

ESAME DI FONDAMENTI DI INFORMATICA T-2 del 9/2/2025

Proff. E. Denti – R. Calegari – A. Molesini

Tempo a disposizione: 3h30

NOME PROGETTO ECLIPSE: CognomeNome-matricola (es. RossiMario-0000123456)

NOME CARTELLA PROGETTO: CognomeNome-matricola (es. RossiMario-0000123456)

NOME ZIP DA CONSEGNARE: CognomeNome-matricola.zip (es. RossiMario-0000123456.zip)

NOME JAR DA CONSEGNARE: CognomeNome-matricola.jar (es. RossiMario-0000123456.jar)

Si devono consegnare DUE FILE: *l'intero progetto Eclipse e il JAR eseguibile*

Si ricorda che compiti *non compilabili*, o che *non passino almeno 2/3 dei test* o siano *palesemente lontani da 18/30* NON SARANNO CORRETTI e causeranno la verbalizzazione del giudizio “RESPINTO**”.**

DOMINIO DEL PROBLEMA: Il personale che lavora su progetti finanziati è tenuto a compilare i cosiddetti *timesheet*, ossia dei *rendiconti mensili* nei quali vengono riportate, per ogni giorno del mese, le ore lavorate su ogni progetto; i rendiconti mensili sono poi raggruppati in un unico *rendiconto annuale*. Un esempio di rendiconto mensile cartaceo è illustrato sotto: come si può vedere, per ogni giorno del mese sono riportate le ore lavorate (nel formato HH:MM) su ogni progetto: opportune righe/colonne di sintesi forniscono i subtotali giornalieri (in verticale) e per progetto (in orizzontale), nonché il totale generale delle ore lavorate nel mese (in basso a destra).

Timesheet del mese di MARZO 2024										
	1	2	3		27	28	29	30	31	tot
Progetto A	01:30	00:00	00:00		01:00	02:00	00:00	00:00	00:00	15:30
Progetto B	01:00	00:00	00:00		01:00	00:00	03:30	00:00	00:00	24:00
Progetto C	00:00	00:00	00:00		01:00	01:30	01:30	00:00	00:00	18:30
...
TOTALE	02:30	00:00	00:00		03:00	03:00	03:00	00:00	00:00	88:00

Naturalmente, i giorni riportati sono tutti e soli quelli previsti dallo specifico mese, ossia 30 per i mesi di 30 giorni, 29 o 28 per febbraio (a seconda se l’anno sia bisestile o meno), 31 per gli altri mesi.

Poiché in molte realtà non è consentito al personale di lavorare di sabato e/o di domenica (come nella figura sopra), le ore riportate per quelle giornate, in tal caso, dovranno necessariamente essere zero. Spesso, inoltre, i contratti di lavoro stabiliscono un *numero massimo* di ore giornaliere lavorabili, oltre le quali non si può andare (tipicamente 8): naturalmente, nessuno può indicare in un dato giorno più di tale numero di ore.

OBIETTIVO: È richiesto di sviluppare un’applicazione che sostituisca i rendiconti cartacei, consentendo di inserire le ore lavorate tramite un’opportuna interfaccia grafica. L’app dovrà effettuare automaticamente i controlli sull’eventuale lavoro di sabato e festivo (ossia, dovrà consentirlo solo se le impostazioni dell’applicazione lo permettono) e sulle ore giornaliere lavorate (che non potranno eccedere il massimo prestabilito), nonché fornire la sintesi delle ore totali lavorate (per mese, per progetto, annuali) o il loro dettaglio, come da figure sotto riportate.

Il file di testo [Projects.txt](#) specifica, nel formato descritto più oltre, i progetti attivi con il numero totale di ore previste per ciascuno. L’applicazione dovrà:

- leggere tali file e caricare i dati nelle opportune strutture-dati interne
- permettere all’utente, tramite la GUI, di introdurre via via i dati necessari e avere sempre sott’occhio la situazione, prevenendo eventuali azioni errate (es. inserimento di ore lavorative di domenica, orari errati, etc.) tramite appositi dialoghi.

TEMPO STIMATO PER SVOLGERE L'INTERO COMPITO:**2h15 – 3h**

PARTE 1 – Modello dei dati: Punti 11

[TEMPO STIMATO: 50-70 minuti]

PARTE 2 – Persistenza: Punti 8

[TEMPO STIMATO: 35-40 minuti]

PARTE 3 – Grafica: Punti 11

[TEMPO STIMATO: 50-70 minuti]

NUMERO MINIMO DI TEST CON SUCCESSO PERCHÉ IL COMPITO SIA CORRETTO**40 su 58****JAVAFX – Parametri run configuration nei LAB**

```
--module-path "C:\applicativi\moduli\javafx-sdk-21.0.2\lib"  
--add-modules javafx.controls
```

Cose da ricordare

- salva costantemente il tuo lavoro: l'informatica a volte può essere “subdolamente ostile”..
- in particolare: se ora compila e stai per fare modifiche, salva la versione attuale (non si sa mai)

Checklist di consegna

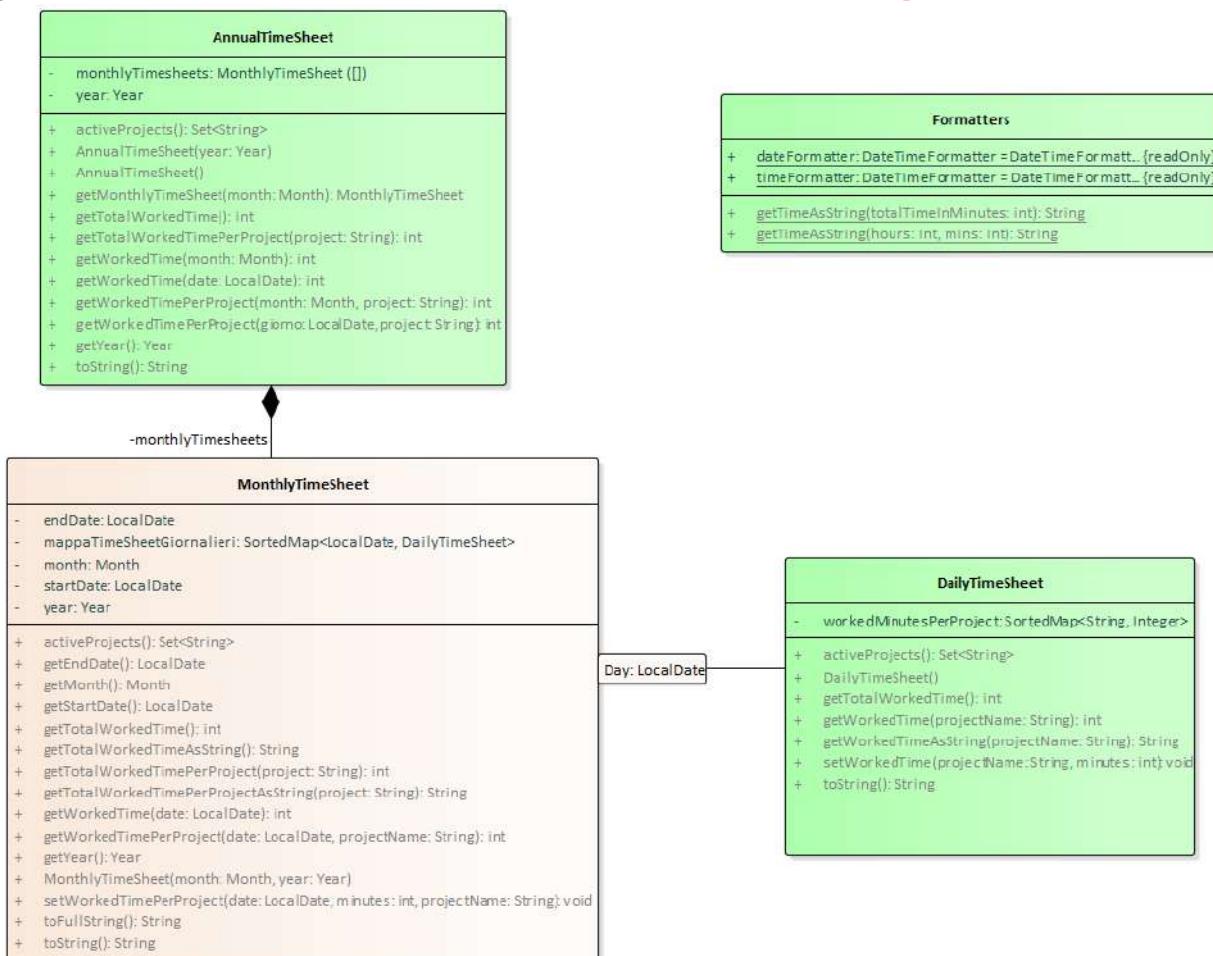
- Hai fatto un **JAR eseguibile**, che contenga cioè l'indicazione del main?
- Hai controllato che **si compili e ci sia tutto**? [NB: non includere il PDF del testo]
- Hai **rinominato IL PROGETTO**, lo ZIP e il JAR esattamente come richiesto?
- Hai **chiamato** la cartella del progetto esattamente come richiesto?
- **Hai fatto un unico file ZIP (NON .7z, rar o altri formati)** contenente **l'intero progetto?**
In particolare, ti sei assicurato di aver incluso tutti i file .java (e non solo i .class)?
- **Hai consegnato DUE file distinti, ossia lo ZIP col progetto e il JAR eseguibile?**
- Su EOL, hai **premuto** il tasto “CONFERMA” per inviare il tuo elaborato?

Parte 1 – Modello dei dati

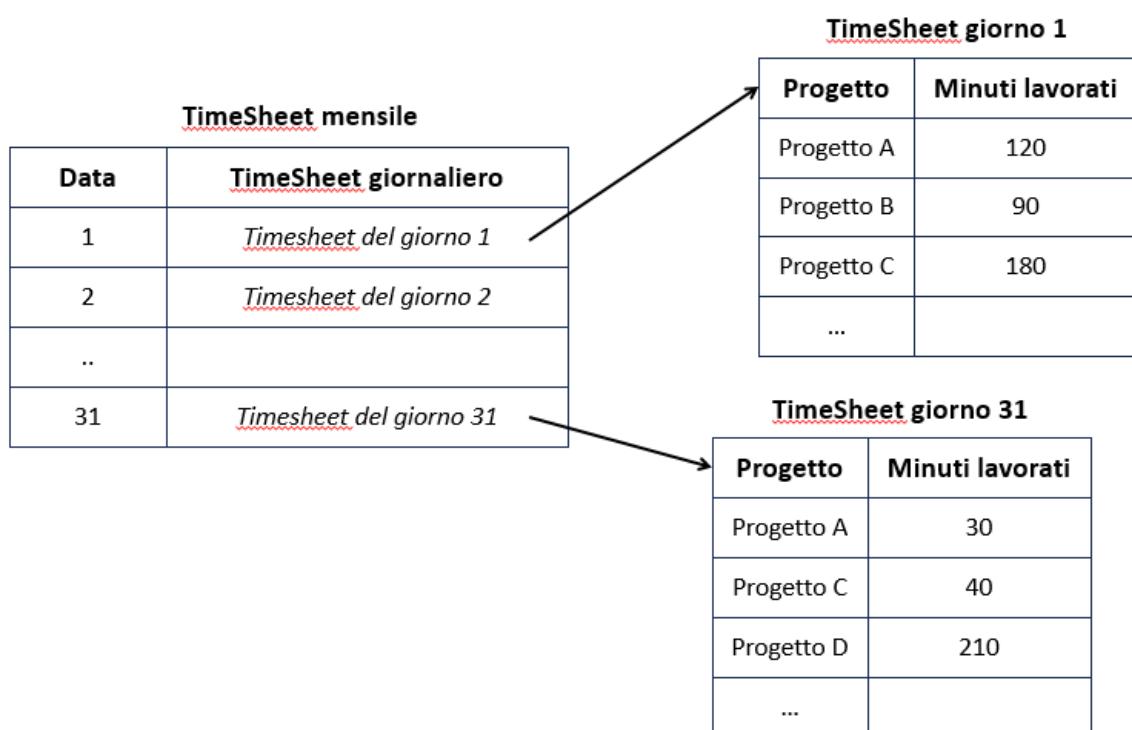
(punti: 11)

Package: timesheet.model

[TEMPO STIMATO: 50-70 minuti]



PREMESSA: poiché la tabella cartacea opera su tre dimensioni (giorni, progetti, ore lavorate) mentre la mappa Java esprime solo due dimensioni (colonne), occorre spezzare la rappresentazione su più strutture collegate. A tal fine, il **timesheet mensile** verrà espresso come mappa (giorni, timesheet giornaliero), in cui il **timesheet giornaliero** è a sua volta una mappa (progetti, minuti lavorati), come mostrato nella figura seguente:



SEMANTICA:

- La classe **Formatters** fornisce alcuni formattatori già configurati per ora e data in standard italiano, nonché due metodi `getTimeAsString` che forniscono una stringa già formattata nel formato HH:MM partendo, rispettivamente, o da un totale in minuti, o da ore e minuti espressi come interi.
- La classe **DailyTimeSheet** rappresenta il resoconto giornaliero, che riporta i minuti lavorati su ogni progetto. I progetti per i quali quel giorno non si è lavorato NON vengono riportati. Il costruttore alloca un resoconto vuoto. Oltre a un'apposita `toString`, sono forniti i seguenti metodi:
 - `setWorkedTime` imposta i minuti lavorati per un dato progetto: il metodo verifica preventivamente che il nome del progetto non sia nullo né blank, o che i minuti non siano negativi, lanciando nel caso, secondo le note linee guida, **NullPointerException** o **IllegalArgumentException**.
 - `getWorkedTime` restituisce i minuti lavorati sul progetto specificato sotto forma di intero, verificando preventivamente la validità dell'argomento come sopra; il metodo gemello `getWorkedTimeAsString` restituisce il tempo lavorato sotto forma di stringa formattata come HH:MM
 - `getTotalWorkedTime` restituisce i minuti lavorati complessivamente quel giorno, in tutti i progetti
 - `activeProjects` restituisce l'insieme dei nomi dei progetti per i quali quel giorno si è lavorato.
- La classe **MonthlyTimeSheet** (da completare) rappresenta il resoconto mensile: a ogni giorno del mese è associata un'opportuna istanza di **DailyTimeSheet**. Il costruttore riceve mese e anno, sotto forma di istanze rispettivamente di **Month** e **Year**: ne verifica la validità e alloca le istanze di **DailyTimeSheet** necessarie.
 - gli accessori `getMonth`, `getYear`, `getStartDate (*)`, `getEndDate (*)` restituiscono rispettivamente mese e anno di riferimento (i due argomenti ricevuti dal costruttore), nonché la data iniziale e finale del mese (ad esempio, per marzo 2024, rispettivamente 01/03/2024 e 31/03/2024), ovviamente tenendo conto della durata in giorni dello specifico mese.
 - `getTotalWorkedTime` restituisce i minuti totali lavorati nel mese, sotto forma di intero; il metodo gemello `getTotalWorkedTimeAsString` restituisce tale tempo lavorato sotto forma di stringa HH:MM
 - `getTotalWorkedTimePerProject` restituisce i minuti totali lavorati nel mese per uno specifico progetto: anche in questo caso, il metodo gemello `getTotalWorkedTimePerProjectAsString` restituisce il tempo lavorato sotto forma di stringa HH:MM.
 - `activeProjects` restituisce l'insieme dei nomi dei progetti per i quali quel mese si è lavorato.
 - Il metodo **getWorkedTime (da fare)** restituisce i minuti lavorati in un dato giorno, espresso sotto forma di **LocalDate**, per tutti i progetti. Più precisamente, il metodo riceve la data richiesta, verifica preventivamente che essa sia non nulla e compresa nel mese (*), lanciando nel caso le opportune eccezioni con opportuni messaggi d'errore: solo in caso positivo accede al corrispondente **DailyTimeSheet** e recupera i minuti totali lavorati in quel giorno. Nel caso in cui non vi sia ancora alcun **DailyTimeSheet** istanziato per quel giorno, deve restituire (ovviamente) 0 minuti lavorati.
 - La coppia di metodi **getWorkedTimePerProject / setWorkedTimePerProject (entrambi da fare)** permette di recuperare / impostare i minuti lavorati in uno dato giorno, espresso sotto forma di **LocalDate**, per un dato progetto. Più precisamente:
 - **getWorkedTimePerProject** riceve la data richiesta e il nome del progetto desiderato, verifica preventivamente che essa sia non nulla e compresa nel mese (*) e che il nome del progetto non sia nullo né blank, lanciando nel caso le opportune eccezioni con opportuni messaggi d'errore: solo in caso positivo accede al corrispondente **DailyTimeSheet** e recupera i minuti lavorati in quel giorno per lo specifico progetto;
 - **setWorkedTimePerProject** riceve la data relativa al giorno richiesto, i minuti lavorati e il nome del progetto a cui si riferiscono: verifica preventivamente che la data sia non nulla e compresa

nel mese (*), che i minuti non siano negativi e che il nome del progetto non sia nullo né blank, lanciando nel caso le opportune eccezioni con opportuno messaggio d'errore; solo se tutto filo liscio, accede al **DailyTimeSheet** del giorno specificato e imposta in esso i minuti lavorati per il progetto specificato.

- `toString` restituisce la sintesi delle ore lavorate nel mese sui diversi progetti, mentre `toFullString` fornisce un livello di dettaglio maggiore, specificando le ore lavorate per ogni singolo giorno.
- La classe **AnnualTimeSheet** rappresenta il resoconto annuale, ottenuto per composizione dei dodici resoconti mensili: a tal fine, mantiene al suo interno un array di **MonthlyTimeSheet**, indicizzato in base al mese. Il costruttore primario riceve l'anno di riferimento e alloca le opportune strutture interne, mentre il costruttore ausiliario senza argomenti assume di default l'anno corrente. Sono forniti i seguenti metodi:
 - `getYear` restituisce semplicemente l'anno a cui il resoconto si riferisce, come istanza di **Year**
 - `getMonthlyTimeSheet` fornisce il riferimento al resoconto mensile interno relativo al mese specificato come argomento (che naturalmente non può essere nullo: altrimenti, **NullPointerException**)
 - `getTotalWorkedTime` restituisce i minuti lavorati complessivamente nell'anno, per tutti i progetti
 - `getTotalWorkedTimePerProject` restituisce i minuti lavorati complessivamente nell'anno per il solo progetto ricevuto come argomento (se nullo, viene lanciata **NullPointerException**)
 - `getWorkedTime(Month)` restituisce i minuti lavorati nel mese specificato, per tutti i progetti, sotto forma di intero; la validità dell'argomento è verificata come sopra
 - `getWorkedTime(LocalDate)` restituisce i minuti lavorati nel giorno specificato, per tutti i progetti, sotto forma di intero; la validità dell'argomento è verificata come sopra
 - `getWorkedTimePerProject(Month, String)` restituisce i minuti lavorati nel mese specificato, per il solo progetto specificato, sotto forma di intero; la validità dell'argomento è verificata come sopra
 - `getWorkedTimePerProject(LocalDate, String)` restituisce i minuti lavorati nel giorno specificato, per il solo progetto specificato, sotto forma di intero; la validità dell'argomento è verificata come sopra
 - `activeProjects` restituisce l'insieme dei nomi dei progetti per i quali si è lavorato in quell'anno
 - `toString` produce il resoconto annuale per composizione dei resoconti mensili.

Parte 2 – Persistenza

Package: `timesheet.persistence`

(punti: 8)

[TEMPO STIMATO: 35-40 minuti]

Il file di testo `Projects.txt` elenca i progetti attivi con il numero totale di ore previste per ciascuno. Ogni riga contiene:

- il nome del progetto (che può contenere spazi), seguito da una o più tabulazioni
- la frase “`ore previste:`”, seguita da uno o più spazi
- un numero intero, che rappresenta il totale di ore previste complessivamente sul progetto

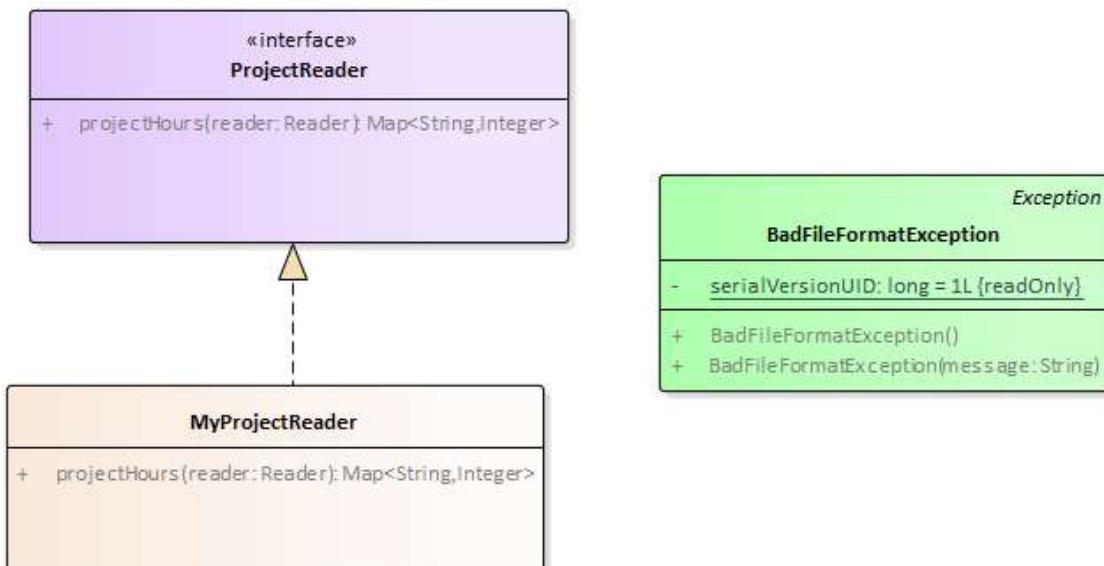
Lezioni di Fondamenti T2	ore previste: 80
Lezioni di Linguaggi	ore previste: 64
Progetto Peonie	ore previste: 350
Progetto Innova	ore previste: 700
...	

