COMMENTO DEL DOCENTE AL TEMA D'ESAME DEL 27/06/2011

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

A differenza del compito precedente, molto semplice come modello di dati ma impegnativo sul lato della persistenza, questo compito propone un'applicazione in cui l'equilibrio è invertito: a una persistenza piuttosto semplice e standard si contrappone un modello dei dati più complesso, ancorché la parte richiesta sia un'unica classe. La parte grafica è molto semplice e standard, grazie ai componenti già forniti.

MODELLO DEI DATI

Sebbene il dominio dell'applicazione possa a prima vista apparire complesso, solo una classe (***BilancioFamiliare***) dev'essere realizzata: le altre sono fornite nello Start Kit. Tre sono gli aspetti da considerare: i dati ricevuti dal costruttore, le strutture dati da usare, e gli algoritmi alla base dei metodi chiave (il costruttore, ***getTotaleEntrate/Uscite***, ***getSaldo***, ***checkVincoli***).

Per quanto riguarda i dati ricevuti dal costruttore, il testo dice chiaramente che si tratta di due collezioni rispettivamente di conti e movimenti: *non dice che i conti contengano già i movimenti*, e d'altronde se così fosse non si comprenderebbe perché passarli a parte – basterebbero i conti da soli. Al contrario, appare di cristallina evidenza leggendo il testo che è compito dell'applicazione verificare che ogni movimento sia lecito e solo in tal caso inserirlo contemporaneamente in entrambi i conti coinvolti: perciò, *il costruttore di* ***BilancioFamiliare*** *deve anche, dopo aver predisposto le strutture dati necessarie, aggiungere tutti i movimenti ai rispettivi conti.* Inoltre, poiché il testo specifica che occorre gestire anche un conto *virtuale* "esterno", il costruttore di ***BilancioFamiliare*** deve provvedere anche a istanziare tale conto e aggiungerlo alla struttura dati usata per i conti; da notare che esso NON può fare già parte della collezione di conti ricevuta dal costruttore, appunto perché si tratta di un conto virtuale (finto, a soli fini di comodo) e non reale (nel dubbio, basta comunque riflettere sulla provenienza della collezione di conti: basta poco per verificare che le rispettive descrizioni sono lette da file di testo in fase di gestione persistenza.. e nel file *non c'è mai la descrizione del conto "esterno", né può esserci vista la natura "omnicomprensiva" e virtuale dello stesso).*

Per quanto riguarda le strutture dati, l'unica indispensabile è un qualche tipo di collezione di Conto: *non* è invece necessaria una struttura ad hoc per i movimenti, essendo essi destinati soltanto a essere inseriti nei conti – o ad essere scartati se inammissibili per violazione dei vincoli. "Guarda caso", il testo suggeriva l'utilizzo di una opportuna mappa: in effetti, scegliendo una ***HashMap<String, Conto>*** per memorizzare i conti insieme ai loro nomi, tutta l'algoritmica ne risulta semplificata. Scegliendo invece una lista, tutti i metodi di accesso e ricerca risultano più complessi e inefficienti, perché viene a mancare un modo pratico per recuperare un conto a partire dal suo nome, ciò che in una applicazione come questa implica cicli di scansione continui, con grave impatto sia sulla leggibilità del codice che sulle prestazioni.

E' naturalmente possibile introdurre una ulteriore struttura dati per i movimenti, anche se non necessaria: in tal caso però occorre prestare attenzione a quale scegliere perché i movimenti sono destinati a essere confrontati da ***MovimentiComparator***, che definisce un criterio di confronto *basato unicamente sulla data* (perché così è utile che sia per la parte seguente dell'applicazione); come effetto collaterale, però, tale comparatore considera due movimenti identici (e quindi uno "duplicato" dell'altro) se avvengono nello stesso giorno, *indipendentemente dal fatto che coinvolgano conti diversi, importi diversi, etc.* Ciò impone di NON scegliere un ***Set*** (o ***SortedSet*** o ***TreeSet***) per tale ipotetica struttura dati di supporto, altrimenti il sistema gestirà un solo movimento per giorno.
E' appena il caso di notare che *nulla di tutto ciò può accadere se si segue il consiglio di usare una mappa..* che semplifica anche tutta l'applicazione. Pertanto, adotteremo questa scelta nel seguito di questo commento.

