il programma lato server, prendendo come parametro una stringa di testo che rappresenta il nome del server;

2. **make loadclient**: il quale lancerà il programma lato client prendendo tre file fissi, quali il file di input, quello di output e una stringa di testo che rappresenta il nome del server.n

Nota: sia se si decida di avviare il programma lato server tramite make o tramite script bash, il registro rmi dovrà essere chiuso precedentemente.

Se si vuole invece lanciare il programma con dei file di test personalizzati e un server qualunque è possibile eseguire i seguenti script bash che lanceranno l'applicativo (la compilazione andrà effettuata precedentemente come descritto nella prima parte di questa sezione):

1. **bash puzzlesolverserver.sh "nome del server"**: questo script riceve come input il nome del server. Lanciandolo verrà eseguito il comando per avviare il registro rmi e in seguito quello per avviare il main del server;

2. **bash puzzlesolverclient.sh input output "nome del server"**: questo script riceve come input tre parametri, quali file di input, file di output e il nome del server. Lanciandolo verrà eseguito il comando per avviare il main del client.

Attenzione: lo script bash: puzzlesolverclient.sh, che lancia l'esecuzione del comando *java* sul main del client principale, va ad eseguire il comando in un livello inferiore rispetto ad esso. Bisogna fare quindi attenzione al percorso del file in input che potrebbe generare un'eccezione qualora non fosse corretto.