SEMANTICA:

- L'interfaccia **ProjectReader** (fornita) dichiara il metodo `projectHours`, che legge dal **Reader** fornito (già aperto) i dati e restituisce una `Map<String, Integer>` che associa a ogni progetto il rispettivo numero di ore; lancia **BadFormatException** con opportuno messaggio d'errore in caso di problemi nel formato del file, o **IOException** in caso di altri problemi di I/O.

b) La classe **MyProjectReader** (da realizzare) implementa **ProjectReader**:

- non c'è alcun costruttore
- Il metodo **projectHours** (da implementare) effettua le letture: lancia **NullPointerException** in caso di reader nullo, o **BadFormatException** con dettagliato messaggio d'errore in caso di mancato rispetto del formato previsto, catturando eventuali altre eccezioni interne. In particolare deve verificare:
 - che gli elementi in ogni riga siano esattamente tre
 - che il secondo elemento sia la frase prevista
 - che l'ultimo elemento sia un intero strettamente positivo



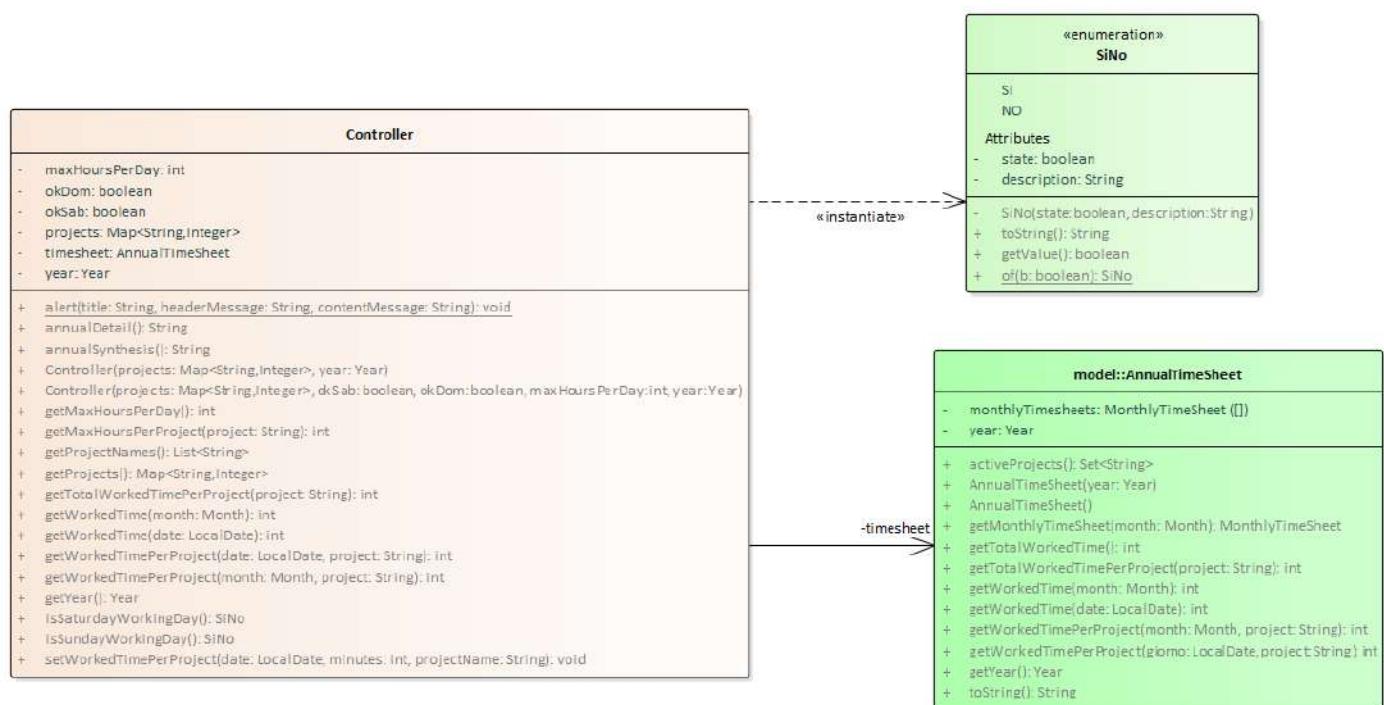
Parte 3

(punti: 11)

Package: timesheet.controller

[TEMPO STIMATO: 10-15 minuti] (punti 3)

Il controller costituisce il nucleo centrale di questa applicazione: crea e mantiene le strutture dati, gestisce la logica di gioco e le modalità dello stesso. A tal fine è organizzato come segue:



SEMANTICA:

- a) L'enumerativo ***SiNo*** esprime le due costanti **SI** e **NO** che encapsulano rispettivamente i booleani *true* e *false*, accoppiando però a ciascuno la descrizione testuale italiana, restituita poi da *toString*; l'accessor *getValue* consente di recuperare il valore booleano associato alle due costanti enumerative, mentre il metodo factory *of* consente, dualmente, di ottenere l'istanza di ***SiNo*** corrispondente al boolean ricevuto come argomento.
- b) La classe ***Controller (da completare nei costruttori)*** governa il funzionamento dell'applicazione: mantiene al suo interno sia la mappa dei progetti ricevuta dal costruttore, sia l'***AnnualTimeSheet*** che costituisce lo stato dell'applicazione stessa. Più precisamente:
- Il **costruttore primario (da fare)** riceve la **Map<String, Integer>** restituita dalla persistenza, due booleani che esprimono l'eventuale possibilità di lavorare anche il sabato o la domenica, il numero massimo di ore lavorabili al giorno e l'anno di lavoro: dopo aver validato gli argomenti, memorizza il riferimento alla struttura dati ricevuta e crea l'***AnnualTimeSheet*** per l'anno specificato, su cui opererà da ora in poi.
 - Il **costruttore ausiliario (da fare)** a due soli argomenti configura il controller nel caso più standard (lavoro permesso solo da lunedì a venerdì, max 8 ore lavorative al giorno) per l'anno specificato.
 - Opportuni metodi delegano le operazioni alle corrispondenti classi del model. Più precisamente:
 - *getProjects* restituisce semplicemente la **Map<String, Integer>** ricevuta dal costruttore
 - *getProjectNames* restituisce i nomi dei progetti elencati in tale mappa (una lista di stringhe)
 - *isSaturdayWorkingDay / isSundayWorkingDay* verificano se rispettivamente sabato / domenica siano giorni lavorativi, restituendo il risultato sotto forma di istanza dell'enumerativo ***SiNo***
 - *getYear* restituisce l'anno specificato al costruttore, a cui si riferiscono le ore di lavoro
 - *getMaxHoursPerDay* restituisce il numero massimo di ore lavorative giornaliere
 - *getMaxHoursPerProject* restituisce il numero di ore previste per un progetto, sotto forma di intero
 - *getTotalWorkedTimePerProject* restituisce il numero totale di minuti lavorati per un dato progetto
 - *getWorkedTime(Month)* restituisce i minuti lavorati nel mese specificato, per tutti i progetti
 - *getWorkedTime(LocalDate)* restituisce i minuti lavorati nel giorno specificato, per tutti i progetti
 - *getWorkedTimePerProject(Month, String)* restituisce i minuti lavorati nel mese specificato, per il solo progetto specificato
 - *getWorkedTimePerProject(LocalDate, String)* restituisce i minuti lavorati nel giorno specificato, per il solo progetto specificato
 - *setWorkedTimePerProject(LocalDate, int, String)* imposta i minuti lavorati nel giorno specificato, per il progetto specificato: verifica che i minuti non siano negativi, che gli argomenti non siano nulli e che il nome del progetto non sia vuoto o blank
 - *annualDetail* restituisce una stringa che fornisce il resoconto dettagliato dell'anno
 - *annualSynthesis* restituisce invece una stringa che fornisce il resoconto sintetico dell'anno

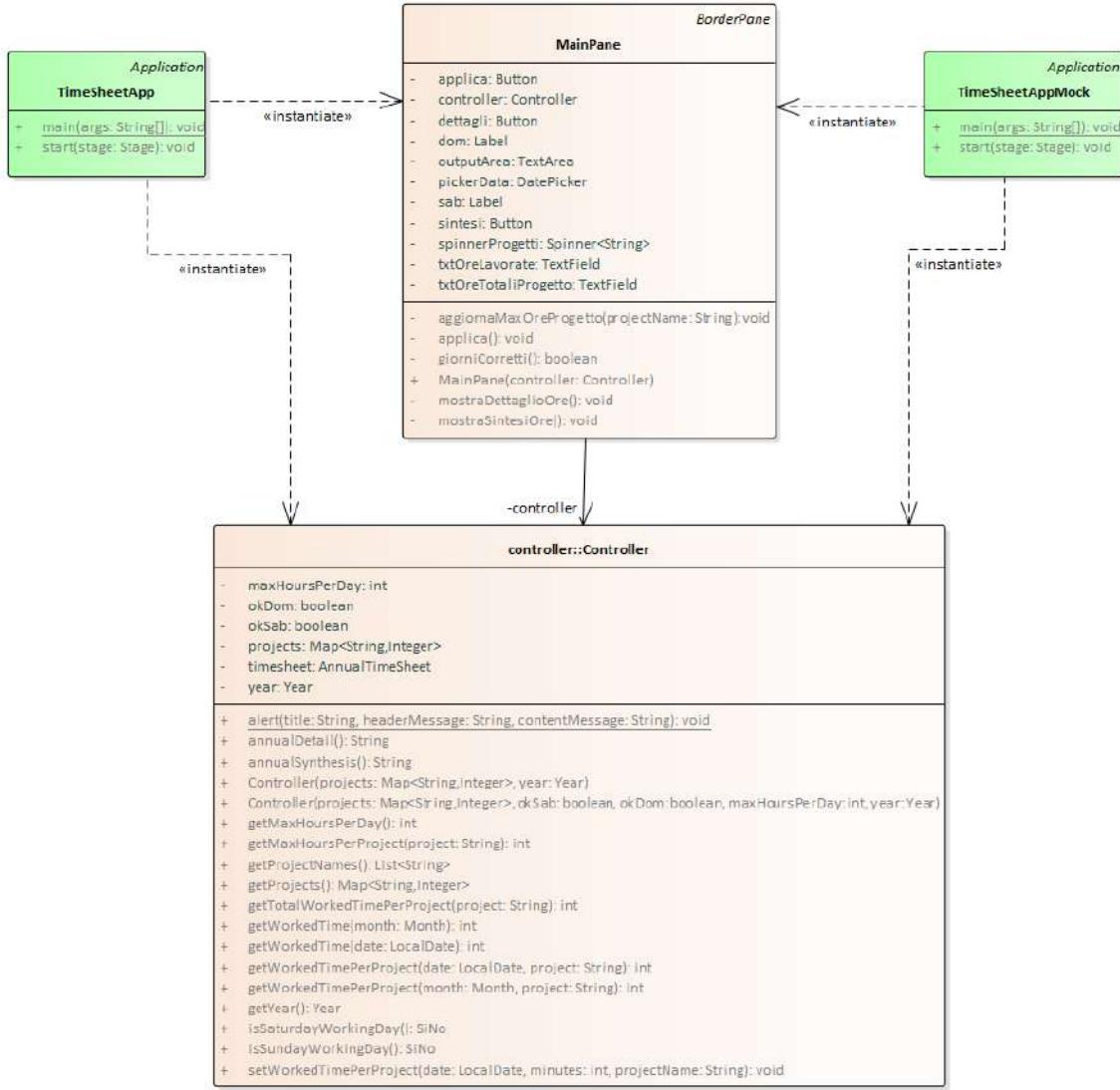
Il metodo statico *alert*, utilizzabile dal ***MainPane*** quando è attiva la grafica, consente di far comparire all'utente una finestra di dialogo che mostri il messaggio d'errore specificato.

Package: timesheet.ui

[TEMPO STIMATO: 40-55 minuti] (punti 8)

La classe ***TimeSheetApp*** (fornita) costituisce l'applicazione JavaFX che si occupa di aprire i file, creare il controller e incorporare il ***MainPane***. Per consentire di collaudare la GUI anche in assenza / in caso di malfunzionamento della parte di persistenza, è possibile avviare l'applicazione mediante la classe ***TimeSheetAppMock***.

L'interfaccia utente è illustrata nelle figure seguenti e segue il modello sotto illustrato:



- In alto sono riportate indicazioni statiche relative alla possibilità di lavorare il sabato e la domenica, e al numero massimo di ore giornaliere permesse.
- Al centro vi sono tutti i controlli per la scelta del progetto (uno **Spinner** di stringhe) e della data (un **DatePicker**), l'inserimento delle ore lavorate (un **TextField**) e i vari pulsanti (**Applica**, **Dettaglio ore**, **Sintesi ore**) che attivano le varie funzionalità dell'applicazione.
- In basso vi è l'area di output, che mostra i messaggi destinati all'utente.

The figure displays two side-by-side screenshots of the 'Compilazione TimeSheet' application. Both screens show a header with status information: 'Lavoro sabato: No', 'Lavoro domenica: No', and 'Max ore giornaliere: 8'. Below the header, there is a form with fields for 'Progetto' (containing a spinner with 'Lezioni di Fondamenti T2'), 'Ore previste' (set to 80), 'Data' (set to 28/12/2024), and 'Ore lavorate' (set to 00:00). At the bottom of the form are three buttons: 'Applica', 'Dettaglio ore', and 'Sintesi ore'. In the second screenshot on the right, the 'Progetto' field has been changed to 'Progetto Innova', and the 'Ore previste' value has been updated to 700, while the other fields remain the same.

Fig. 1: situazione iniziale: notare le ore previste, adeguate al progetto mostrato nello spinner a sinistra.

Operando sulle frecce dello spinner si può cambiare progetto: le ore indicate a lato cambiano di conseguenza.

Agendo sul calendario si può scegliere la data di proprio interesse (Fig 2, sinistra) e inserire le ore lavorate premendo il pulsante **Applica**: l'applicazione risponderà mostrando un messaggio di conferma, Nel caso il giorno scelto sia sabato o domenica, se l'impostazione non lo permette verrà mostrato un messaggio d'errore (Fig 2, destra) e non verrà effettuata alcuna operazione.

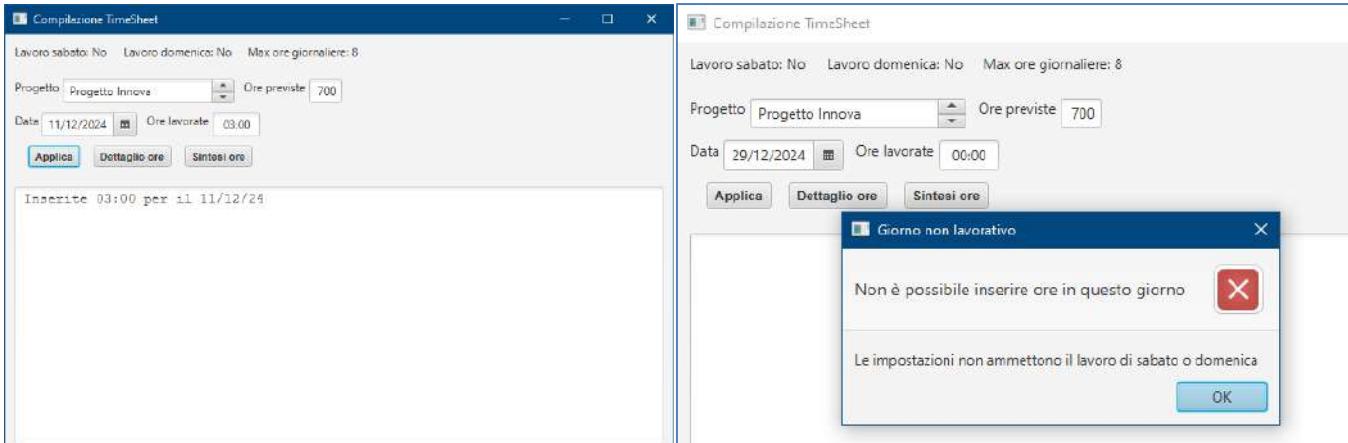


Fig. 2: selezione data e inserimento ore lavorate.

Procedendo a scegliere date e inserire ore per i vari progetti, si può via via compilare tutto il rendiconto. In ogni momento, il pulsante **Dettaglio ore** mostra il dettaglio delle ore lavorate in ogni singolo giorno per ogni progetto (Fig.3), mentre il pulsante **Sintesi ore** fornisce la visione d'insieme (Fig. 4).

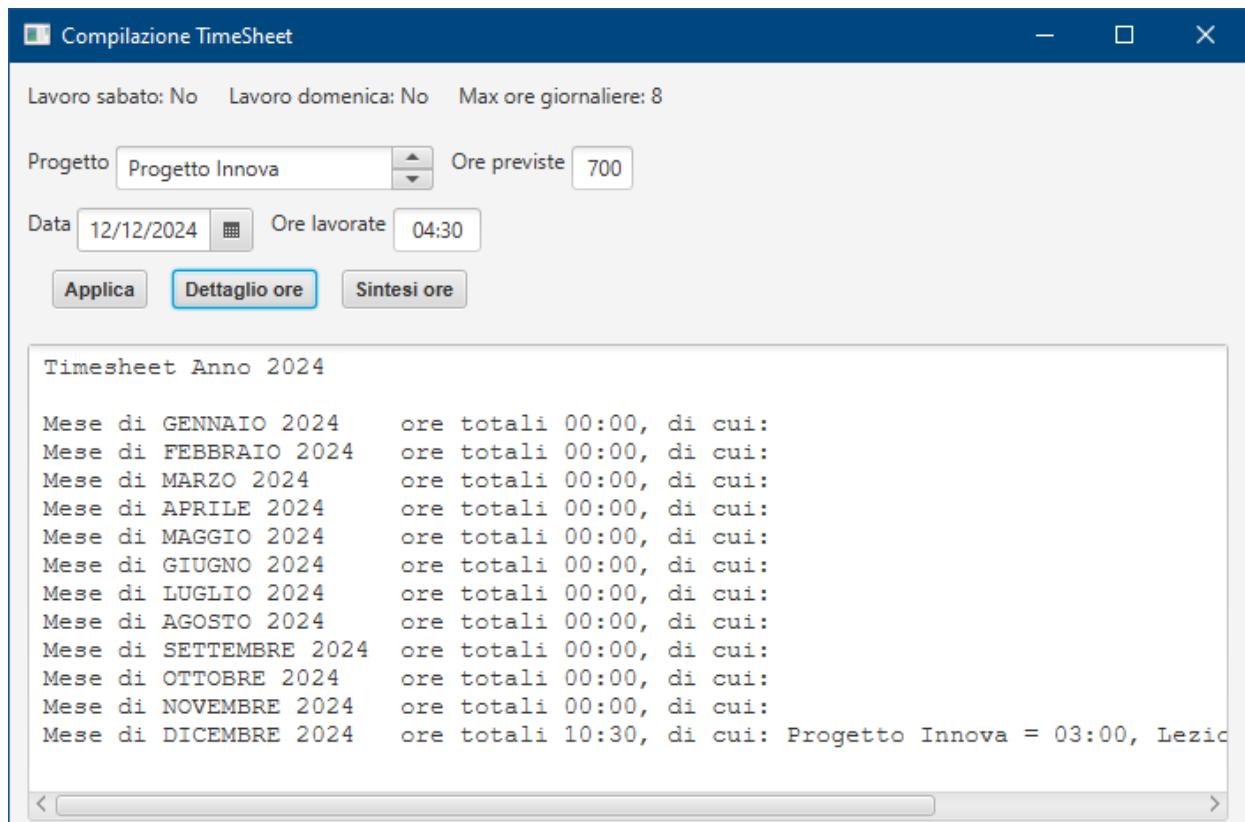


Fig. 3: dettaglio ore lavorate: per ogni mese vengono indicate le ore totali e la ripartizione fra i singoli progetti

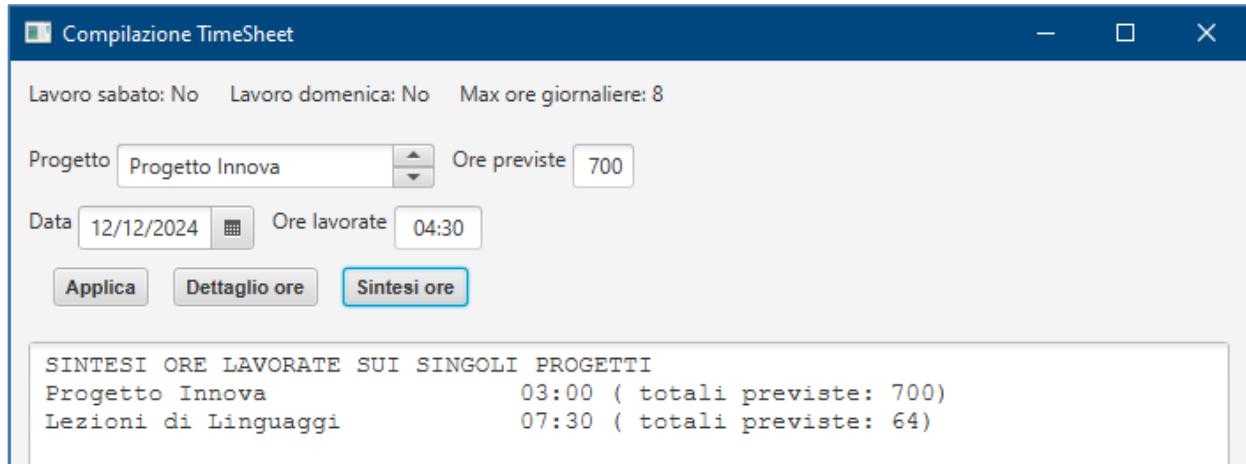
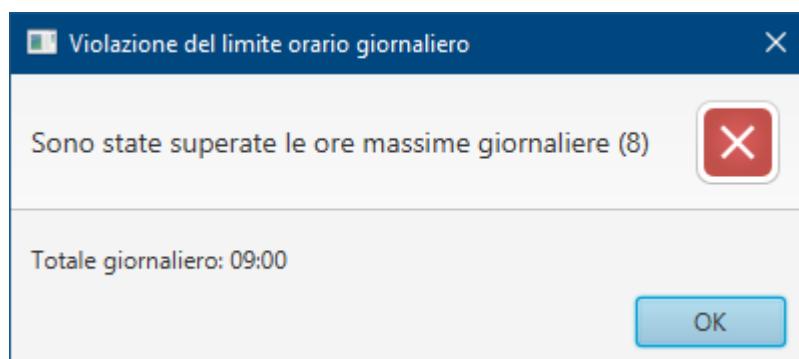


Fig. 4: sintesi ore lavorate: vengono elencate le ore complessivamente svolte in ogni progetto, con a fianco quelle totali previste per il progetto stesso.

Nel caso in cui si tenti di inserire, in un dato giorno, più ore di quelle massime permesse, viene emesso un messaggio d'errore (Fig. 5)



Il MainPane è fornito quasi completamente realizzato: è presente tutta la parte strutturale, mentre rimangono da realizzare due gestori degli eventi (metodi privati *applica* e *giorniCorretti*).