Per quanto riguarda la parte algoritmica:

- l'aggiunta di un movimento al ***BilancioFamiliare*** deve preventivamente verificare che i conti indicati come sorgente e destinazione nel movimento esistano, lanciando ***IllegalArgumentException*** in caso contrario; dopo di che, il vero inserimento può essere delegato all'omonimo metodo di ***Conto***. Da notare che se si è scelto di mantenere i movimenti in una struttura dati ulteriore, per coerenza è necessario inserire il movimento anche in essa, ma *solo dopo* che abbia avuto successo l'inserimento nei due conti coinvolti; ragione di più per evitare tale inutile struttura dati col relativo appesantimento gestionale.
- i due metodi ***getTotaleEntrate***/***Uscite*** possono essere implementati in due modi: o cercando tutti i movimenti da/verso "esterno" e sommando il corrispondente importo (ottenibile col metodo ***getDenaroMosso***), oppure

osservando più semplicemente che *tutte le entrate dall'esterno non sono altro che tutte le "uscite" del conto virtuale Esterno,* e dualmente *tutte le uscite verso l'esterno non sono altro che tutte le "entrate" del conto virtuale Esterno* (dualità nota anche come principio di pareggio del bilancio). Alla luce di ciò, è possibile evitare qualunque ciclo di calcolo e accumulo, definendo semplicemente ***getTotaleEntrate(inizio, fine)*** come `esterno.getTotaleUscite(inizio, fine)` e analogamente per ***getTotaleUscite***.

- il metodo ***getSaldo*** deve soltanto calcolare la differenza fra i due totali, usando come data iniziale la minima possibile (`new Date(0)`)
- i due metodi ***getMovimenti*** devono restituire un insieme ordinato: il modo di realizzarli dipende molto dalla struttura dati sottostante. Se si è usata una mappa, qui occorrerà un minimo di fatica extra per recuperare tutti i movimenti dei vari conti, evitando di averli tutti duplicati (a tal fine è utile appoggiarsi a qualche tipo di Set); se invece si è mantenuta una struttura esplicita, basterà ordinaria (se già non lo era) e restituirla.
- infine, ***checkVincoli*** deve semplicemente invocare l'omonimo metodo sulla sorgente e sulla destinazione dello specifico movimento passato, accumulando e restituendo gli eventuali messaggi d'errore ottenuti.

Ciò conclude i primi 12/30 del compito.

PERSISTENZA

Come già accennato nell'introduzione, quest'area stavolta è piuttosto standard: ***MyMovimentiManager*** definisce due classiche funzioni ***leggiMovimenti***/***scriviMovimenti***, che si riducono alla singola ***readObject***/***writeObject*** *in quanto i movimenti non vengono letti/scritti singolarmente, ma come collezione*: ciò è esplicitamente detto dal testo ["L'interfaccia ***MovimentiManager*** modella entità in grado di leggere/salvare una lista di movimenti"] e d'altronde in caso contrario la lettura sarebbe problematica, non potendo sapere quando fermarsi poiché il numero di oggetti nel file non è noto e, anzi, può aumentare nel tempo. Poco da dire anche su ***MyContiReader***, la cui ***readAll*** deve vedersela con un formato di file molto "basic", a linea singola e con pochi valori.

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, i metodi ***save*** e ***load*** si basano su ***scriviMovimenti*** e ***leggiMovimenti***, i vari ***getConto*** e ***getBilancio*** sugli analoghi metodi di ***Bilancio***, ***openMovimenti*** sulla ***showUI*** di ***MyMovimentoController*** (fornita)

GUI: INTERFACCIA UTENTE

Anche questa parte si appoggia molto a quanto fornito nello start kit: in particolare gran parte della complessità grafica della previsione è incapsulata in ***SummaryPanel*** (fornita), la qual cosa semplifica molto la realizzazione di ***MainFrame.*** Punto chiave a questo riguardo è notare come si sfruttino due pannelli identici (sopra e sotto), da mantenere ovviamente coerenti nel tempo (utile un metodo refresh che invochi refreshData di entrambi): il resto sono soltanto bottoni classici con relativi ascoltatori di eventi. In particolare, un unico actionPerformed può facilmente discriminare in questo caso fra le tre sorgenti di eventi (pulsante SALVA, pulsante VISUALIZZA MOVIMENTI, scelta del conto nelle casella a discesa dei vari conti), scatenando la corrispondente operazione del controller.
(Stima tempi: non oltre 1 ora.) Ciò conclude il compito.

COMMENTO FINALE

Come sempre, il compito poteva apparire lungo, 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.