COMMENTO DEL DOCENTE AL TEMA D'ESAME DEL 14/06/2011

PREREQUISITI

Il corso di Fondamenti T2 dà per acquisita da Fondamenti T1, che ne costituisce il necessario prerequisito, la capacità di impostare algoritmi anche non banali, indipendentemente dal linguaggio di codifica e dalle specifiche strutture dati utilizzate.

PROLOGO AL TEMA D'ESAME

Il compito si pone principalmente come variante di quello dei voli aerei ampiamente discusso in aula e svolto in laboratorio durante varie esercitazioni, per una durata complessiva superiore alle 6 ore; la parte grafica, invece, è del tutto analoga a quella dell'esercitazione "agenda", anch'essa discussa in aula e svolta in laboratorio per molte ore.

MODELLO DEI DATI

Sebbene il dominio dell'applicazione possa a prima vista apparire complesso, solo una delle sette classi (***Train***) dev'essere in realtà realizzata, le altre essendo fornite nello Start Kit. Si tratta invero di una classe molto standard, con alcune semplici proprietà e metodi di accesso: l'unico metodo che richieda un minimo di ragionamento è ***travelDuration***, che tuttavia si trova a operare su dati (la lista delle fermate) già disponibili in quanto passate al costruttore dell'istanza. La logica di tale algoritmo è molto semplice:

- con un primo ciclo sull'insieme delle fermate (stops), si cerca quella il cui nome di stazione corrisponda alla stazione di partenza richiesta, e una volta trovata si memorizza il corrispondente orario di partenza;
- con un secondo ciclo identico al precedente, si cerca la fermata il cui nome di stazione corrisponda alla stazione di arrivo richiesta, e una volta trovata si memorizza il corrispondente l'orario di arrivo;
- si calcola infine la differenza fra i due orari, con una banale sottrazione.

In realtà, può bastare un unico ciclo, in cui cercare contemporaneamente sia la stazione di partenza che quella di arrivo: in tal caso occorre prestare attenzione però a non considerare erroneamente treni che vadano nelle direzione opposta – situazione comunque facilmente identificabile dal tempo di viaggio, che in tal caso risulterebbe negativo. In definitiva, dunque, i primi 6/30 del compito si riducono a poco più che scrivere una funzione con un ciclo con alcuni confronti al suo interno.

PERSISTENZA

La prima cosa richiesta è implementare la classe ***CityStations*** che deve leggere una serie di oggetti serializzati da un file binario. Poiché il numero di oggetti nel file è noto e fornito come intero all'inizio del file stesso, il ciclo di lettura si riduce a un *for* che effettui esattamente *N* readObject, analogo a quelli visti e discussi sia a lezione sia in molte esercitazioni. Il testo non precisa in che struttura dati debbano essere posti gli oggetti letti: tuttavia, il diagramma UML contiene un suggerimento implicito a creare una mappa (metodo privato ***createMap***), che "casualmente" potrà essere sfruttata per implementare poi in un attimo il metodo pubblico ***getMap*** – di fatto, in una sola istruzione *return*. Tale mappa rende altresì immediato implementare le due versioni overloaded del metodo ***getStations,*** in quanto una si appoggia all'altra, e quest'ultima estrae semplicemente l'elenco delle stazioni dalla mappa.

Per quanto riguarda il metodo ***getCity,*** che deve restituire la città relativa a una stazione, ci sono due possibilità – una più rapida, che però presuppone una mappa extra, e una più classica, che non la prevede. Nel primo caso, si approfitta di ***createMap*** per creare una mappa extra `mx` (*stationName*, *city*), grazie alla quale il metodo ***getCity*** può poi essere implementato tramite `mx.containsKey(station) ? mx.get(station) : null`. Nel secondo caso, occorre cercare nella mappa *(nomecittà, stazioni)*, tramite `map.containsKey(name)`, il nome della città e poi ricostruire l'istanza di ***City*** richiesta. Per tenere conto del caso in cui il nome della città contenga spazi, la soluzione ottimale dovrebbe cercare dapprima solo una parola, e solo in un secondo momento (se la prima non basta), le parole successive.

La successiva lettura del file di testo è del tutto analoga a quelle viste a lezione e nei compiti svolti in laboratorio: l'unico punto degno di nota è la mancanza di spazi fra la tipologia di treno e il suo numero (es. R2135), peraltro facilmente affrontabile con l'approccio più volte spiegato a lezione di considerare l'insieme delle cifre "0123456789" come separatore del token [questo stesso esempio è presente nelle slide del package di I/O, consultabili on-line anche durante l'esame, precisamente alla slide n.69, e ribadito poco oltre, alle slide n.78-79].