In particolare:

- *giorniCorretti* viene invocato all'atto della selezione di una data sul datepicker e deve verificare se il giorno selezionato sia lavorativo, in base alle impostazioni correnti dell'applicazione; più precisamente:
 - i giorni da lunedì a venerdì sono sempre lavorativi
 - il sabato e la domenica lo sono solo se ciò è permesso dall'impostazione iniziale del costruttore
 Nel caso il giorno selezionato risulti non lavorativo, il metodo deve emettere un messaggio d'errore tramite il metodo statico *alert* del Controller e restituire *false*; in caso invece di esito positivo, deve semplicemente restituire *true*.
- *applica* è associato alla gestione del pulsante omonimo e deve gestire tutto l'inserimento delle ore con i necessari controlli. In particolare, deve:
 - recuperare le ore lavorate, controllarne il formato e, se tutto risulta corretto, verificare tramite *giorniCorretti* che la data selezionata sia un giorno lavorativo valido.
 - solo in caso positivo, deve verificare che, aggiungendo le ore attuali, non si superi il massimo numero di ore lavorative giornaliere [NB: nel ricalcolo si presti attenzione a scorporare le ore lavorate già inserite in precedenza: ad esempio, se in precedenza si fossero inserite per quel giorno 5 ore, deve essere possibile cambiarle con 7, evitando che la semplice somma 5+7 = 12 dia la falsa impressione di aver superato il massimo giornaliero, es. di 8 ore]

- o solo se anche questa verifica è positiva, recuperare dallo **Spinner** il nome del progetto e impostare le ore lavorate tramite il metodo `setWorkedTimePerProject` del **Controller**, emettendo nell'area di testo un apposito messaggio di conferma (vedere Fig. 2 sopra).
- o In tutti i casi di errore sopra evidenziati, il metodo dovrà emettere tramite `alert` un apposito, dettagliato messaggio d'errore all'utente.

Cose da ricordare

- salva costantemente il tuo lavoro: l'informatica a volte può essere "subdolamente ostile"...
- in particolare: se ora compila e stai per fare modifiche, salva la versione attuale (non si sa mai)

Checklist di consegna

- Hai fatto un **JAR eseguibile**, che contenga cioè l'indicazione del main?
- Hai controllato che **si compili e ci sia tutto**? [NB: non includere il PDF del testo]
- Hai **rinominato IL PROGETTO**, lo ZIP e il JAR esattamente come richiesto?
- Hai **chiamato** la cartella del progetto esattamente come richiesto?
- **Hai fatto un unico file ZIP (NON .7z, rar o altri formati) contenente l'intero progetto?**
In particolare, ti sei assicurato di aver incluso tutti i file .java (e non solo i .class)?
- **Hai consegnato DUE file distinti, ossia lo ZIP col progetto e il JAR eseguibile?**
- Su EOL, hai **premuto** il tasto "CONFERMA" per inviare il tuo elaborato?

ESAME DI FONDAMENTI DI INFORMATICA T-2 del 15/1/2025

Proff. E. Denti – R. Calegari – A. Molesini

Tempo a disposizione: 3h30

NOME PROGETTO ECLIPSE: CognomeNome-matricola (es. RossiMario-0000123456)

NOME CARTELLA PROGETTO: CognomeNome-matricola (es. RossiMario-0000123456)

NOME ZIP DA CONSEGNARE: CognomeNome-matricola.zip (es. RossiMario-0000123456.zip)

NOME JAR DA CONSEGNARE: CognomeNome-matricola.jar (es. RossiMario-0000123456.jar)

Si devono consegnare DUE FILE: *l'intero progetto Eclipse e il JAR eseguibile*

Si ricorda che compiti *non compilabili*, o che *non passino almeno 2/3 dei test* o siano *palesemente lontani da 18/30* NON SARANNO CORRETTI e causeranno la verbalizzazione del giudizio “RESPINTO**”.**

È richiesto di sviluppare un'applicazione per il preventivo del costo di un noleggio auto. L'applicazione deve innanzitutto proporre all'utente le città in cui è possibile noleggiare l'auto, indi – dopo che la città sarà stata scelta – popolare un'apposita combo con le agenzie disponibili in tale città. Dovrà poi consentire all'utente di inserire giorno e ora di ritiro e di riconsegna dell'auto, il tipo di auto e di specificare se desideri lasciare l'auto in una città diversa (non specificata). Acquisiti tali dati, dovrà innanzitutto verificare che l'agenzia di noleggio prescelta sia aperta nel giorno e ora indicati come inizio del noleggio, e se sì calcolare e mostrare a video il preventivo corrispondente.

DESCRIZIONE DEL DOMINIO DEL PROBLEMA

In ogni **città** possono essere presenti più **agenzie**, ognuna identificata da una descrizione (nome) univoca: ogni agenzia è caratterizzata dai suoi **orari di apertura**, distinti per giorni infrasettimanali (da lunedì a venerdì), sabato, domenica.

Un orario di apertura può avere tre forme:

- La costante CHIUSO
- Uno **slot orario**, della forma HH:MM-HH:MM (orario minimo: 00:00; orario massimo: 23:59)
- Due slot orari separati da “/”, secondo il pattern HH:MM-HH:MM/HH:MM-HH:MM, per esprimere orari di apertura non continuati, con pausa intermedia (tipicamente, mattino/pomeriggio).

Il tipo di auto può essere A (mini), B (piccole), C (medie), D (lusso).

Il sistema tariffario prevede, per ogni tipo di auto:

- Una **tariffa giornaliera**, da applicarsi ogni 24h di noleggio (es. dalle 15 di un giorno alle 14:59 del giorno successivo)
- Una **tariffa speciale weekend**, omnicomprensiva, per noleggi compresi fra venerdì e domenica (inclusi)
- Una **sovrottassa fissa**, detta di “*drop off*”, per noleggi con *riconsegna in una città diversa* da quella di ritiro.

Per calcolare il costo del noleggio l'applicazione deve innanzitutto calcolare la durata in giorni del noleggio: eventuali frazioni comportano l'addebito di un'intera giornata ulteriore (ad esempio, un noleggio di 1 giorno e 1 minuto deve essere tariffato come 2 giorni). L'importo base si ottiene quindi moltiplicando la durata in giorni per la tariffa giornaliera. Nel caso in cui il noleggio ricada interamente in un weekend (inizio e termine fra venerdì e domenica dello stesso fine settimana), si potrà applicare la tariffa speciale prevista, se più conveniente rispetto alla tariffa standard. Al costo così calcolato dovrà infine essere aggiunta la sovrottassa di drop-off, nel caso di riconsegna in altra città.

Il file di testo **agencies.txt** contiene, nel formato descritto più oltre, l'elenco delle agenzie disponibili. L'applicazione dovrà:

- a) leggere tale file e caricare i dati nelle opportune strutture-dati interne
- b) permettere all'utente, tramite la GUI, di introdurre tutti i dati necessari per il preventivo e mostrare il risultato del calcolo, specificando anche in dettaglio la durata in giorni, se sia stata applicata la tariffa weekend e se sia stata aggiunta la sovrottassa di drop-off; in caso di azioni errate, la GUI dovrà avvisare l'utente tramite appositi dialoghi.

TEMPO STIMATO PER SVOLGERE L'INTERO COMPITO: **2h15 – 3h**

PARTE 1 – Modello dei dati: Punti 5	[TEMPO STIMATO: 20-30 minuti]
PARTE 2 – Persistenza: Punti 10	[TEMPO STIMATO: 40-60 minuti]
PARTE 3 – Grafica: Punti 15	[TEMPO STIMATO: 75-90 minuti]

NUMERO MINIMO DI TEST CON SUCCESSO PERCHÉ IL COMPITO SIA CORRETTO

Considerando solo OpeningTime, i due reader e il controller	40 su 56
Considerandoli tutti:	56 su 83

JAVAFX – Parametri run configuration nei LAB

```
--module-path "C:\applicativi\moduli\javafx-sdk-21.0.2\lib"  
--add-modules javafx.controls
```

Cose da ricordare

- salva costantemente il tuo lavoro: l'informatica a volte può essere “subdolamente ostile”..
- in particolare: se ora compila e stai per fare modifiche, salva la versione attuale (non si sa mai)

Checklist di consegna

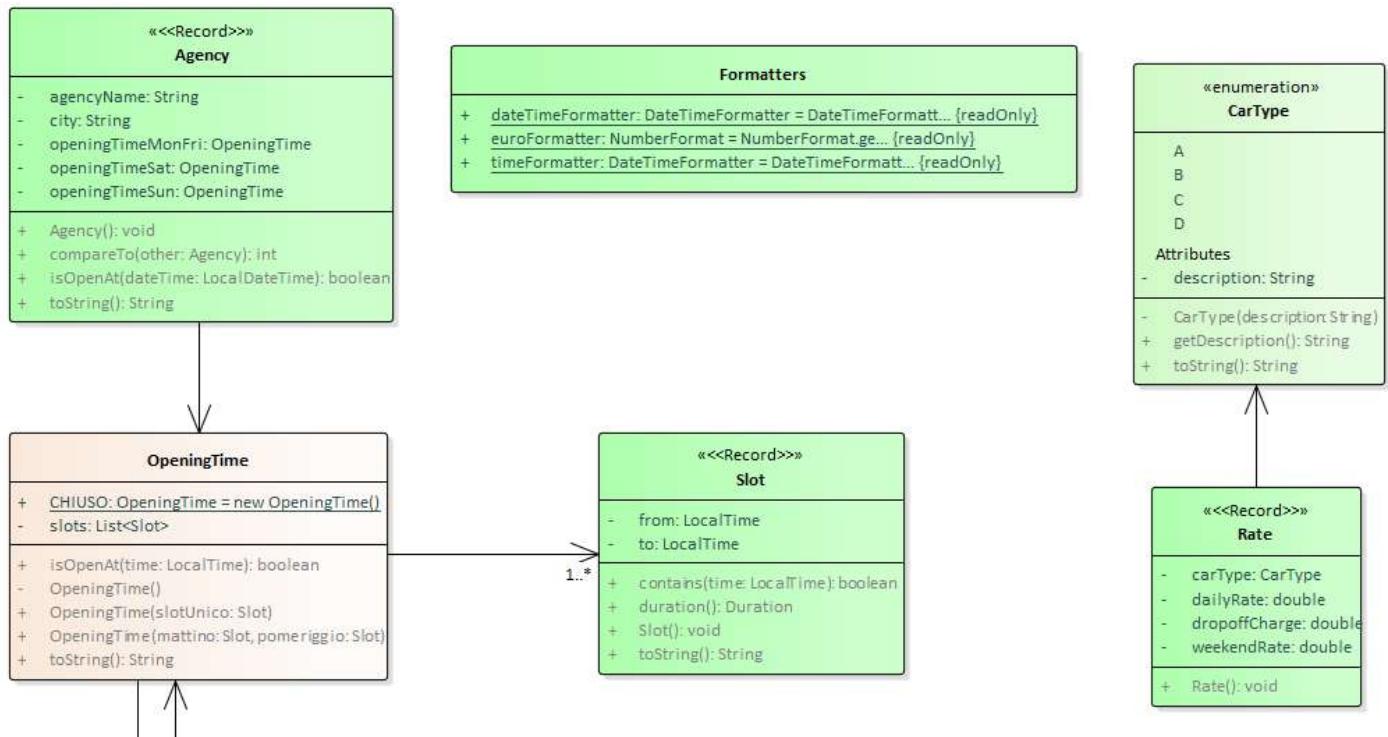
- Hai fatto un **JAR eseguibile**, che contenga cioè l'indicazione del main?
- Hai controllato che **si compili e ci sia tutto**? [NB: non includere il PDF del testo]
- Hai **rinominato IL PROGETTO**, lo ZIP e il JAR esattamente come richiesto?
- Hai **chiamato** la cartella del progetto esattamente come richiesto?
- **Hai fatto un unico file ZIP (NON .7z, rar o altri formati)** contenente **l'intero progetto?**
In particolare, ti sei assicurato di aver incluso tutti i file .java (e non solo i .class)?
- **Hai consegnato DUE file distinti, ossia lo ZIP col progetto e il JAR eseguibile?**
- Su EOL, hai premuto il tasto “CONFERMA” per inviare il tuo elaborato?

Parte 1 – Modello dei dati

Package: autonoleggio.model

(punti: 5)

[TEMPO STIMATO: 20-30 minuti]



SEMANTICA:

- L'enumerativo **CarType** definisce semplicemente i quattro tipi di auto, con i relativi metodi accessori
- La classe **Formatters** contiene alcuni formattatori per valuta (euro), ora, data e ora in standard italiano
- Il record **Slot** specifica una fascia oraria di apertura, da un certo orario a un altro. Il costruttore effettua i necessari controlli sugli argomenti in ingresso. Oltre a un'apposita **toString**, sono forniti i seguenti metodi:
 - **contains** verifica se l'orario fornito come argomento sia contenuto nell'intervallo specificato dallo slot
 - **duration** restituisce la durata dell'intervallo stesso
- Il record **Agency** specifica i dati di un'agenzia di autonoleggio (nome città, nome univoco agenzia, orari di apertura rispettivamente da lunedì a venerdì, il sabato, la domenica): il costruttore effettua accurati controlli sugli argomenti in ingresso, mentre un'apposita **compareTo** garantisce la confrontabilità (e quindi l'ordinamento) per città e in subordine per nome agenzia e un'adeguata **toString** emette una rappresentazione sintetica dell'agenzia stessa (città + nome agenzia). Il metodo **isOpenAt** consente di verificare se l'agenzia sia aperta nel giorno e ora indicato, fornito come istanza di **LocalDateTime**.
- Il record **Rate** specifica i dati di una tariffa (tipo di auto, tariffa giornaliera, tariffa weekend, costo di drop-off): il costruttore effettua i necessari controlli sugli argomenti in ingresso.
- La classe **OpeningTime (da completare)** rappresenta l'orario di apertura di un'agenzia. Come specificato nel Dominio del Problema, esso può assumere tre forme:
 - la costante pubblica statica **CHIUSO**
 - un unico slot orario (es. per esprimere agenzie aperte con orario continuato, o solo mattina/solo pom.)
 - due slot orari (es. per esprimere agenzie aperte mattina e pomeriggio, con pausa intermedia): in questo caso, il primo slot deve precedere interamente il secondo.

La costante CHIUSO è costruita da un costruttore privato (fornito), mentre **gli altri due costruttori (da fare)** devono gestire il caso del singolo slot e dei due slot, rispettivamente. Entrambi devono verificare che gli

argomenti forniti non siano nulli, emettendo nel caso **NullPointerException** con apposito messaggio; il costruttore a due argomenti deve anche verificare che il secondo slot sia successivo al primo.

- Il metodo **isOpenAt (da realizzare)** deve verificare se l'agenzia sia aperta all'orario fornito come argomento, distinguendo e gestendo opportunamente i tre casi
- Un'apposita **toString (da realizzare)** deve emettere la rappresentazione stampabile adeguata, costituita o dalla stringa "chiuso" o da uno o più slot, eventualmente separati da '/'.

Parte 2 – Persistenza

(punti: 10)

Package: autonoleggio.persistence

[TEMPO STIMATO: 40-60 minuti]

Il file di testo `agencies.txt` contiene l'elenco delle agenzie disponibili, nel formato sotto descritto e illustrato negli esempi in calce. Esso specifica un'agenzia per ogni riga, tramite una serie di campi separati da virgole. Nell'ordine sono riportati: città, descrizione agenzia, orario di apertura per i giorni infrasettimanali da lunedì a venerdì, orario di apertura del sabato, orario di apertura della domenica. Gli orari di apertura hanno la seguente forma:

- una delle tre stringhe "lun-ven", "sab", "dom", seguita da uno o più spazi
- poi, o la parola "chiuso"
- oppure un singolo slot orario, della forma HH:MM-HH:MM, senza spazi intermedi
- oppure due slot orari, ciascuno della forma HH:MM-HH:MM, separati da "/", senza spazi intermedi

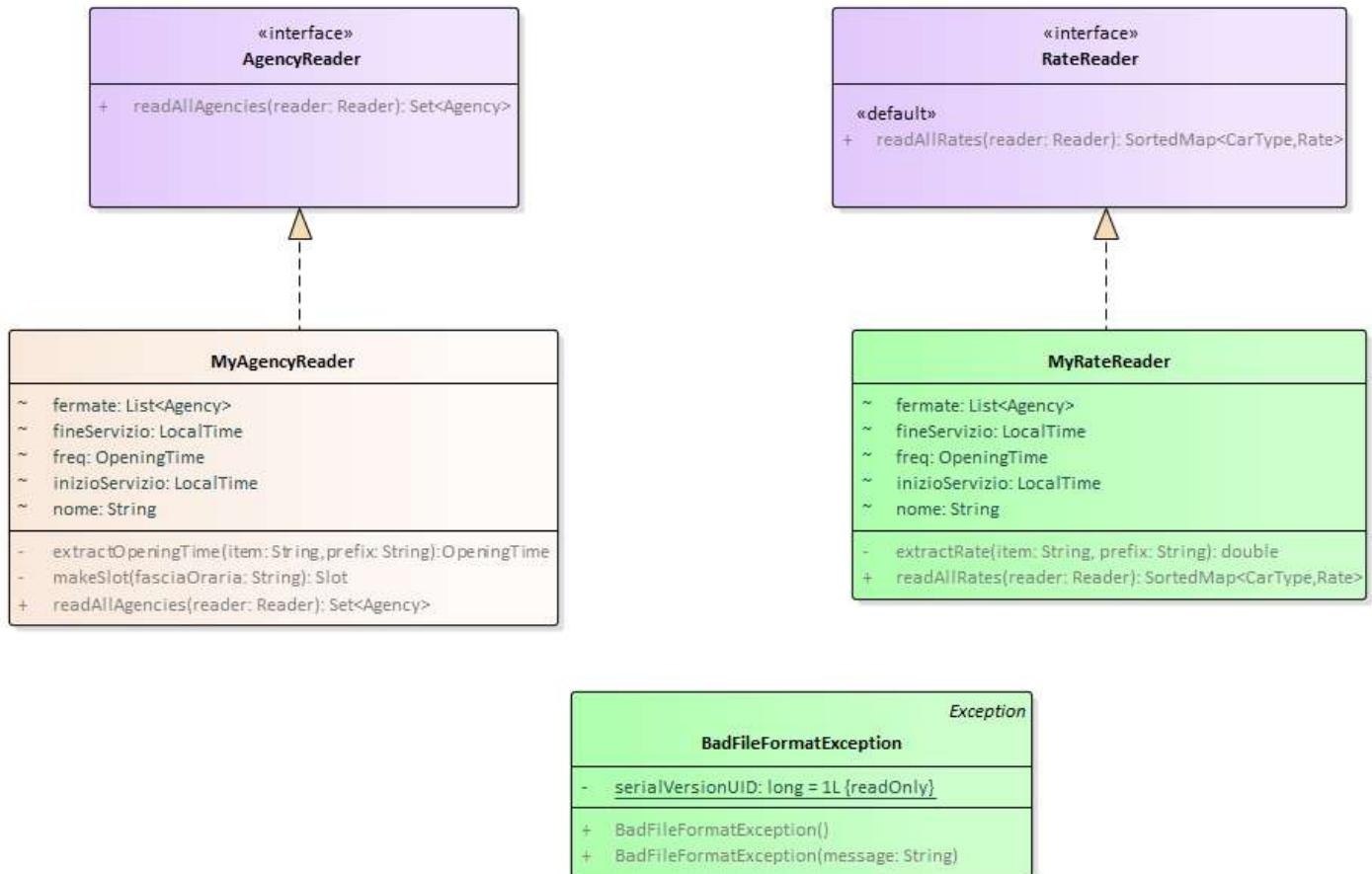
```
BOLOGNA, Aeroporto Marconi, lun-ven 08:00-23:59, sab 08:00-23:59, dom 08:00-23:59
BOLOGNA, Stazione centrale, lun-ven 09:00-13:00/14:00-18:00, sab chiuso, dom chiuso
MODENA, centro, lun-ven 08:30-12:00/15:00-18:00, sab chiuso, dom chiuso
...
```

SEMANTICA:

- a) L'interfaccia **AgencyReader** (fornita) dichiara il metodo `readAllAgencies`, che legge dal **Reader** fornito (già aperto) i dati e restituisce un `Set<Agency>` contenente tanti oggetti **Agency** quante le righe lette dal file; lancia **NullPointerException** nel caso di reader nullo, **BadFormatException** con opportuno messaggio d'errore in caso di problemi nel formato del file, o **IOException** in caso di altri problemi di I/O.
- b) La classe **MyAgencyReader (da implementare)** implementa **AgencyReader**:
 - non c'è alcun costruttore
 - Il metodo **readAllAgencies (da realizzare)** deve fare uso del metodo privato `extractOpeningTime` per estrarre dalla sottostringa, formattato come sopra descritto, un orario di apertura. Nel caso di reader nullo dev'essere lanciata apposita **NullPointerException** con adeguato messaggio d'errore.
 - il metodo ausiliario **extractOpeningTime (da realizzare)** riceve la sottostringa (ad esempio, "lun-ven 08:00-23:59", "dom 08:00-23:59", "sab chiuso", "lun-ven 09:00-13:00/14:00-18:00", etc.) e il prefisso da considerare (uno dei tre "lun-ven", "sab", "dom") e restituisce l'oggetto **OpeningTime** corrispondente, perfettamente configurato, catturando eventuali **IllegalArgumentException** da rilanciare esternamente come **BadFormatException**. Si avvale a sua volta del metodo privato ausiliario `makeSlot`.
 - il metodo ausiliario **makeSlot (da realizzare)** riceve la sottostringa che rappresenta una fascia oraria nel formato HH:MM-HH:MM e restituisce l'oggetto **Slot** corrispondente, adeguatamente configurato; verifica che la stringa sia correttamente formattata, catturando eventuali **DateTimeParseException** e/o **IllegalArgumentException**, da rilanciare esternamente come **BadFormatException**.

Le tariffe previste per i diversi tipi di auto sono elencate nel file `rates.txt`, che tuttavia non deve essere letto perché, per semplicità, i relativi dati di default sono già incapsulati nel codice dell'interfaccia **RateReader** seguente. Esso è quindi incluso nello start kit solo per completezza, per lo studente che voglia esercitarsi in futuro.

- c) L'interfaccia **RateReader** (fornita) dichiara il metodo `readAllRates`, che legge dal **Reader** fornito (già aperto) i dati e restituisce una `SortedMap<CarType, Rate>` contenente tante **Rate** quante le righe lette dal file, ossia una per ogni tipo di auto; lancia **BadFormatException** in caso di problemi nel formato del file, o **IOException** in caso di altri problemi di I/O. **Questa interfaccia contiene già un'implementazione di default (fasulla) di tale metodo, quindi NON deve essere implementata**: è usata dall'**AutoNoleggioApp** così com'è. [Lo studente che in futuro desideri esercitarsi potrà implementare la classe **MyRateReader**, secondo l'usuale schema, modificando poi l'app principale affinché usi tale classe anziché l'implementazione di default fornita in **RateReader**]



PARTE 3: SEGUE NELLA PAGINA SUCCESSIVA → → →

Parte 3

(punti: 15)

Package: autonoleggio.controller

[TEMPO STIMATO: 25-35 minuti] (punti 6)

Il controller costituisce il nucleo centrale di questa applicazione: crea e mantiene le strutture dati, e gestisce la logica di funzionamento. A tal fine è organizzato come segue:



SEMANTICA:

- Il record **Result** (fornito) specifica la coppia (importo, messaggio), utile per accoppiare il risultato del calcolo a un messaggio esplicativo che spieghi come tale risultato è stato ottenuto.
- La classe **Controller (da completare)** implementa già le funzionalità di base: rimane da realizzare solo il metodo **calcolaTariffa**, che deve implementare la logica descritta nel Dominio del Problema. Più precisamente:
 - Il costruttore riceve il **Set<Agency>** e la **SortedMap<CarType,Rate>** restituite dalla persistenza: effettua i necessari controlli e memorizza internamente i riferimenti alle strutture dati ricevute.
 - Oppurtuni metodi consentono di recuperare l'insieme delle agenzie (**getAgencies**), la mappa delle tariffe (**getRates**), la lista delle città (**getCities**) e la lista delle agenzie di una data città (**getAgencies(String city)**)
 - Il metodo **calcolaTariffa (da realizzare)** riceve la città (una stringa), l'oggetto **Agency** che rappresenta l'agenzia, due **LocalDateTime** che rappresentano ora e giorno rispettivamente di inizio e di fine noleggio, un **CarType** che rappresenta il tipo di veicolo, e un **boolean** che specifica se debba essere applicata la sovrattassa di drop-off (vero = sì, falso = no). Il metodo deve:
 - Verificare accuratamente che tutti gli argomenti ricevuti siano non nulli, lanciando eventualmente una **NullPointerException** con apposito e specifico messaggio d'errore
 - Verificare che la città non sia una stringa vuota o blank, lanciando in caso **IllegalArgumentException** con apposito e dettagliato messaggio d'errore
 - Verificare che giorno e ora di fine noleggio non precedano quello di inizio noleggio, lanciando anche in tal caso **IllegalArgumentException** con apposito e dettagliato messaggio d'errore
 - Calcolare la durata del noleggio e applicare, in base ai giorni della settimana, la tariffa opportuna, ricordando che la tariffa speciale weekend va applicata solo se più conveniente rispetto alla tariffa standard; alla tariffa così ottenuta dovrà essere aggiunta la sovrattassa apposita nel caso di riconsegna in città diversa (drop-off).

Il metodo deve restituire un'opportuna istanza di **Result**, il cui messaggio abbia la forma:

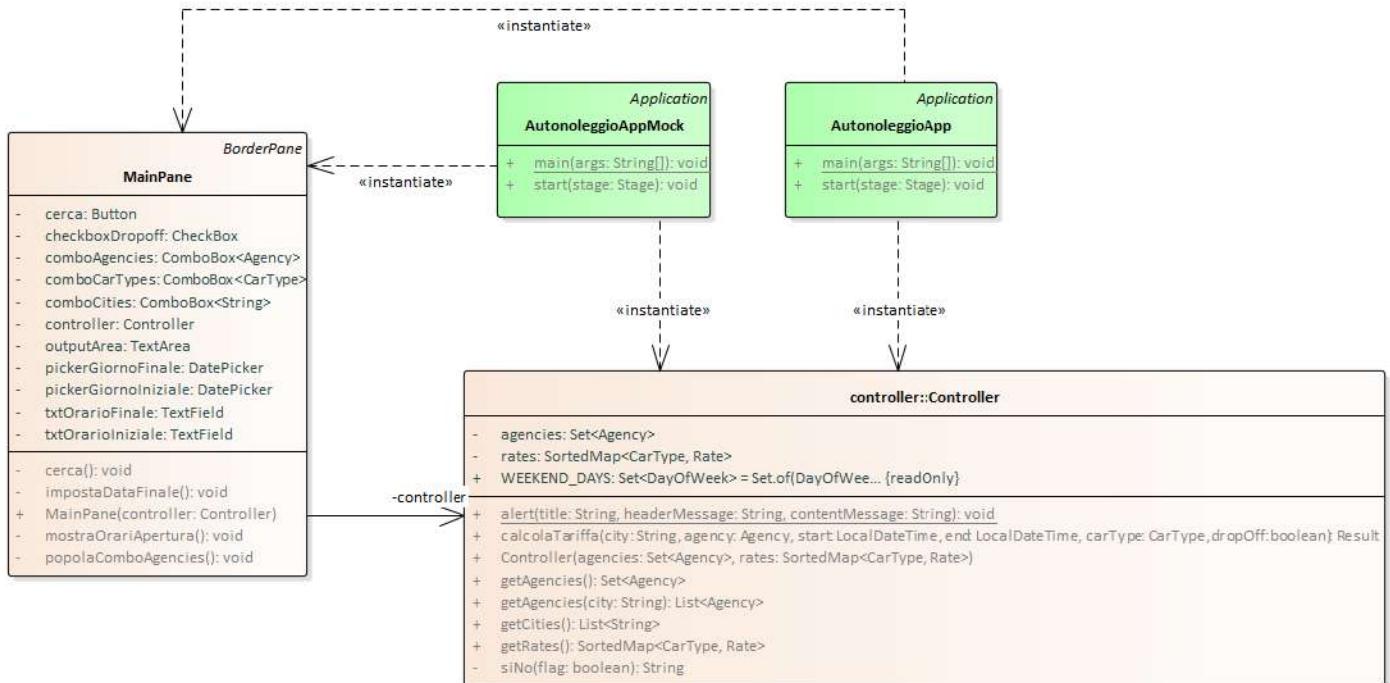
“Giorni: N, tariffa weekend: XX, dropOff: XX”

dove *N* è il numero di giorni e *XX* è una delle due stringhe “sì” o “no”.

Il metodo statico **alert**, utilizzabile dal **MainPane** quando è attiva la grafica, consente di far comparire all'utente una finestra di dialogo che mostri il messaggio d'errore specificato.

La classe **AutonoleggioApp** (fornita) costituisce l'applicazione JavaFX che si occupa di aprire i file, creare il controller e incorporare il **MainPane**. Per consentire di collaudare la GUI anche in assenza / in caso di malfunzionamento della parte di persistenza, è possibile avviare l'applicazione mediante la classe **AutonoleggioAppMock**.

L'interfaccia utente è illustrata nelle figure seguenti e segue il modello sotto illustrato:



- In alto vi sono le combo per la selezione della città (prima) e dell'agenzia (poi)
- Al centro vi sono i controlli per l'inserimento di data e ora di inizio e fine noleggio, del tipo di auto e per l'eventuale scelta di riconsegnare l'auto in una città diversa, nonché il pulsante CERCA che attiva il calcolo
- In basso vi è l'area di output, che mostra i messaggi destinati all'utente.

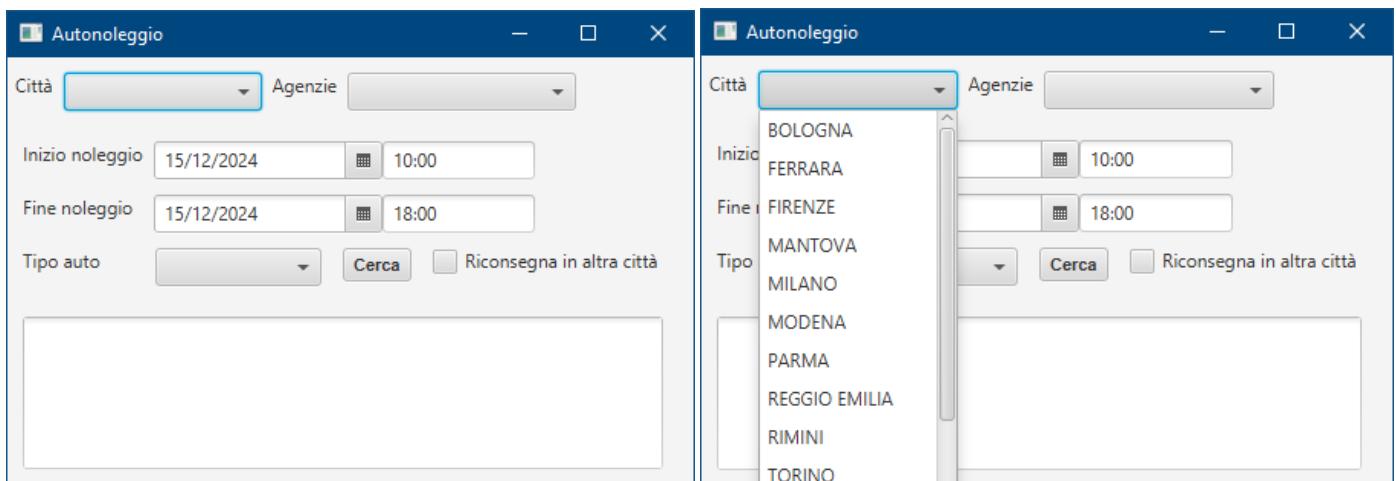


Fig. 1: situazione iniziale della GUI (a sinistra) ed elenco delle città (a destra).

Cliccando sulla combo città si può scegliere la città di proprio interesse: l'applicazione risponderà caricando nella combo agenzie le agenzie di quella specifica città (Fig. 2, sinistra). Selezionandone una, l'area di testo dovrà mostrare i rispettivi orari di apertura, correttamente formattati (Fig. 2, destra).

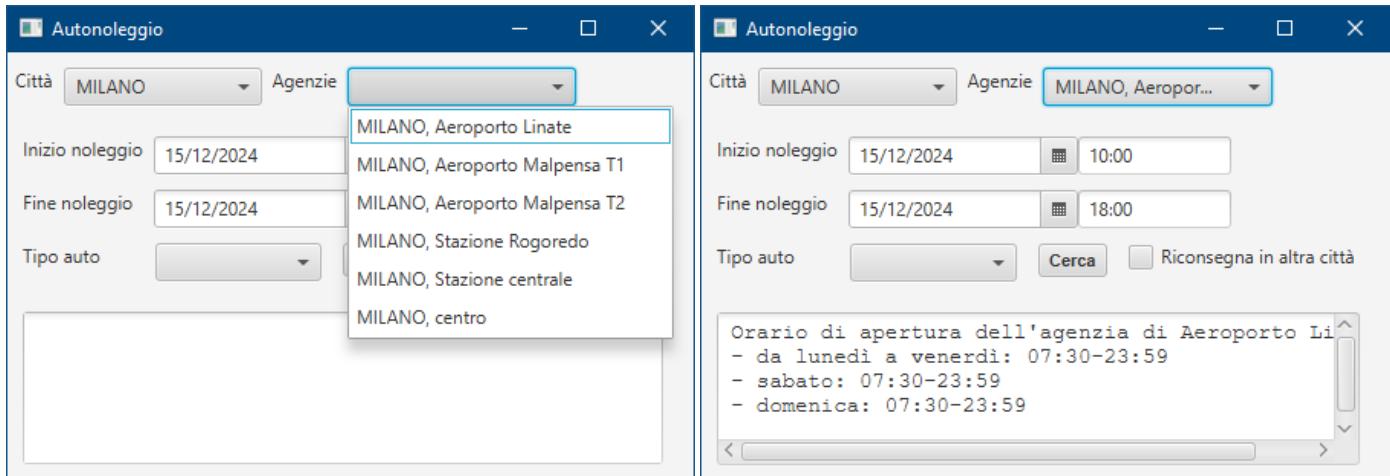


Fig. 2: selezione agenzie e visualizzazione orari di apertura dell'agenzia prescelta.

Scegliendo data e ora di inizio e fine noleggio, nonché il tipo di auto, è possibile farsi fare il preventivo premendo il pulsante “CERCA” (Fig.3, sinistra); si può facilmente vedere la differenza di costo nel caso di riconsegna in altra città, semplicemente selezionando l'apposita opzione (Fig. 3, destra).

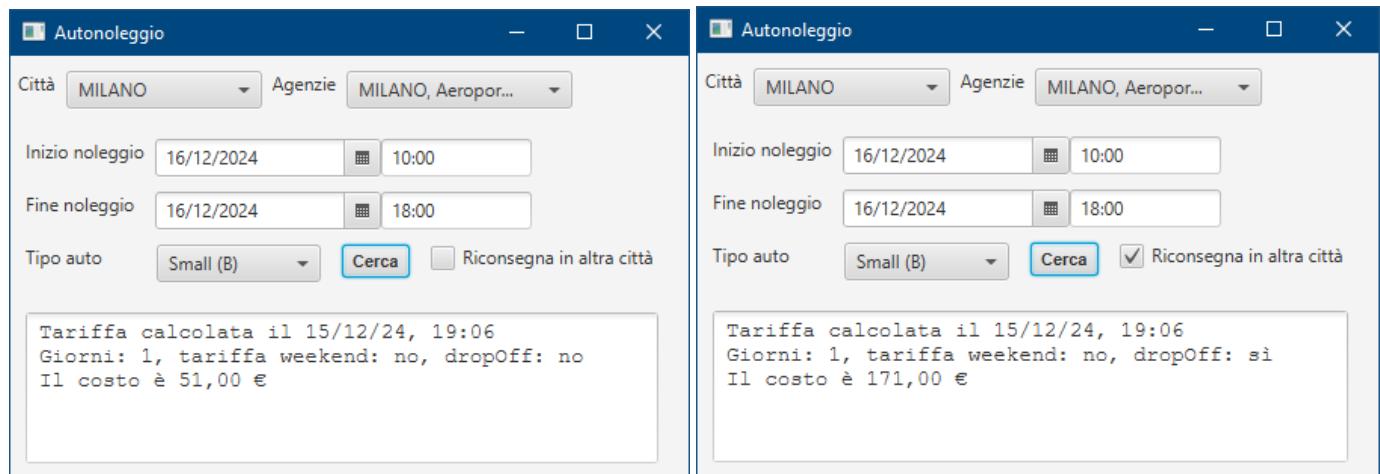


Fig. 3: calcolo del costo del noleggio (senza/con riconsegna in altra città)

Nel caso in cui il noleggio avvenga interamente in un weekend, dovrà essere applicata la tariffa apposita (Fig. 4)

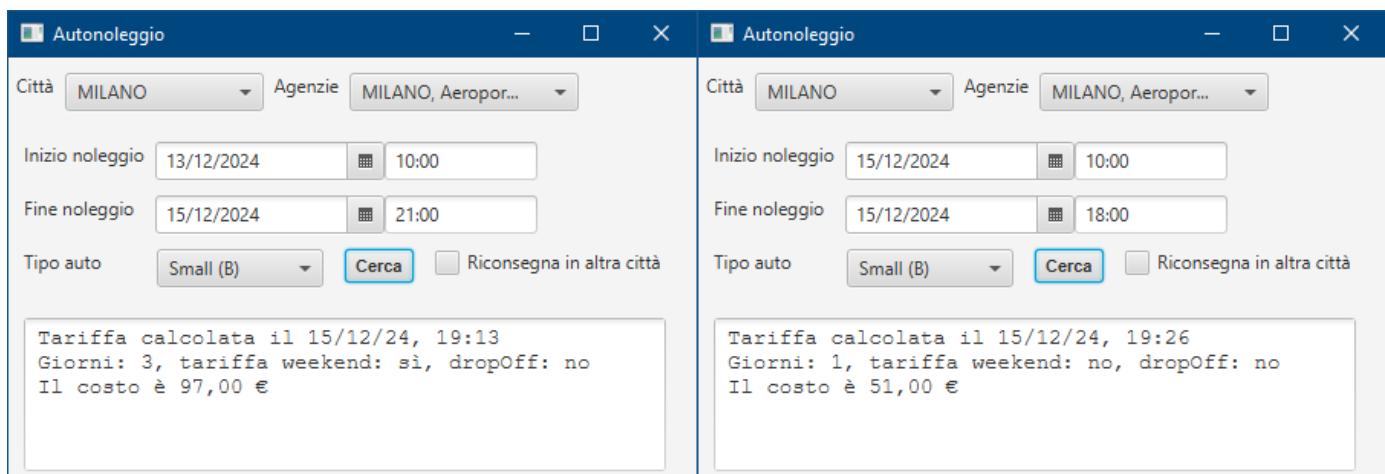


Fig. 4: caso di applicazione della tariffa weekend (sinistra) e di non applicazione perché non conveniente (destra)

L'applicazione deve segnalare errore, con apposito dialogo dotato di adeguato messaggio d'errore (Fig. 5), se:

- non è stata selezionata la città nell'apposita combo
- non è stata selezionata l'agenzia nell'apposita combo
- non sono state selezionate la data e/o l'ora di inizio e/o fine noleggio
- non è stato selezionato il tipo di auto nell'apposita combo
- [l'agenzia di noleggio è chiusa nel giorno/ora di inizio noleggio](#) (Fig 5, sotto).

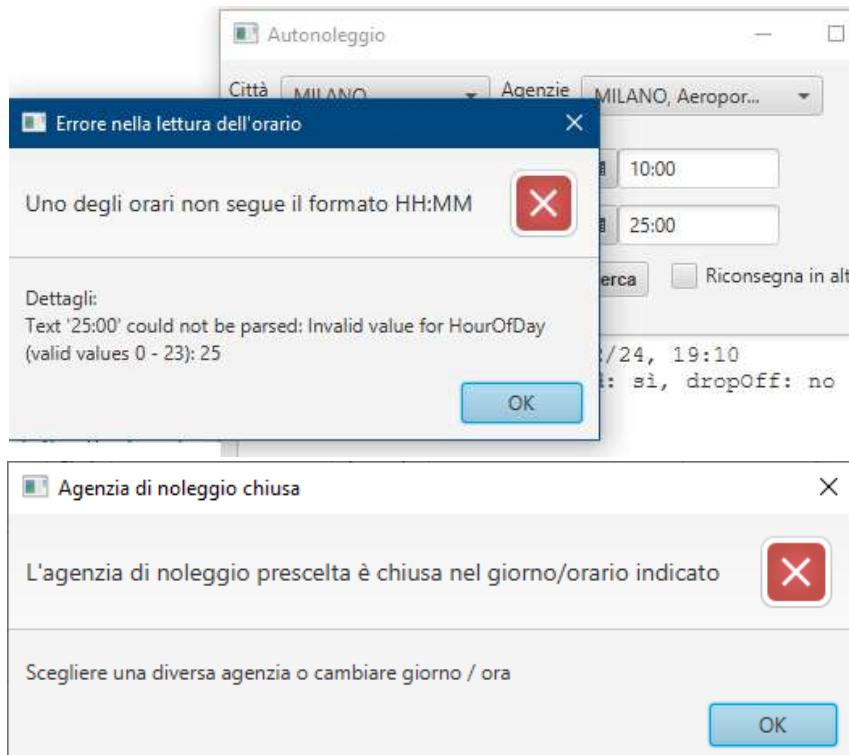


Fig. 5: segnalazioni di errore: sopra, orario illegale; sotto, agenzia chiusa al momento del ritiro dell'auto.

Il MainPane è fornito *parzialmente realizzato*: è presente tutta la parte strutturale, mentre rimangono da realizzare i gestori degli eventi (metodi privati *impostaDataFinale*, *popolaComboAgencies*, *mostraOrariApertura*, *cerca*).

In particolare:

- *impostaDataFinale* deve impostare sul calendario relativo alla data di fine noleggio lo stesso giorno selezionato dall'utente nel calendario relativo alla data di inizio noleggio
- *popolaComboAgencies* deve recuperare la città selezionata nell'apposita combo e popolare, di conseguenza, *comboAgencies* con le sole agenzie relative alla città selezionata; in caso di errore nella selezione della città (valore nullo), dovrà emettere tramite *alert* apposito messaggio d'errore all'utente
- *mostraOrariApertura* deve recuperare l'agenzia selezionata e mostrare a video, nell'area di output, i relativi orari di apertura, nel formato mostrato in figura; in caso di errore nella selezione dell'agenzia (valore nullo), dovrà emettere tramite *alert* apposito messaggio d'errore all'utente
- *cerca* deve recuperare dalla GUI tutti i dati utili, controllarli accuratamente e quindi controllare se l'agenzia sia aperta nel giorno e ora indicati per il ritiro dell'auto: in caso positivo, provvede a invocare il metodo *calcolaTariffa* del *Controller* e mostrare a video, nell'area di output, il dettaglio del calcolo (ovvero il contenuto del messaggio interno a *Result*) e il costo risultante, nel formato mostrato in figura. Nel caso l'agenzia sia chiusa nel momento indicato, dovrà essere emesso apposito messaggio d'errore, tramite il metodo *alert* del Controller (Fig. 5, sotto). Gli altri controlli da effettuare riguardano:

- la verifica che tutte le combo siano state correttamente selezionate (ossia, i valori da essere recuperati non siano nulli), emettendo nel caso, per ciascuno, un apposito messaggio d'errore tramite ***alert*** (Fig. 5, sopra);
- che l'orario iniziale e finale siano correttamente formattati nella forma HH:MM, ovvero che i valori di HH e MM siano corretti per un orario, emettendo nel caso, per ciascuno, un apposito messaggio d'errore tramite ***alert*** (Fig. 5, sopra).

Cose da ricordare

- salva costantemente il tuo lavoro: l'informatica a volte può essere "subdolamente ostile"...
- in particolare: se ora compila e stai per fare modifiche, salva la versione attuale (non si sa mai)

Checklist di consegna

- Hai fatto un **JAR eseguibile**, che contenga cioè l'indicazione del main?
- Hai controllato che **si compili e ci sia tutto**? [NB: non includere il PDF del testo]
- Hai **rinominato IL PROGETTO**, lo ZIP e il JAR esattamente come richiesto?
- Hai **chiamato** la cartella del progetto esattamente come richiesto?
- **Hai fatto un unico file ZIP (NON .7z, rar o altri formati) contenente l'intero progetto?**
In particolare, ti sei assicurato di aver incluso tutti i file .java (e non solo i .class)?
- **Hai consegnato DUE file distinti, ossia lo ZIP col progetto e il JAR eseguibile?**
- Su EOL, hai **premuto** il tasto "CONFERMA" per inviare il tuo elaborato?

ESAME DI FONDAMENTI DI INFORMATICA T-2 dell'11/9/2024

Proff. E. Denti – R. Calegari – A. Molesini

Tempo a disposizione: 3h30

NOME PROGETTO ECLIPSE: CognomeNome-matricola (es. RossiMario-0000123456)

NOME CARTELLA PROGETTO: CognomeNome-matricola (es. RossiMario-0000123456)

NOME ZIP DA CONSEGNARE: CognomeNome-matricola.zip (es. RossiMario-0000123456.zip)

NOME JAR DA CONSEGNARE: CognomeNome-matricola.jar (es. RossiMario-0000123456.jar)

Si devono consegnare DUE FILE: *l'intero progetto Eclipse e il JAR eseguibile*

Si ricorda che compiti non compilabili, o che non passino almeno 2/3 dei test o siano palesemente lontani da 18/30 NON SARANNO CORRETTI e causeranno la verbalizzazione del giudizio “RESPINTO”.