Si tratta dunque, in definitiva, di predisporre in *readAll* un classico ciclo di lettura, che – *come da suggerimento implicito nel diagramma UML* – potrà appoggiarsi a una funzione *readTrain* per la lettura del singolo treno, aggiungendo via via le nuove istanze a una lista. A sua volta, *readTrain* potrà utilmente appoggiarsi a *readDays* [analoga a quella dell'esercizio voli aerei], *readStops, readStation* ed eventualmente *readTime.* Queste funzioni sono tutte sostanzialmente analoghe e gestibili tramite opportuni StringTokenizer: al termine si saranno conquistati altri 6/30, più 1/30 "regalato" per la definizione (banalissima) dell'eccezione ***BadFileFormatException*** .

Ciò conclude la prima parte del compito, del valore di 17/30, realisticamente fattibile in 1h30 (2h al più).

GUI: CONTROLLER

Questa parte si appoggia in larga misura a quanto svolto in precedenza: in particolare, il metodo ***getSortedCities*** si implementa in una riga, in quanto l'insieme delle città è ottenibile direttamente, tramite il metodo ***keySet***, dalla mappa già costruita; basta quindi ordinarlo tramite `Collections.sort` e infine restituirlo.

Il metodo ***searchTrains*** richiede un poco più di ragionamento, ma di fatto comporta di iterare sull'insieme di tutti i treni, identificando quelli che a) circolano nel giorno richiesto, b) viaggiano fra le due città richieste. La condizione a) è semplice, potendosi scrivere come `t.getDays().contains(dayOfWeek))` essendo ovviamente `dayOfWeek` l'istanza di `DayOfWeek` corrispondente alla data fornita come argomento. La condizione b) può essere verificata in due modi: o in forma esplicita (controllando che le stazioni di partenza e arrivo esistano fra le fermate del treno, e la prima preceda la seconda: più pulita ed elegante, ma un po' più lunga) o in forma implicita (un poco meno elegante, ma più veloce), sfruttando il metodo ***travelDuration,*** che restituisce un oggetto ***Time*** corrispondente a una durata positiva solo se il viaggio da A a B è possibile. In tal caso, basta memorizzare questa coppia di informazioni in una mappa (da restituire al termine del ciclo); diversamente, quel treno non interessa e va scartato. (Stima tempi: 30-45 minuti max)

GUI: INTERFACCIA UTENTE

Questa parte è analoga a quella dell'esercizio "voli aerei" sviluppato nel corso, a cui ci si poteva ampiamente ispirare. L'essenza dell'algoritmo sta nel ciclo di ricerca, che può essere così schematizzato: detto P l'insieme delle stazioni della città di partenza, e detto A l'insieme delle stazioni della città di arrivo, cercare – tramite la ***searchTrains*** del controller – tutti i treni fra una qualunque stazione $p \in$ P e qualunque stazione $a \in$ A, nella data indicata:

```
Set<Station> departureStations =  cities.getStations(departureCity);
Set<Station> arrivalStations =    cities.getStations(arrivalCity);
...
for(Station dep: departureStations)
  for(Station arr: arrivalStations) {
          Map<Train, Time> m = controller.searchTrains(dep, arr, date);
          if(m.size()>0) // visualizza dati
  }
```

La stampa finale su file è a sua volta un banale ciclo di scrittura con *PrintWriter*.
(Stima tempi: non oltre 1 ora.) Ciò conclude il compito.

COMMENTO FINALE

Il compito poteva apparire lungo e complesso, ma lo diveniva solo se si ci metteva a "scrivere codice a capofitto" senza prima riflettere costruirsi un'idea chiara delle relazioni fra le entità e delle funzionalità che ciascuna offriva. Conviene sempre dedicare almeno 20-30 minuti, all'inizio, a ragionare sul testo e a capire bene quali dati sono disponibili e dove, quali servizi sono offerti e da chi, come è sempre stato fatto durante tutte le esercitazioni. Può essere utile in tale fase farsi un piccolo riassunto e/o schema su carta, se ciò collima con le proprie abitudini; gli strumenti di sviluppo (Eclipse) potrebbero essere lasciati utilmente chiusi in questa fase, in quanto *non si guadagna tempo, ma anzi se ne perde, cominciando subito a scrivere codice senza riflettere prima in modo approfondito.*

Il compito non va preso come una "lotta contro il tempo": al contrario, occorre sforzarsi di mantenere la calma, *ricordando che il compito è costruito per essere risolto con intelligenza (ivi inclusi i suggerimenti impliciti nel diagramma UML..), non per creare difficoltà inutili*. Se una cosa appare (troppo) complessa, forse è perché la si sta guardando dal lato sbagliato.. e forse non si stanno sfruttando appieno funzionalità già esistenti e fornite.