È stato richiesto lo sviluppo di un'applicazione che, a partire da un classico cruciverba già risolto, produca un crittocruciverba (o cruciverba cifrato), ossia quel tipo di cruciverba in cui al solutore non vengono date definizioni per le parole, ma vige il principio secondo cui “a lettera uguale corrisponde numero uguale”: il solutore dovrà ricostruire lo schema in base al proprio intuito, partendo dalle poche lettere inizialmente disposte sullo schema.

L'applicazione dovrà generare il crittocruciverba, popolarlo con le lettere iniziali previste e poi mostrarlo a video così da consentire al giocatore-utente di risolverlo interattivamente. In base alla bravura del giocatore, si potrà scegliere fra due modalità, AGEVOLATA ed ESPERTA:

- in modalità agevolata, le lettere inizialmente fornite saranno 4 e l'interfaccia grafica non consentirà al solutore di inserire associazioni lettera/numero errate;
- in modalità esperta, invece, le lettere fornite saranno solo 3 e non vi sarà alcun impedimento a inserire associazioni errate.

DESCRIZIONE DEL DOMINIO DEL PROBLEMA

Nel crittocruciverba le caselle sono numerate secondo il principio “a lettera uguale corrisponde numero uguale”. Lo schema si presenta inizialmente vuoto, con le sole caselle nere: le celle vuote sono pre-numerate secondo tale principio. Il solutore deve ricostruire lo schema attribuendo una lettera diversa a ogni numero fino a formare tutte parole di senso compiuto.

Per generare il crittocruciverba si parte da un cruciverba classico già risolto (immagine in alto) e se ne numerano le celle partendo da quella in alto a sinistra e procedendo poi verso destra, dall'alto in basso, secondo il predetto principio: la prima lettera sarà quindi messa in corrispondenza col numero 1, la seconda (se diversa dalla precedente) col numero 2, e così via. Naturalmente, una volta che si stabilisce una corrispondenza lettera-numero, tutte le caselle che contengono la stessa lettera saranno etichettate con lo stesso numero. Per ipotesi, le caselle nere non sono numerate. Nel caso dell'esempio sopra, ciò produce la situazione illustrata nell'immagine intermedia.

Per proporre lo schema al giocatore, tuttavia, è necessario disporre preliminarmente alcune lettere sullo schema: a tal fine viene scelta casualmente una parola corta, composta rispettivamente di 3 (modalità esperta) o 4 (modalità agevolata) lettere distinte, che vengono poi riprodotte ovunque sullo schema (nell'esempio sopra, in modalità agevolata, è stata sorteggiata la parola “NOLI”, quindi l'intero schema è stato pre-popolato con le associazioni previste per le lettere ‘L’ = 1, ‘I’ = 2, ‘N’ = 5, ‘O’ = 6).

L	I	■	A	V	■	N	O	L	I	■	■	V	L	A	D	■
E	S	T	■	A	L	E	A	■	R	E	T	E	■	S	I	M
M	A	I	■	J	E	T	■	■	■	V	E	R	B	A	N	O
■	A	C	○	○	○	○	○	○	○	○	○	○	○	○	○	○
E	C	■	E	N	T	U	S	I	A	S	M	A	R	■	S	■
A	■	A	T	T	A	N	A	G	L	I	A	T	O	■	M	E
■	I	N	T	■	M	O	L	E	C	O	■	L	A	■	L	■
C	A	T	O	N	E	■	T	R	O	N	I	■	S	E	G	A
O	S	E	■	I	N	I	A	■	E	■	S	U	G	A	R	■
C	I	■	A	L	T	E	R	C	■	A	N	S	A	■	G	■
O	■	A	L	O	E	■	E	N	O	T	E	C	A	■	A	O

01 02	■ 03 04	■ 05 06	01 02	■ ■ 04 01	03 07	■
08 09 10	■ 03 01 08 03	■ 11 08 10 08	■ 09 02 12			
12 03 02	■ 13 08 10	■ ■ ■ 04 08 11 14	03 05 06			
■ 03 15 15 06 05 10 08 05 10 08 05 10 08 03 11 09 02 ■ 06 11						
08 15 ■ 08 05 10 16 09 02 03 09 12 03 11 08 ■ 09 03 ■ 03 10 10 03 05 03 17 01 02 03 10 06 ■ 12 08						
■ 02 05 10 ■ 12 06 01 08 15 06 01 03 ■ 01 03 ■ 15 03 10 06 05 08 ■ 10 11 06 05 02 ■ 09 08 17 03 15 03 10 06 05 08 ■ 02 05 02 03 ■ ■ 08 ■ 09 16 17 03 11 06 09 08 ■ 03 01 10 08 11 15 06 ■ 03 05 09 03 ■ 17 06 ■ 03 01 06 08 ■ 08 05 06 10 08 15 03 ■ 03 06						

L I ■ 03 04 N O L I ■ 04 L 03 07 ■
10 09 10 ■ 03 L 08 03 ■ 11 08 10 08 ■ 09 I 12
12 03 13 ■ 13 08 10 ■ ■ ■ 04 08 11 14 03 N O
■ 03 15 15 O 10 08 ■ 10 03 11 09 ■ ■ O 11
10 08 15 ■ 08 N 10 16 09 I 03 09 12 03 11 08 ■ 09
10 03 10 10 03 N 03 17 L I 03 10 01 O ■ 12 08
■ I N 10 ■ 12 O L 08 15 O L 03 10 ■ L 03 ■
15 03 10 10 08 ■ 10 11 O N 10 ■ 10 11 O N I ■ 09 08 17 03
O 09 08 ■ I N ■ 03 ■ 08 ■ 08 ■ 09 16 17 03 11
15 02 ■ 03 01 10 08 11 15 06 ■ 03 05 09 03 ■ 17 06 ■ 03 01 06 08 ■ 08 05 06 10 08 15 03 ■ 03 06

Ovviamente, non è detto che uno schema usi tutte le 26 lettere dell'alfabeto: i numeri utilizzati saranno quindi una sequenza di valori continui da 1 fino al numero di lettere distinte presenti nello schema (nell'esempio sopra 17, non essendo presenti le nove lettere F,H,K,P,Q,W,X,Y,Z).

Il file di testo `schema.txt` contiene, nel formato descritto più oltre, il cruciverba di partenza risolto. L'applicazione dovrà:

- a) leggere tale file e caricare il cruciverba risolto
 - b) utilizzarlo per generare lo schema numerato del crittocruciverba
 - c) permettere al giocatore-solutore, tramite la GUI, di risolvere per tentativi lo schema inserendo via via le associazioni lettera/numero e mostrando l'evoluzione del gioco. In modalità AGEVOLATA, la GUI deve altresì prevenire l'inserimento di associazioni errate.

TEMPO STIMATO PER SVOLGERE L'INTERO COMPITO: 2h15 – 3h

PARTE 1 – Modello dei dati: Punti 7	[TEMPO STIMATO: 30-40 minuti]
PARTE 2 – Persistenza: Punti 8	[TEMPO STIMATO: 30-45 minuti]
PARTE 3 – Grafica: Punti 15	[TEMPO STIMATO: 75-95 minuti]

NUMERO MINIMO DI TEST CON SUCCESSO PERCHÉ IL COMPITO SIA CORRETTO

Considerando solo SchemaNumerato , MyReader e MyController 18 su 27
Considerandoli tutti: 26 su 38

JAVAFX – Parametri run configuration nei LAB

```
--module-path "C:\applicativi\moduli\javafx-sdk-21.0.2\lib"  
--add-modules javafx.controls
```

Cose da ricordare

- salva costantemente il tuo lavoro: l'informatica a volte può essere “subdolamente ostile”..
 - in particolare: se ora compila e stai per fare modifiche, salva la versione attuale (non si sa mai)

Checklist di consegna

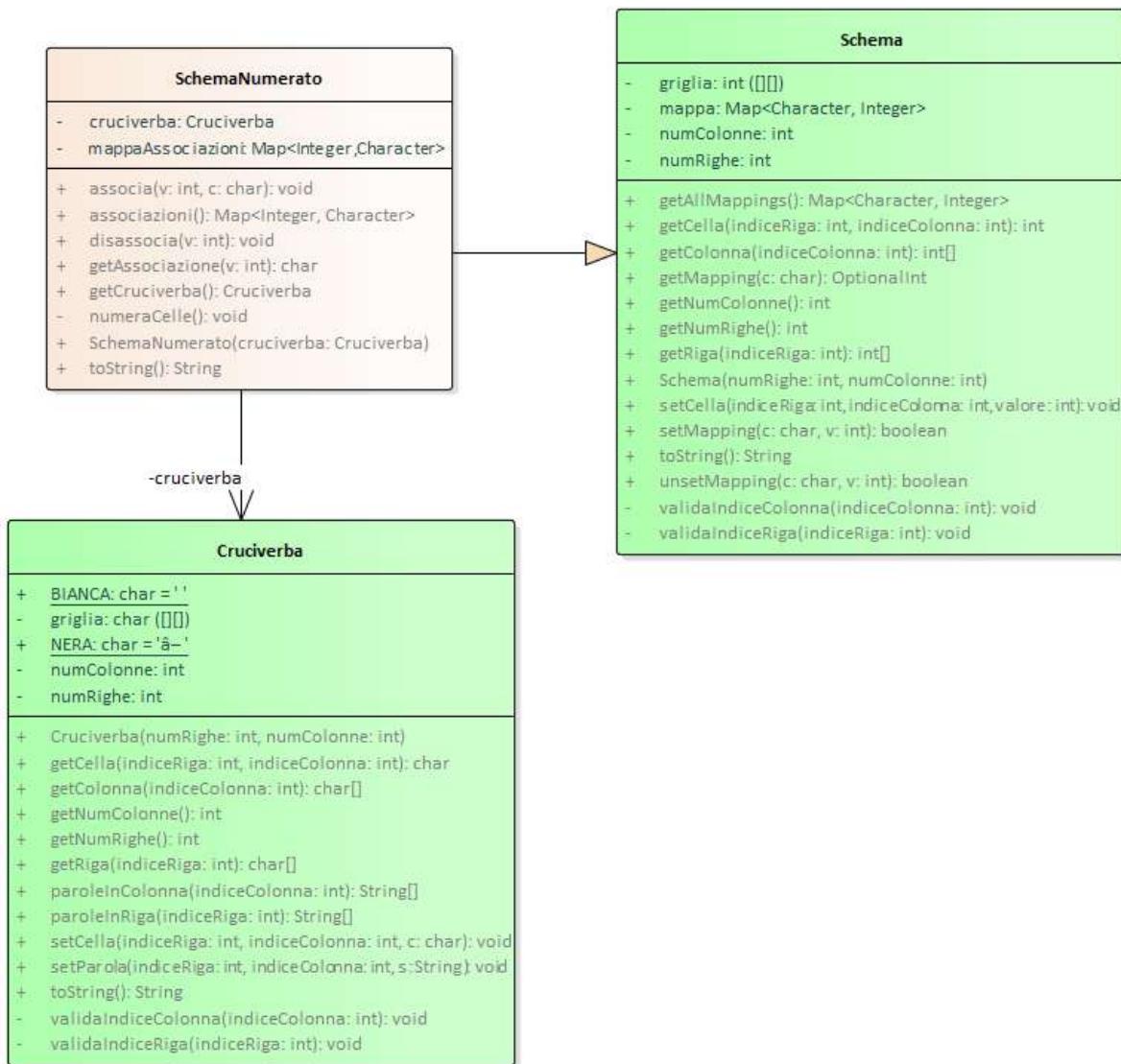
- Hai fatto un **JAR eseguibile**, che contenga cioè l'indicazione del main?
 - Hai controllato che **si compili e ci sia tutto**? [NB: non includere il PDF del testo]
 - Hai **rinominato IL PROGETTO**, lo ZIP e il JAR esattamente come richiesto?
 - Hai **chiamato** la cartella del progetto esattamente come richiesto?
 - Hai fatto un **unico file ZIP (NON .7z, rar o altri formati)** contenente l'intero progetto?
In particolare, ti sei assicurato di aver incluso tutti i file .java (e non solo i .class)?
 - **Hai consegnato DUE file distinti, ossia lo ZIP col progetto e il JAR eseguibile?**
 - Su EOL, hai **premuto** il tasto “CONFERMA” per inviare il tuo elaborato?

Parte 1 – Modello dei dati

(punti: 7)

Package: cruciverba.model

[TEMPO STIMATO: 30-40 minuti]



SEMANTICA:

- La classe **Cruciverba** (fornita) rappresenta un cruciverba classico:
 - Sono definite le due costanti **BIANCA** e **NERA**, atte a rappresentare rispettivamente la casella vuota (non ancora numerata) o nera (da non numerare)
 - Il costruttore alloca la griglia di caratteri e la popola inizialmente con tutte caselle nere, verificando preventivamente che il numero di righe e colonne fornito sia positivo (altrimenti viene lanciata un'apposita **IllegalArgumentException** con idoneo messaggio d'errore); il numero di righe e colonne è recuperabile tramite l'apposita coppia di accessori `getNumRighe/getNumColonne`
 - La coppia di metodi `getCella/setCella` permette di recuperare/impostare il carattere nella cella di coordinate (riga,colonna) specificate: entrambi lanciano **IllegalArgumentException** con idoneo messaggio d'errore nel caso di indici fuori range
 - Il metodo di utilità `setParola` permette di impostare rapidamente una singola parola orizzontale in una serie di celle adiacenti, posto che naturalmente non si ecceda la dimensione prevista; in tutti i casi problematici viene lanciata **IllegalArgumentException** con idoneo messaggio d'errore

- La coppia di metodi `paroleInRiga/paroleInColonna` restituisce l'elenco, sotto forma di array di stringhe, di parole presenti nella riga o colonna specificata; anche in questo caso naturalmente viene lanciata `IllegalArgumentException` con idoneo messaggio d'errore nel caso di indici fuori range
- un'apposita `toString` emette una rappresentazione stampabile del cruciverba.
- La classe **Schema** (fornita) rappresenta invece lo schema base di un crittocruciverba: mantiene al suo interno una griglia di interi e una tabella in cui registra, per ogni carattere, il numero ad esso attribuito. *Nel seguito ci si riferirà a questa corrispondenza come "mapping". Tale tabella riflette semplicemente il cruciverba iniziale: non ha alcuna relazione con le azioni del solutore, che saranno gestite altrove.*
 - Il costruttore alloca la griglia di numeri di dimensioni date (due valori strettamente positivi), recuperabili tramite l'apposita coppia di accessori `getNumRighe/getNumColonne`; predisponde inoltre la tabella di mapping caratteri/numeri, popolandola inizialmente con l'unica entry della casella **NERA**, a cui viene fatto corrispondere convenzionalmente il numero 0 (le altre caselle sono numerate a partire da 1)
 - La coppia di metodi `getCella/setCella` permette di recuperare/impostare il numero nella cella di coordinate (riga,colonna) specificate: entrambi lanciano `IllegalArgumentException` con idoneo messaggio d'errore nel caso di indici fuori range
 - La coppia di metodi `getRiga/getColonna` recupera l'intera riga (o colonna) di indice specificato: ovviamente lanciano `IllegalArgumentException` nel caso di indici fuori range
 - La terna di metodi `setMapping/unsetMapping/getMapping` gestisce i mapping carattere/numero, rispettivamente impostando (`setMapping`), rimuovendo (`unsetMapping`) o recuperando (`getMapping`) un mapping carattere/numero; sono accettabili solo caratteri maiuscoli e valori numerici compresi fra 1 e 26: diversamente verrà lanciato `IllegalArgumentException` con idoneo messaggio d'errore
 - un'apposita `toString` emette una rappresentazione stampabile dello schema.
- La classe **SchemaNumerato** (da completare) specializza **Schema** incorporando la logica di numerazione delle caselle e gestendo le associazioni intero/carattere inserite dall'utente durante il gioco. Da notare che, mentre i "mapping" lettera/numero della classe precedente sono pre-calcolati all'atto della generazione del crittocruciverba perché riflettono semplicemente la numerazione attribuita ai caratteri, queste "associazioni" numero/lettera riflettono invece le "mosse" del giocatore che cerca di risolvere lo schema e verranno quindi aggiunte/tolte dinamicamente durante il gioco. Più precisamente:
 - il costruttore riceve un **Cruciverba**, che usa per istanziare uno **Schema** di uguali dimensioni, e predispone la mappa in cui saranno poi registrate le associazioni numero/lettera durante il gioco; infine numera le caselle, tramite il metodo privato ausiliario `numeraCelle` (fornito), che incorpora la logica di numerazione descritta nel Dominio del Problema.
 - La quaterna di metodi **associa/disassocia/getAssociazione/associazioni (da realizzare)** gestisce le associazioni numero/lettera, rispettivamente impostando (`associa`), rimuovendo (`disassocia`) o recuperando (`getAssociazione`) una specifica associazione numero/lettera; se l'associazione non è presente, per convenzione getAssociazione deve restituire la costante BIANCA. Il metodo `associazioni` restituisce l'intera mappa con tutte le associazioni attualmente in essere: pertanto, essa *non deve contenere* le associazioni ancora mancanti (neppure a livello di chiavi). Come già per **Schema**, sono accettabili solo valori numerici compresi fra 1 e 26 e, nel caso di `associa`, il carattere deve essere maiuscolo: altrimenti verrà lanciata `IllegalArgumentException` con idoneo messaggio d'errore
 - un'apposita `toString` emette una rappresentazione stampabile dello schema numerato.

Parte 2 – Persistenza

(punti: 8)

Package: cruciverba.persistence

[TEMPO STIMATO: 30-45 minuti]

Il file di testo `schema.txt` descrive il cruciverba classico, nel formato sotto descritto e illustrato nell'esempio in calce:

- la prima riga contiene l'intestazione “Cruciverba **n** x **m**”, dove **n** ed **m** sono interi positivi (**n** = numero di righe, **m** = numero di colonne = lunghezza della riga) e le parole “Cruciverba” e “x” sono scritte esattamente così; i vari termini sono separati fra loro da *uno o più spazi*
- tutte le altre **n** righe hanno *eguale struttura* e devono contenere lo stesso *numero m di elementi*: ogni riga contiene una sequenza di caratteri singoli (per le caselle vuote, il carattere spazio) separati dal carattere speciale '|', usato anche come marca di inizio e fine riga.

```
Cruciverba 11 x 17
|L|I| |A|V| |N|O|L|I| | |V|L|A|D| |
|E|S|T| |A|L|E|A| |R|E|T|E| |S|I|M|
|M|A|I| |J|E|T| | | |V|E|R|B|A|N|O|
| |A|C|C|O|N|T|E|N|T|A|R|S|I| |O|R|
|E|C| |E|N|T|U|S|I|A|S|M|A|R|E| |S|
|A| |A|T|T|A|N|A|G|L|I|A|T|O| |M|E|
| |I|N|T| |M|O|L|E|C|O|L|A| |L|A| |
|C|A|T|O|N|E| |T|R|O|N|I| |S|E|G|A|
|O|S|E| |I|N|I|A| | |E| |S|U|G|A|R|
|C|I| |A|L|T|E|R|C|O| |A|N|S|A| |G|
|O| |A|L|O|E| |E|N|O|T|E|C|A| |A|O|
```

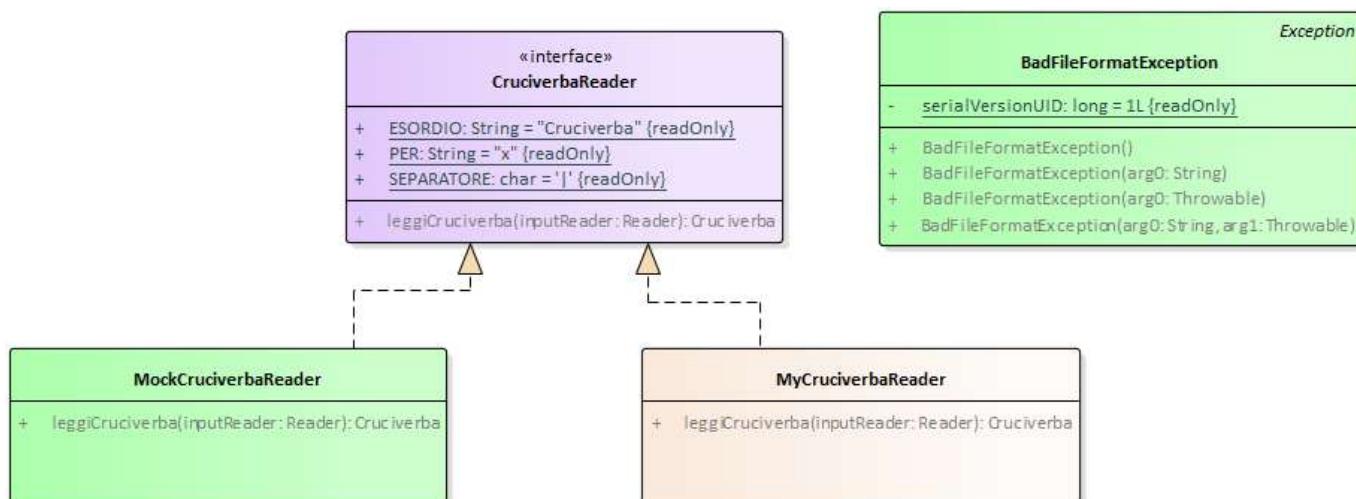
SEMANTICA:

a) L'interfaccia `CruciverbaReader` (fornita) dichiara:

- le tre costanti SEPARATORE, ESORDIO e PER che fattorizzano rispettivamente le costanti '|' (un carattere singolo), “Cruciverba” e “x” (stringhe)
- il metodo `leggiCruciverba`, che legge dal `Reader` (già aperto) fornito i dati necessari e restituisce un `Cruciverba` perfettamente configurato; esso lancia `BadFormatException` con opportuno messaggio d'errore in caso di problemi nel formato del file, ivi inclusi i casi in cui il numero di colonne non sia **m** e/o il numero di righe non sia **n**, o `IOException` in caso di altri problemi di I/O

b) La classe `MyCruciverbaReader` (da completare) implementa `CruciverbaReader`:

- non c'è alcun costruttore
- Il metodo `leggiCruciverba` legge il file, lo interpreta secondo le specifiche sopra fornite e, se tutto è ok, crea e popola il `Cruciverba` cella per cella; ogni deviazione dal formato previsto, incluso il caso in cui il numero di colonne non sia **m** e/o il numero di righe non sia **n**, deve causare una `BadFormatException` con dettagliato e specifico messaggio d'errore (vedere test)



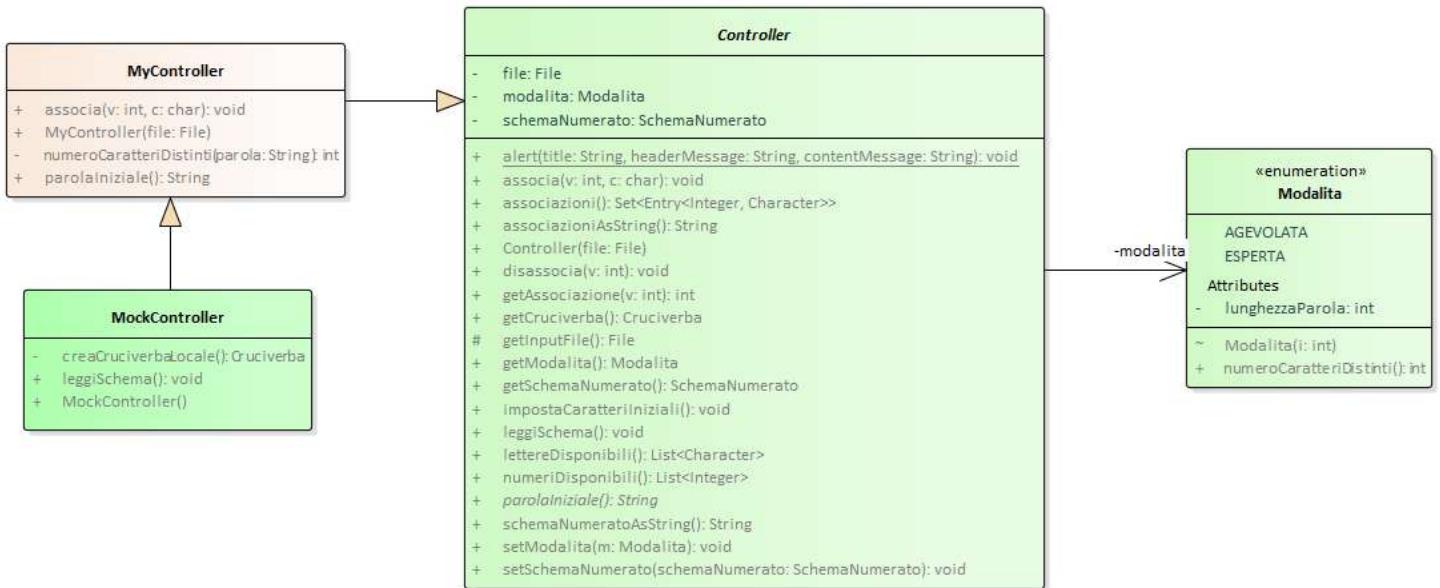
Parte 3

(punti: 15)

Package: cruciverba.controller

[TEMPO STIMATO: 35-45 minuti] (punti 8)

Il Controller costituisce il nucleo centrale di questa applicazione: crea e mantiene le strutture dati, gestisce la logica di gioco e le modalità dello stesso. A tal fine è organizzato come segue:



SEMANTICA:

- a) La classe astratta **Controller** (fornita) implementa già il grosso delle funzionalità: rimane astratto solo il metodo **parolaIniziale**, che dovrà essere implementato in una sottoclasse concreta.
- Il costruttore riceve il **File** da cui dovranno essere letti i dati (cioè il cruciverba classico di partenza), recuperabile dall'omonimo accessor **getInputFile**
 - Il metodo **leggiSchema** legge, tramite un apposito **CruciverbaReader** di sua proprietà, il **File** fornito e usa il **Cruciverba** così ottenuto (eventualmente recuperabile tramite l'accessor **getCruciverba**) per creare lo **SchemaNumerato** corrispondente, che viene quindi memorizzato nel proprio stato interno; imposta altresì, di default, la modalità di funzionamento **ESPERTA**.
 - Il metodo **impostaCaratteriIniziali** si procura, tramite il metodo **parolaIniziale** (da realizzare nella sottoclasse concreta), la parola iniziale sorteggiata, ne estrae i caratteri distinti e li usa per impostare sia il mapping lettera/numero (tramite il metodo **setMapping** di **SchemaNumerato**) sia l'associazione duale numero/lettera (tramite il metodo **associa** di **SchemaNumerato**), così da tenere tutto sincronizzato
 - I due metodi **lettereDisponibili** e **numeriDisponibili** restituiscono rispettivamente la lista ordinata delle sole lettere / dei soli numeri presenti nel crittocruciverba ma non ancora usati dal giocatore/solutore (ossia, presenti nei mapping ma non nelle associazioni): quindi, eventuali lettere dell'alfabeto non presenti in un dato schema (come 'W' nell'esempio) non saranno mai restituite.
 - Le due coppie di accessori **getModalita/setModalita**, **setSchemaNumerato/schemaNumeratoAsString** accedono / impostano rispettivamente la modalità e lo schema numerato interni al Controller.

Il metodo statico **alert**, utilizzabile dal **MainPane** quando è attiva la grafica, consente di far comparire all'utente una finestra di dialogo che mostri il messaggio d'errore specificato.

La sottoclasse **ControllerMock** (fornita) è utilizzata esclusivamente da **CruciverbaAppMock** (fornita, vedere sotto) per simulare la presenza dei dati letti da file: pertanto, non deve essere considerata in questo compito.

- b) La classe concreta **MyController** (da realizzare) implementa il metodo **parolaIniziale** astratto nel **Controller** e ridefinisce conservativamente il metodo **associa** in modo che tenga conto anche della modalità di funzionamento (AGEVOLATA o EXPERTA) prescelta. Più precisamente:

- Il costruttore riceve il **File** da cui dovranno essere letti i dati (cioè il cruciverba classico di partenza), e rimpalla l'operazione sulla classe base
- Il metodo **associa (da realizzare)**, già definito in **Controller**, deve essere ridefinito in modo che, nel caso di modalità AGEVOLATA, ogni tentativo di impostare una associazione lettera/numero errata venga respinto lanciando **UnsupportedOperationException** con idoneo messaggio; diversamente, se tutto è ok, deve rimpallare l'operazione sulla classe base.
- Il metodo **parolaIniziale (da realizzare)** contiene l'algoritmica fondamentale: esso deve dapprima estrarre dal **Cruciverba** tutte le parole esistenti in riga, poi filtrare solo quelle in cui il numero di lettere distinte sia coerente con la modalità selezionata (ossia, 3 per la modalità ESPERTA e 4 per la modalità AGEVOLATA), infine sorteggiare a caso una parola fra queste e restituirla. Ai fini di questo compito, si può fare l'ipotesi che una tale parola esista sempre.

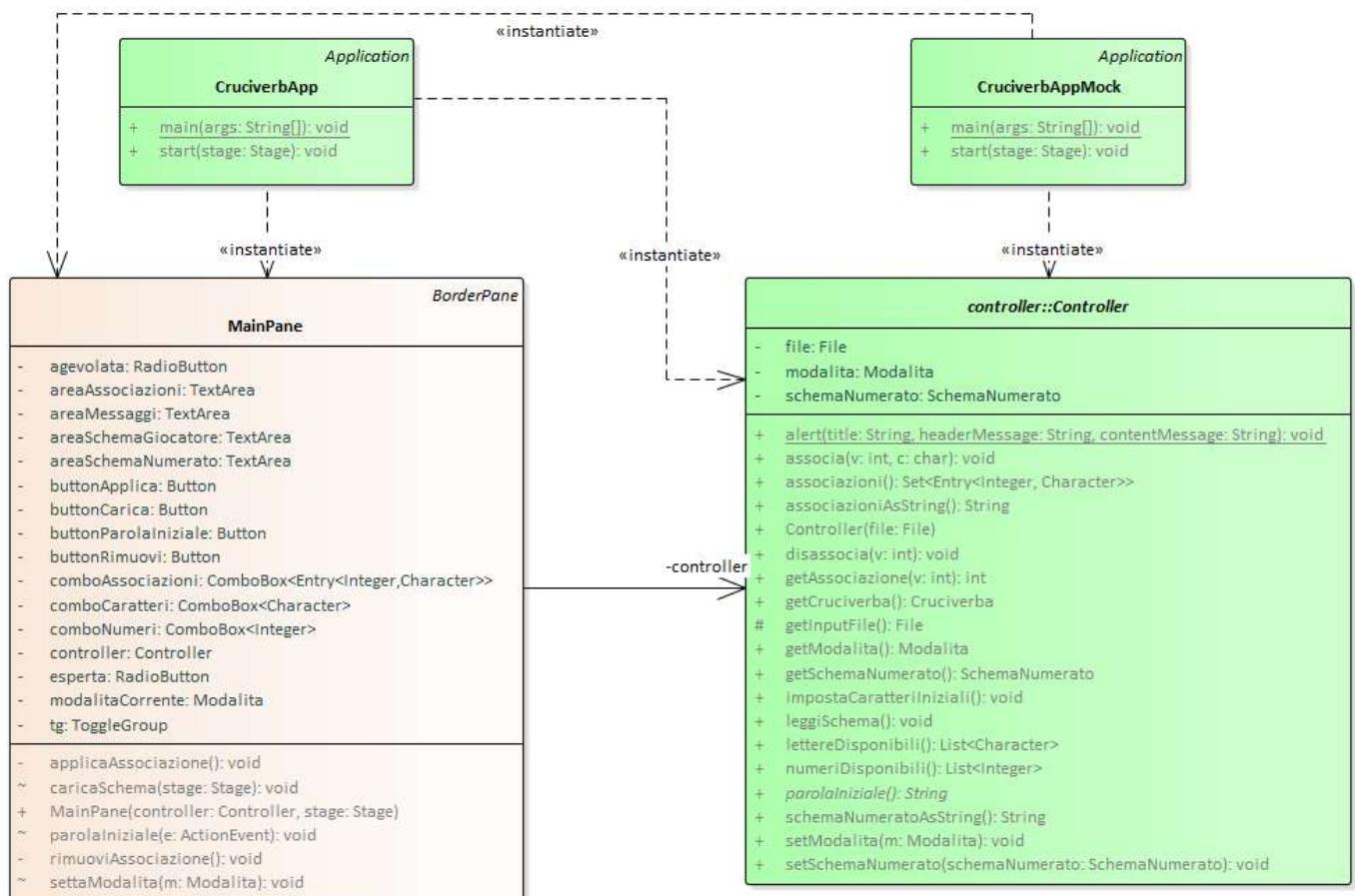
NB: il numero di caratteri distinti è ottenibile dall'omonimo metodo di ausilio fornito a corredo.

Package: cruciverba.ui

[TEMPO STIMATO: 40-50 minuti] (punti 7)

La classe **CruciverbApp** (fornita) costituisce l'applicazione JavaFX che si occupa di aprire i file, creare il controller e incorporare il **MainPane**. Per consentire di collaudare la GUI anche in assenza / in caso di malfunzionamento della parte di persistenza, è possibile avviare l'applicazione mediante la classe **CruciverbAppMock** (che si avvale del controller ausiliario **MockController** e del **MockCruciverbaReader**).

L'interfaccia utente è illustrata nelle figure seguenti e segue il modello sotto illustrato:



- a sinistra vi sono tutti i pulsanti e i controlli (radiobutton, combobox, aree messaggi)
- a destra vi sono invece le due aree di output, che mostrano rispettivamente lo schema iniziale (sopra) e quello di gioco (sotto) secondo l'evoluzione corrente.

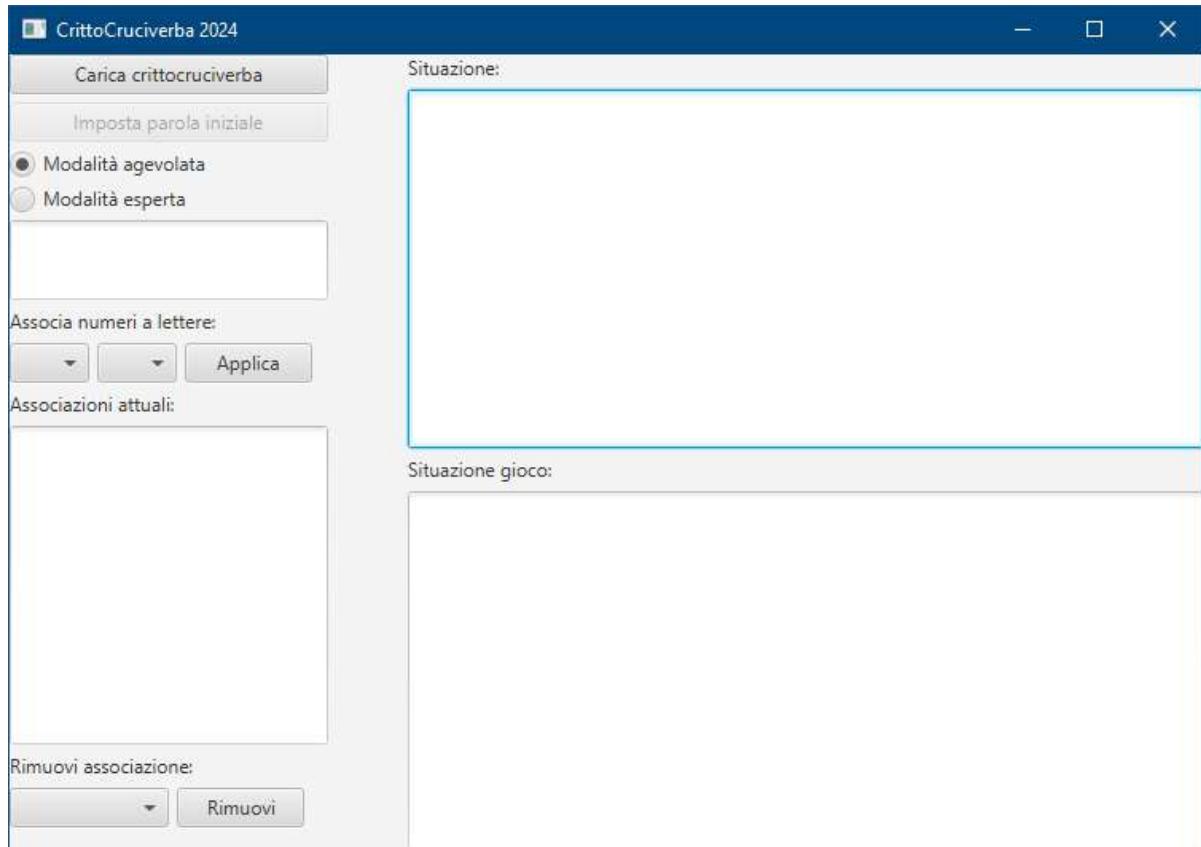


Fig. 1: situazione iniziale della GUI.

Inizialmente è usabile solo il pulsante “Carica Crittocruciverba”: ogni tentativo di cliccare su “Applica” o “Rimuovi” in questa fase causerà l’apparizione di un dialogo con msg d’errore (Fig. 2); parimenti, le combo sono tutte vuote.

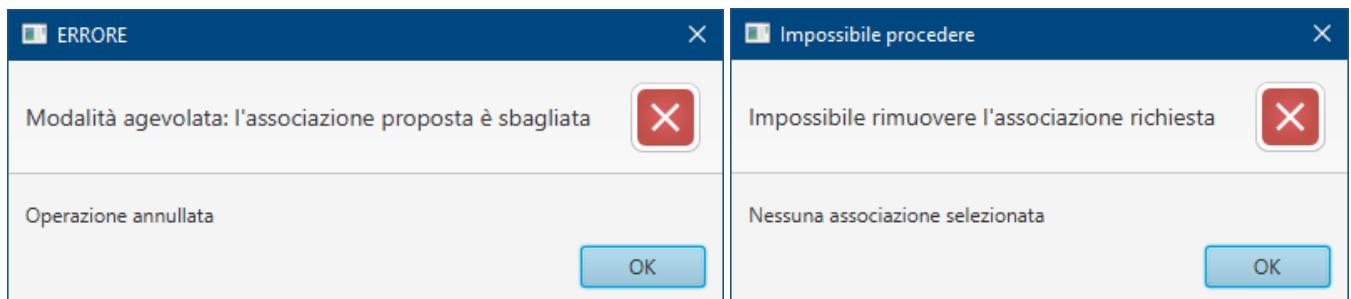


Fig. 2: messaggi d’errore premendo “Applica” o “Rimuovi” in questa fase iniziale.

Premendo il pulsante “Carica Crittocruciverba” viene caricato il cruciverba classico da file e viene generato e mostrato lo schema numerato: il pulsante di caricamento si disabilita, mentre si abilita il successivo che consente l’impostazione della parola iniziale, previa scelta della modalità (di default l’AGEVOLATA) (Fig. 3). L’area messaggi mostra le operazioni via via eseguite (caricamento cruciverba, selezione modalità).

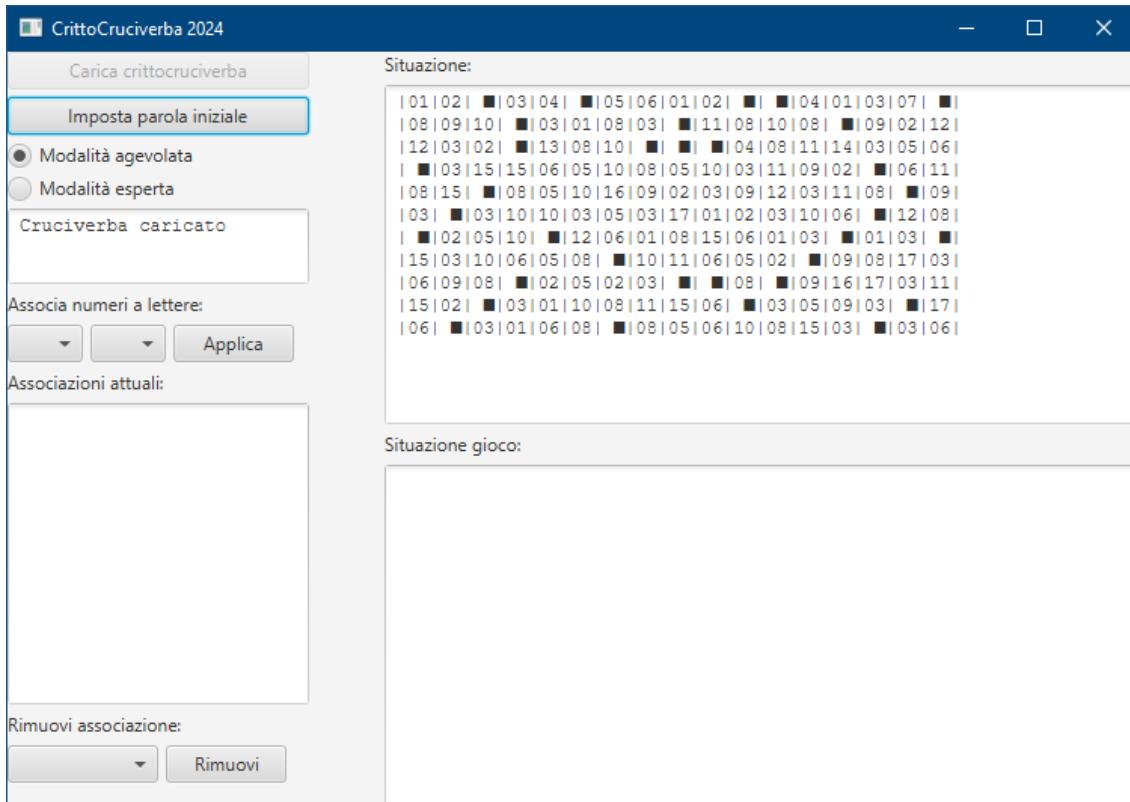


Fig. 3: la GUI dopo aver premuto il pulsante “Carica crittocruciverba”.

Premendo ora il pulsante “Imposta parola iniziale”, il sistema sorteggerà una parola orizzontale adatta e ne propagherà le lettere su tutto lo schema, popolando altresì le combo di numeri e lettere con quelli attualmente disponibili. In Fig. 4 si vede la situazione risultante: in questo caso è stata sorteggiata la parola “VLAD” (prima riga).

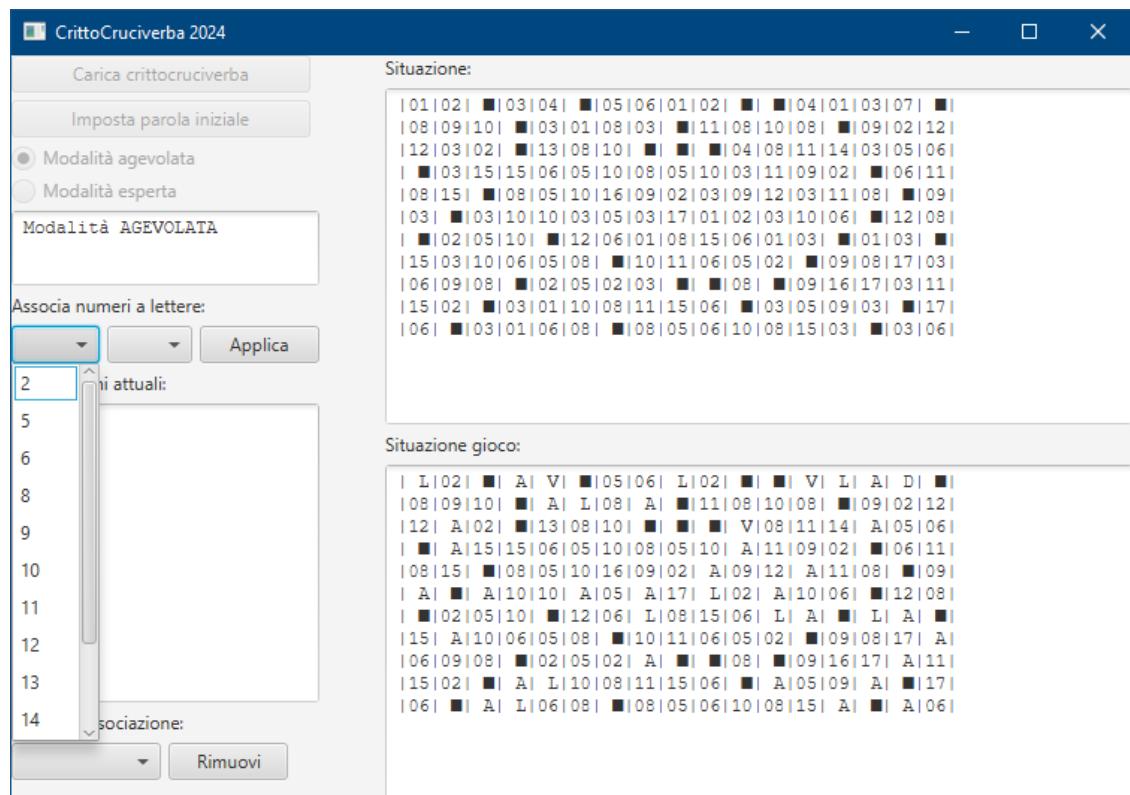


Fig. 4: la GUI dopo aver premuto il pulsante “Imposta parola iniziale”, che subito dopo si disabilita. Notare le combo numeri/lettere, popolate coi numeri/lettere ancora non svelati nello schema (“VLAD” usava i numeri 4,1,3,7, che infatti mancano nell’elenco della combo; lo stesso vale per le lettere A,D,L,V per l’elenco adiacente).

A questo punto si può iniziare a giocare: il solutore sceglie possibili associazioni e le imposta premendo il pulsante “Applica” (Fig. 5); in modalità agevolata, un’associazione errata verrà respinta (Fig. 6) mentre in modalità esperta sarà comunque accettata. Dopo ogni mossa, l’area sottostante mostra le associazioni in vigore, che sono anche selezionabili nella combo in basso per l’eventuale rimozione nel caso in cui ci accorga di aver sbagliato e si voglia tornare indietro a uno stadio precedente del gioco (cosa ovviamente sensata solo in modalità ESPERTA, dato che in agevolata è materialmente impedito inserire associazioni errate).

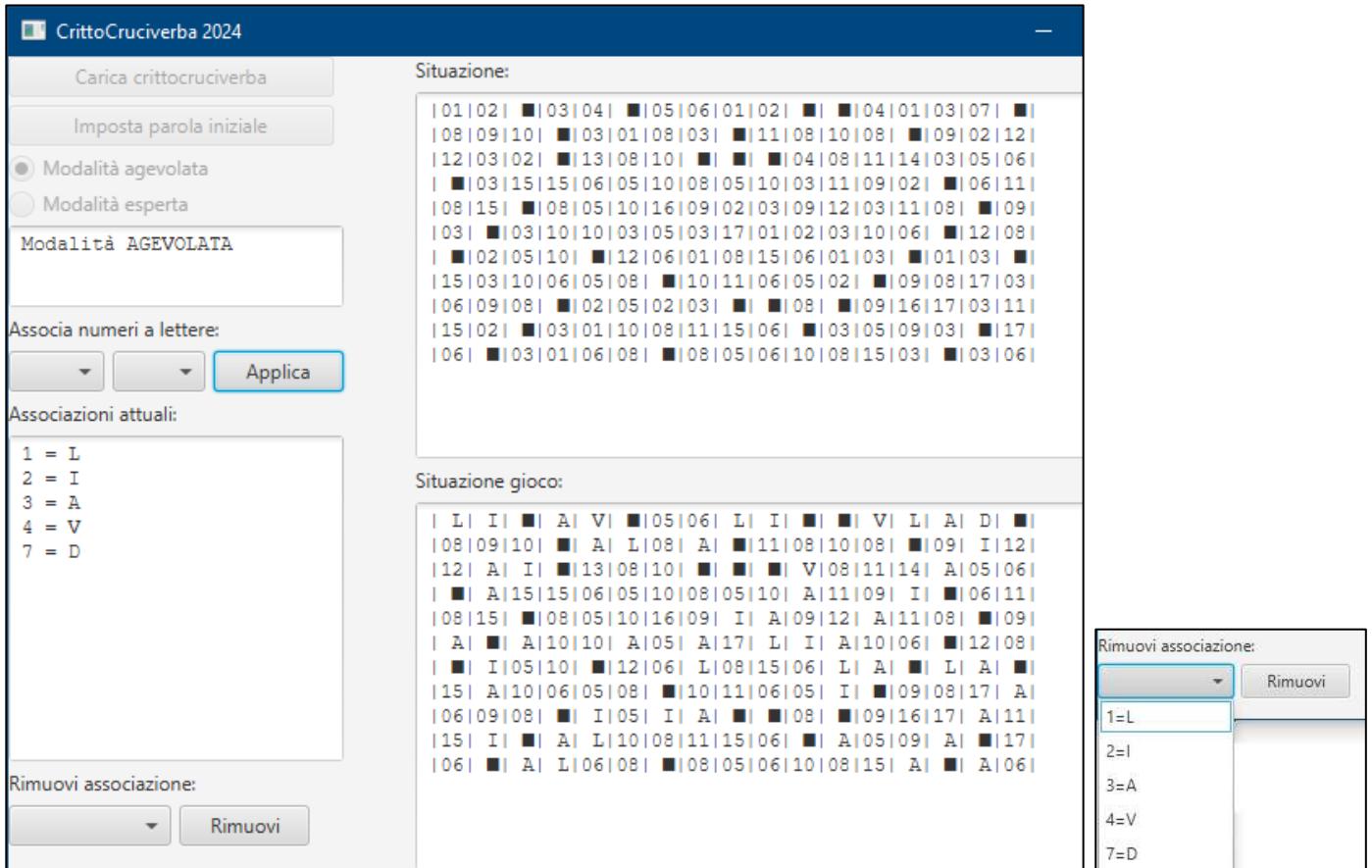


Fig. 5: la GUI dopo aver inserito l’associazione $2 = ‘I’$: notare che questi due valori vengono rimossi dai possibili item selezionabili nelle due combo singole, mentre compare l’associazione $2=I$ nell’elenco associazioni in basso.

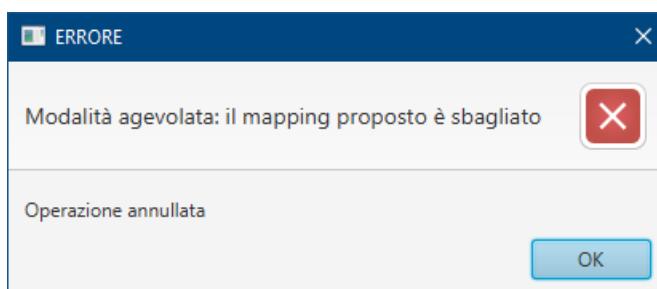


Fig. 6: messaggio d’errore nel caso si imposti una associazione errata in modalità AGEVOLATA.

Ad esempio, se si volesse rimuovere l’associazione $1=L$, basterebbe selezionarla nella combo e premere il pulsante “Rimuovi”: dallo schema numerato sparirebbero le L, sostituite da “1” come a uno stadio precedente.

Proseguendo, si giungerà infine ad aver utilizzato tutte le lettere e tutti i numeri, completando lo schema (Fig. 7)

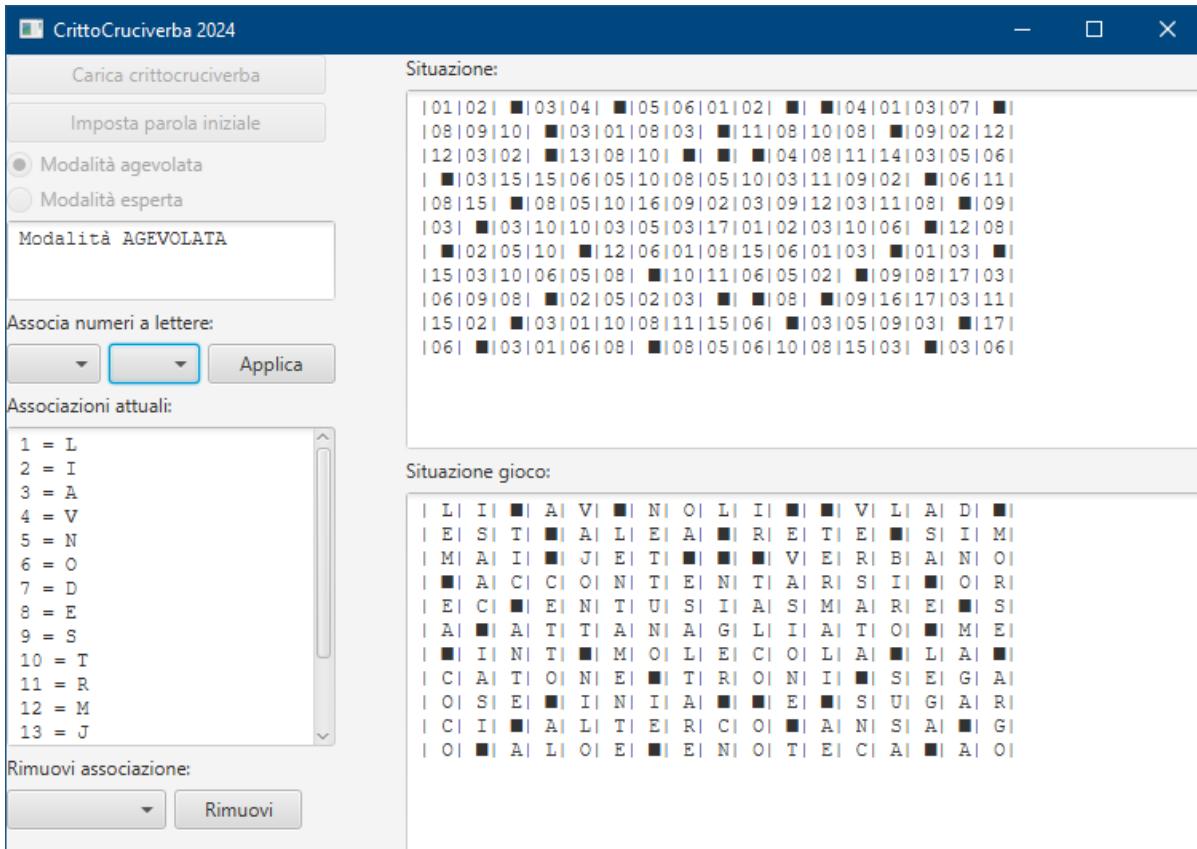


Fig. 7: lo schema completato.

Il MainPane è fornito *quasi completamente realizzato*: è presente tutta la parte strutturale, mentre rimangono da realizzare due gestori degli eventi (metodi privati *applicaAssociazione*, *rimuoviAssociazione*) e da garantire la mutua esclusione fra i due radiobutton.

In particolare:

- *parolaIniziale* imposta sul **Controller** la modalità prescelta, indi imposta le lettere iniziali, mostra lo schema numerato nell'apposita area di uscita, disabilita i pulsanti già utilizzati e popola le combo con le lettere e i numeri ancora disponibili
- *settaModalita* imposta sul **Controller** la modalità attualmente selezionata sui radiobutton
- *caricaSchema* attiva sul **Controller** la lettura da file e la successiva generazione dello schema numerato, abilitando/disabilitando i pulsanti come opportuno; in caso di errore di I/O o di formato, emette un'alert differenziato, con opportuna messaggistica
- *applicaAssociazione* recupera gli item selezionati nelle due combo (se uno o entrambi non sono selezionati, viene emesso opportuno msg d'errore e non si fa altro) e li usa per impostare un'associazione sul controller. Poi, ripopola le due combo col nuovo elenco aggiornato (che non comprende più i valori ora usati) ottenibile dai metodi del controller, aggiorna la visualizzazione dello schema numerato e infine aggiorna l'elenco delle associazioni correnti dell'apposita combo in basso; tale elenco viene anche stampato nell'area messaggi.
- *rimuoviAssociazione* agisce simmetricamente, recuperando dalla combo associazioni l'elemento selezionato (se esiste, altrimenti errore da mostrare tramite opportuna *alert*) usandolo per disassociare quei due elementi: di conseguenza deve essere aggiornato il contenuto delle due combo numeri e lettere, che devono ora includere nuovamente gli elementi dell'associazione cancellata, e della combo associazioni stessa, che avrà un elemento in meno.

Cose da ricordare

- salva costantemente il tuo lavoro: l'informatica a volte può essere “subdolamente ostile”...
- in particolare: se ora compila e stai per fare modifiche, salva la versione attuale (non si sa mai)

Checklist di consegna

- Hai fatto un **JAR eseguibile**, che contenga cioè l'indicazione del main?
- Hai controllato che **si compili e ci sia tutto?** [NB: non includere il PDF del testo]
- Hai **rinominato IL PROGETTO**, lo ZIP e il JAR esattamente come richiesto?
- Hai **chiamato** la cartella del progetto esattamente come richiesto?
- **Hai fatto un unico file ZIP (NON .7z, rar o altri formati) contenente l'intero progetto?**
In particolare, ti sei assicurato di aver incluso tutti i file .java (e non solo i .class)?
- **Hai consegnato DUE file distinti, ossia lo ZIP col progetto e il JAR eseguibile?**
- Su EOL, hai premuto il tasto “CONFERMA” per inviare il tuo elaborato?

ESAME DI FONDAMENTI DI INFORMATICA T-2 del 25/7/2024

Proff. E. Denti – R. Calegari – A. Molesini

Tempo a disposizione: 3h30

NOME PROGETTO ECLIPSE: CognomeNome-matricola (es. RossiMario-0000123456)

NOME CARTELLA PROGETTO: CognomeNome-matricola (es. RossiMario-0000123456)

NOME ZIP DA CONSEGNARE: CognomeNome-matricola.zip (es. RossiMario-0000123456.zip)

NOME JAR DA CONSEGNARE: CognomeNome-matricola.jar (es. RossiMario-0000123456.jar)

Si devono consegnare DUE FILE: l'intero progetto Eclipse e il JAR eseguibile

Compiti non compilabili, o che non passino almeno 2/3 dei test o siano lontani da 18/30 NON SARANNO CORRETTI e causeranno la verbalizzazione del giudizio “RESPINTO”.

La Repubblica di Dentinia ha deciso di adottare, per le proprie elezioni, il sistema elettorale australiano, basato sull'approccio di voto singolo trasferibile. Si chiede di sviluppare un'applicazione che consenta, in ogni collegio uninominale, di determinare il vincitore, applicando l'algoritmo di scrutinio sotto specificato.

DESCRIZIONE DEL DOMINIO DEL PROBLEMA

Nel sistema elettorale australiano per la House of Representatives il territorio è diviso in tanti collegi uninominali: ogni collegio elegge un solo rappresentante. A differenza però di altri sistemi, in questo caso si chiede all'elettore di numerare in ordine di preferenza tutti i candidati, da 1 (il più gradito) a N (il meno gradito): a lato un esempio di scheda.

Per essere eletto, un candidato deve avere la maggioranza assoluta ($50\%+1$) dei voti: la particolarità di questo sistema è che, laddove nessuno abbia tale maggioranza in prima battuta, anziché prevedere più turni o ballottaggi, si conteggiano le seconde, terze, quarte,... preferenze, agendo per fasi successive.

Ciò richiede uno specifico **algoritmo di scrutinio** a più fasi:

1. In prima battuta, si considerano solo le prime preferenze: se un candidato raggiunge la maggioranza assoluta viene eletto, altrimenti si passa alla fase successiva.
2. Se nessuno ha la maggioranza assoluta, si provvede a eliminare dalla competizione il candidato meno votato e trasferire le sue schede agli altri, guardando le seconde preferenze. Se, così facendo, qualcuno raggiunge la maggioranza assoluta, egli/ella viene eletto/a; altrimenti, si itera il procedimento, eliminando via via sempre il candidato meno votato e trasferendo agli altri le sue schede, guardando le seconde, terze, quarte preferenze, etc. Salvo il caso di parità, con N candidati il massimo numero di iterazioni è $N-2$: a quel punto rimangono sicuramente in gioco solo due candidati e quindi uno di essi raggiunge per forza la maggioranza assoluta, e viene eletto.
3. In caso di parità fra due o più candidati, la scelta su quale candidato eliminare viene effettuata secondo regole varie: qui si supporrà che avvenga a sorteggio (quindi, si potrà eliminare uno qualsiasi dei candidati a pari merito).

ESEMPIO: si supponga che vi siano 4 candidati (Maria, Ari, Joe, Lauren) e 100 elettori, e che il conteggio delle prime preferenze sia quello illustrato a lato. Poiché nessuno raggiunge la maggioranza assoluta di 51 voti, si procede eliminando la candidata meno votata (Lauren) e attribuendo le sue 6 schede agli altri, in base alle seconde preferenze indicate su tali schede. Se, supponiamo, 4 di queste sono per Ari e 1 a testa per gli altri due, la situazione è quella illustrata a fianco.

Poiché nessuno raggiunge ancora la maggioranza assoluta, occorre iterare il procedimento: si elimina il meno votato dei tre candidati rimasti (Joe) trasferendo i suoi 21 voti agli altri in base alle successive preferenze indicate in quelle 21 schede. Se 6 di queste sono per Maria e 15 per Ari, il risultato è quello a lato: stavolta un candidato (Ari) ha la maggioranza assoluta ed è eletto.

Da notare che, dei 21 voti di Joe, 20 erano suoi fin dall'inizio (prima preferenza), mentre 1 era un voto originariamente per Lauren, trasferito a Joe (seconda preferenza) quando Lauren è stata eliminata. Pertanto, eliminando Joe, quella scheda va ora assegnata al terzo candidato in ordine di preferenza; e così via. In generale, quando si elimina un candidato e si trasferiscono le sue schede ad altri, ogni scheda va assegnata al candidato successivo, in ordine di preferenza, a quello a cui la scheda è attualmente assegnata.



MARIA	ARI	JOE	LAUREN
39	35	20	6

MARIA	ARI	JOE	LAUREN
39	35	20	6
1	4	1	
40	39	21	

MARIA	ARI	JOE	LAUREN
40	39	21	
6	15		
46	54		

Una serie di file di testo, nel formato descritto più oltre, contiene le schede votate dagli elettori in vari collegi. L'applicazione lavorerà volta per volta su uno solo di essi, che verrà scelto dall'utente tramite l'interfaccia grafica.

Scopo dell'applicazione è consentire all'utente di a) scegliere il file e caricare le relative schede, b) effettuare i calcoli per determinare il vincitore e c) poter vedere, a richiesta, il dettaglio (log) delle operazioni di scrutinio, così da poter capire come si sia giunti alla determinazione del risultato. Le immagini in calce illustrano il look & feel della GUI.

TEMPO STIMATO PER SVOLGERE L'INTERO COMPITO: **2h15 – 3h**

PARTE 1 – Modello dei dati:	Punti 16	[TEMPO STIMATO: 100-120 minuti]
PARTE 2 – Persistenza:	Punti 10	[TEMPO STIMATO: 25-40 minuti]
PARTE 3 – Grafica:	Punti 4	[TEMPO STIMATO: 10-20 minuti]

NUMERO MINIMO DI TEST CON SUCCESSO PERCHÉ IL COMPITO SIA CORRETTO

Considerando solo Scrutinio e MySchedeReader	11 su 16
Considerandoli tutti:	22 su 32

JAVAFX – Parametri run configuration nei LAB

```
--module-path "C:\applicativi\moduli\javafx-sdk-21.0.2\lib"  
--add-modules javafx.controls
```

Cose da ricordare

- salva costantemente il tuo lavoro: l'informatica a volte può essere “subdolamente ostile”..
- in particolare: se ora compila e stai per fare modifiche, salva la versione attuale (non si sa mai)

Checklist di consegna

- Hai fatto un **JAR eseguibile**, che contenga cioè l'indicazione del main?
- Hai controllato che **si compili e ci sia tutto**? [NB: non includere il PDF del testo]
- Hai **rinominato** IL PROGETTO, lo ZIP e il JAR esattamente come richiesto?
- Hai **chiamato** la cartella del progetto esattamente come richiesto?
- **Hai fatto un unico file ZIP (NON .7z, rar o altri formati)** contenente **l'intero progetto?**
In particolare, ti sei assicurato di aver incluso tutti i file .java (e non solo i .class)?
- **Hai consegnato DUE file distinti, ossia lo ZIP col progetto e il JAR eseguibile?**
- Su EOL, hai **premuto** il tasto “CONFERMA” per inviare il tuo elaborato?

Parte 1 – Modello dei dati

(punti: 16)

Package: *australia.elections.model*

[TEMPO STIMATO: 100-120 minuti]

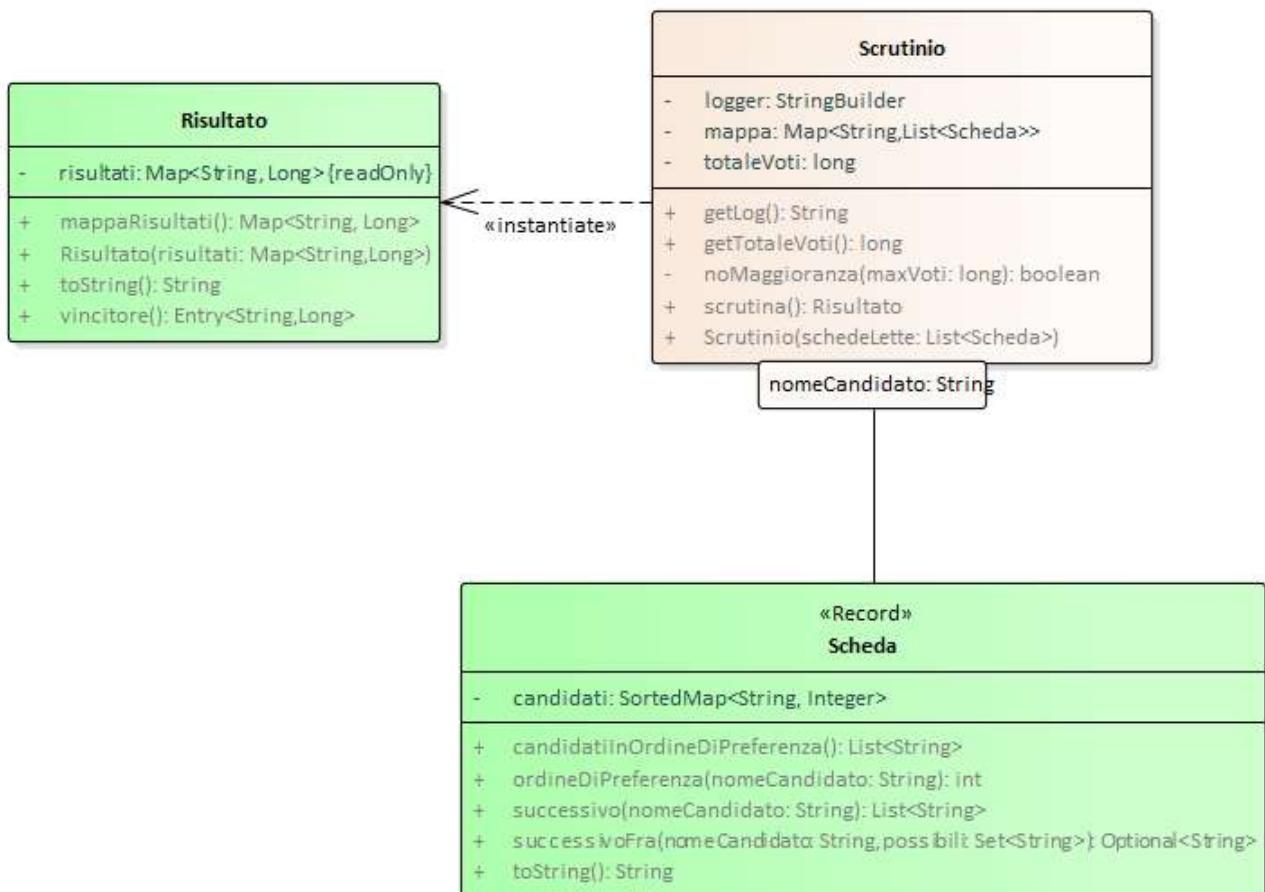
Conviene organizzare il modello dei dati su una struttura a mappa, che associa ogni candidato alla lista delle sue schede: la figura illustra come sarebbe la situazione iniziale nel caso dell'esempio sopra descritto.

Candidato	Lista schede
Maria	→ <i>Lista schede di Maria (39)</i>
Ari	→ <i>Lista schede di Ari (35)</i>
Joe	→ <i>Lista schede di Joe (20)</i>
Lauren	→ <i>Lista schede di Lauren (6)</i>

Ovviamente, la lunghezza di ogni lista indica il numero di voti attribuiti in un dato momento a ogni candidato.

Quando si procede a eliminare un candidato le sue schede devono quindi essere spostate nelle altre liste, in base alla preferenza successiva indicata su ogni scheda.

Le classi che esprimono ciò sono le seguenti:



- Il record **Scheda** (fornito) rappresenta la scheda elettorale, espressa da una mappa <stringa,intero> che associa a ogni candidato il corrispondente numero di preferenza ([a lato un esempio di scheda per Maria](#)).
 - Il costruttore verifica che la mappa ricevuta non sia null né vuota (lanciando rispettivamente **NullPointerException** o **IllegalArgumentException** in caso contrario) e che i numeri specificati siano tutti distinti e compresi fra 1 e il numero di candidati (anche in questo caso, lanciando **IllegalArgumentException** in caso di violazione)
 - il metodo **candidatiInOrdineDiPreferenza** restituisce la lista dei candidati in

Candidato	Ordine di preferenza
Maria	→ 1
Ari	→ 4
Joe	→ 2
Lauren	→ 3

ordine di preferenza, dal più gradito (n°1) al meno gradito (n° N)

- il metodo `ordineDiPreferenza(String nomeCandidato)` restituisce la posizione di quel candidato nell'ordine di preferenza, sotto forma di intero compreso fra 1 e N: lancia `NullPointerException` se il nome del candidato è nullo, o `IllegalArgumentException` se esso è vuoto o un tale candidato non esiste nella scheda
 - il metodo `successivo(String nomeCandidato)` restituisce il nome del candidato successivo a quello dato, secondo l'ordine di preferenza: se il candidato era ultimo e non ha successivi, viene restituito `Optional.empty` mentre se il nome del candidato è nullo o vuoto o inesistente vengono lanciate le stesse eccezioni del caso precedente
 - il metodo `successivoFra(String nomeCandidato, Set<String> possibili)` restituisce il nome del candidato successivo a quello dato *considerando come possibili solo i candidati citati nel secondo argomento* (ciò è utile nella 2° fase dell'algoritmo, per saltare i candidati già eliminati)
 - un'apposita `toString` emette una rappresentazione stampabile della scheda.
- La classe **Risultato** (fornita) rappresenta il risultato dello scrutinio, espresso come mappa <stringa, long> che associa a ogni candidato i voti ottenuti nell'ultimo turno di esecuzione dell'algoritmo.
- Il costruttore verifica che la mappa ricevuta non sia null né vuota (lanciando `NullPointerException` o `IllegalArgumentException` in caso contrario)
 - il metodo `vincitore` restituisce il nome del vincitore
 - un'apposita `toString` emette una rappresentazione stampabile del risultato
- La classe **Scrutinio** (*da completare nella parte algoritmica*) incorpora la logica di scrutinio: la gestione del log mantiene traccia dei principali passaggi svolti.
- il costruttore riceve la lista di schede su cui operare: verifica, al solito, che tale lista non sia nulla né vuota, lanciando nel caso le opportune eccezioni, e costruisce la tabella (mappa) iniziale contenente, per ogni candidato, la lista delle *schede in cui il candidato è la prima preferenza* (fase 1 dell'algoritmo); inizializza inoltre i valori del totale voti e lo status del logger interno (uno `StringBuilder`)
 - gli accessori `getTotaleVoti` e `getLog` restituiscono rispettivamente il totale dei voti espressi (su cui si basa a sua volta il metodo privato `noMaggioranza` per verificare il raggiungimento o meno della maggioranza assoluta) e la stringa di log che elenca le operazioni svolte
 - il metodo **scrutina (da realizzare)** incorpora la logica di scrutinio descritta nel Dominio del Problema. Pertanto, esso deve agire iterativamente come segue:
 - in primis estrarre il numero di voti del candidato più votato (*) e verificare se abbia o meno la maggioranza assoluta: se ce l'ha, il procedimento termina
 - se non ce l'ha, identificare il candidato meno votato (*), recuperare le sue schede, eliminarlo e ri-assegnare tali schede, una ad una, a un altro candidato, seguendo l'ordine di preferenza indicato su ogni singola scheda

	MARIA		ARI
46	54		

SUGGERIMENTO: a tal fine, conviene invocare su ogni scheda l'apposito metodo `successivoFra`, a cui passare come nomi possibili i soli candidati ancora in lizza: il risultato sarà il nome del candidato a cui assegnare tale scheda.

- (*) in caso di parità la scelta può essere effettuata secondo un criterio qualunque.

Al termine del ciclo, il metodo sintetizza e restituisce il **Risultato** dello scrutinio.

Durante le varie operazioni il metodo deve trascrivere sul log i momenti salienti, ovvero:

- a ogni iterazione, lo stato della mappa voti e del massimo dei voti
- ogni rimozione di candidato

- al termine, la mappa voti finale col risultato, che mostra il vincitore

Ciò consentirà alla fine di mostrare sulla GUI come si è giunti alla determinazione del risultato
 (vedere figure in calce).

Parte 2 – Persistenza

(punti: 10)

Package: *australia.elections.persistence*

[TEMPO STIMATO: 25-40 minuti]

Vari file di testo elencano le schede votate in diversi collegi: il reader dovrà però preoccuparsi di leggere un singolo file, che sarà scelto dall'utente tramite la GUI.

Ogni file elenca, una per riga, le schede votate in quel collegio, nel seguente formato:

- tutte le righe hanno *eguale struttura* e devono contenere lo *stesso numero di elementi* (separati da virgolette) e gli stessi nomi di candidati
- tutte le righe **alternano un nome** (che non può essere costituito di soli caratteri numerici) e un numero (che è **costituito ovviamente di soli caratteri numerici**)

```
Maria, 1, Ari, 3, Joe, 2, Lauren, 4, Juliet, 5, Fred, 6
Maria, 1, Ari, 3, Joe, 2, Lauren, 4, Juliet, 5, Fred, 6
Maria, 1, Ari, 2, Joe, 3, Lauren, 4, Juliet, 6, Fred, 5
Maria, 1, Ari, 6, Joe, 3, Lauren, 4, Juliet, 2, Fred, 5
Maria, 1, Ari, 2, Joe, 4, Lauren, 6, Juliet, 3, Fred, 5
Maria, 1, Ari, 6, Joe, 2, Lauren, 3, Juliet, 4, Fred, 5
```

SEMANTICA:

a) L'interfaccia **SchedeReader** (fornita) dichiara:

- il metodo *leggiSchede*, che legge dal **Reader** (già aperto) fornito i dati necessari e restituisce una lista di **Scheda** perfettamente configurato; esso lancia **BadFormatException** con opportuno messaggio d'errore in caso di problemi nel formato del file o **IOException** in caso di altri problemi di I/O

b) La classe **MySchedeReader** (da completare) implementa **SchedeReader** secondo le specifiche sopra descritte.

- non c'è alcun costruttore
- Il metodo *leggiSchede* legge le righe, memorizzando numero di elementi e nomi della prima riga per usarli poi come confronto per le righe successive, e le elabora estraendo i dati per costruire le singole schede, popolando quindi la lista da restituire.



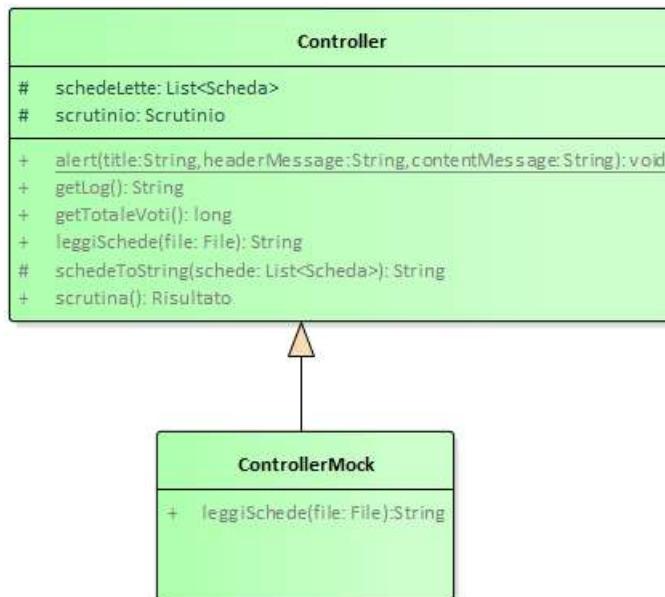
Parte 3

(punti: 4)

Package: australia.elections.controller

(punti 0)

Il Controller funge semplicemente da front-end verso l'analizzatore, ed è quindi è organizzato come segue:



SEMANTICA:

La classe **Controller** (fornita) non ha costruttore e definisce i seguenti metodi:

- **leggiSchede** legge, tramite il reader, il **File** fornito e memorizza la lista di schede così ottenuta nel proprio stato interno; quindi, istanzia e configura lo **Scrutinio**, che effettuerà poi i calcoli. Restituisce intanto le schede lette, sotto forma di stringa pronta per la visualizzazione o stampa.
- **scrutina** attiva lo scrutinio e ne restituisce il **Risultato**
- **getTotaleVoti** e **getLog** restituiscono le stesse informazioni restituite da **Scrutinio**.

Il metodo statico **alert**, utilizzabile dal **MainPane** quando è attiva la grafica, consente di far comparire all'utente una finestra di dialogo che mostri il messaggio d'errore specificato.

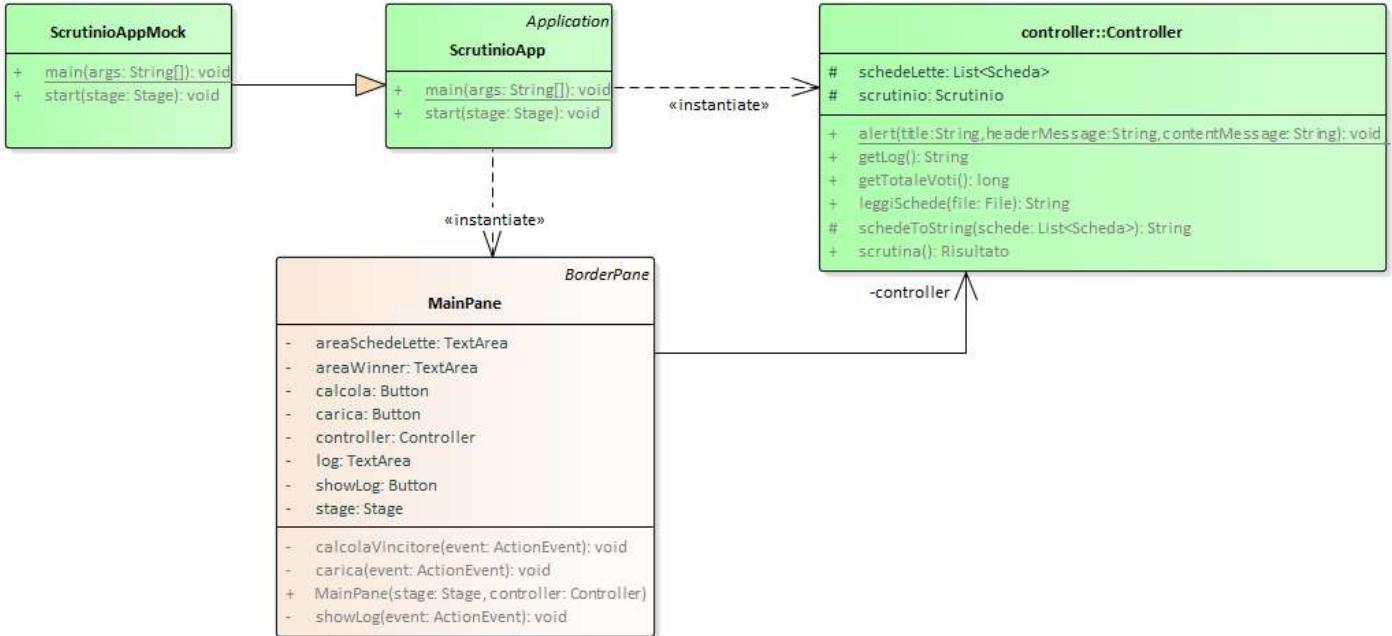
La sottoclassificazione **ControllerMock** (fornita) è utilizzata esclusivamente da **ScrutinioAppMock** (fornita, vedere sotto) per simulare la presenza dei dati letti da file: pertanto, non deve essere considerata in questo compito.

Package: australian.elections.ui

[TEMPO STIMATO: 10-20 minuti] (punti 4)

La classe **ScrutinioApp** (fornita) costituisce l'applicazione JavaFX che si occupa di aprire i file, creare il controller e incorporare il **MainPane**. Per consentire di collaudare la GUI anche in assenza / in caso di malfunzionamento della parte di persistenza, è possibile avviare l'applicazione mediante la classe **ScrutinioAppMock** (che si avvale del controller ausiliario **ControllerMock**).

L'interfaccia utente è illustrata nelle figure seguenti e segue il modello sotto illustrato:



- a sinistra vi sono i pulsanti “Carica file” e “calcola”, ognuno seguito dalla relativa mini-area di testo, rispettivamente per la visualizzazione delle schede caricate e per il risultato del calcolo (ossia, il vincitore);
- a destra vi è invece il solo pulsante “Show log”, seguito dall’area di testo di maggiore dimensione destinata a mostrare i dettagli (log) delle operazioni di scrutinio.

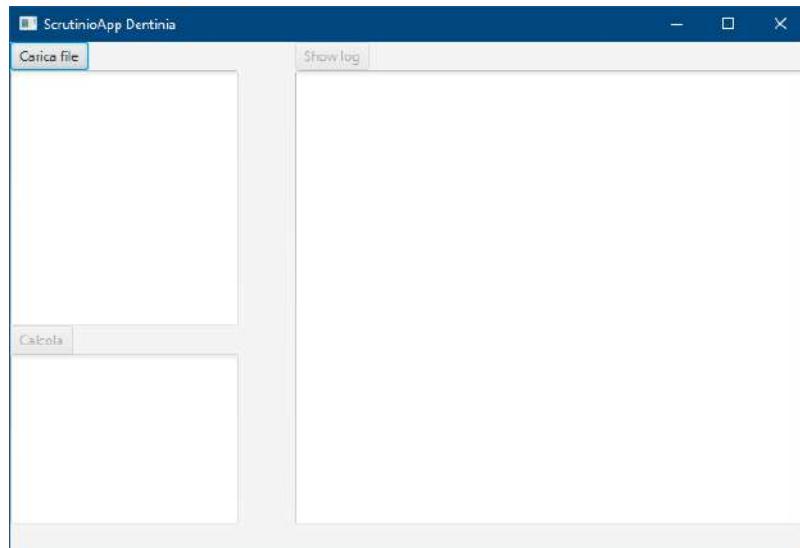


Fig. 1: vista generale: a sinistra i controlli per il caricamento del file e la visualizzazione delle schede (in alto), per il calcolo e la visualizzazione del risultato (in basso); a destra, i controlli per la visualizzazione dei dettagli (log) delle operazioni di scrutinio. Da notare che inizialmente il solo pulsante abilitato è quello relativo al caricamento del file.

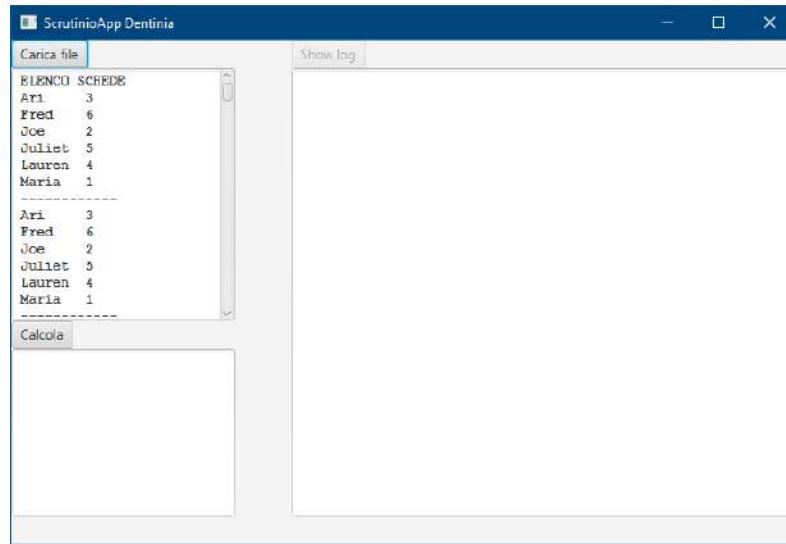


Fig. 2: la GUI dopo aver premuto il pulsante “Carica file”: notare che il pulsante “Calcola” è ora abilitato.

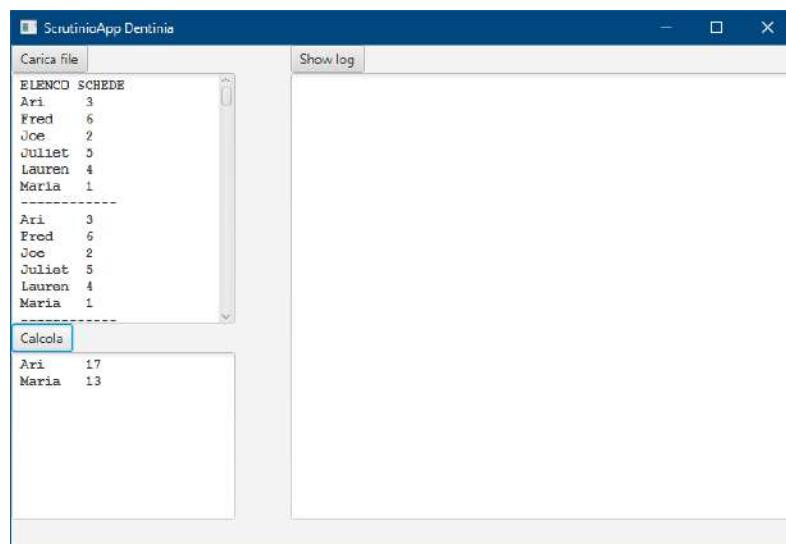


Fig. 3: la GUI dopo aver premuto il pulsante “Calcola” : notare che il pulsante “Show log” è ora abilitato

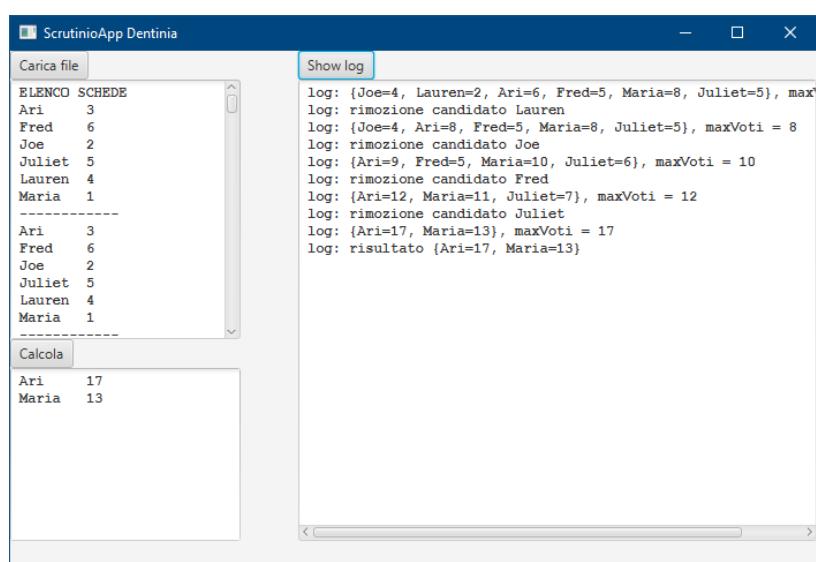


Fig. 4: la GUI dopo aver premuto il pulsante “Show log”. Per analizzare un altro file si può ripartire da capo riprendendo “Carica file”.

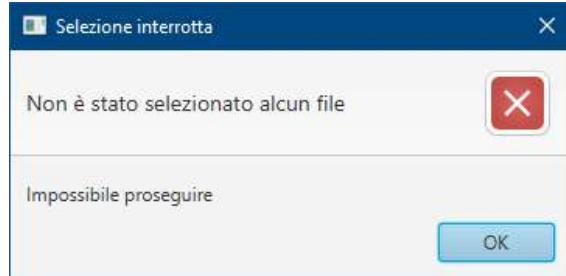


Fig. 5: messaggio d'errore nel caso si prema “Annulla” nella finestra (chooser) di scelta file, che compare premendo il pulsante “Carica file”.

Il MainPane è fornito *parzialmente realizzato*: è presente tutta la parte strutturale, mentre rimangono da realizzare due gestori degli eventi (metodi privati *calcolaVincitore*, *showLog*)

In particolare:

- *calcolaVincitore* deve chiedere al **Controller** di effettuare lo scrutinio, abilitare il pulsante “Show log” e mostrare il vincitore nell'apposita area di testo in basso a sinistra.
- *showLog* deve chiedere al **Controller** i dettagli del log e mostrarli nell'area di testo a destra.

Cose da ricordare

- salva costantemente il tuo lavoro: l'informatica a volte può essere “subdolamente ostile”...
- in particolare: se ora compila e stai per fare modifiche, salva la versione attuale (non si sa mai)

Checklist di consegna

- Hai fatto un **JAR eseguibile**, che contenga cioè l'indicazione del main?
- Hai controllato che **si compili e ci sia tutto**? [NB: non includere il PDF del testo]
- Hai **rinominato IL PROGETTO**, lo ZIP e il JAR esattamente come richiesto?
- Hai **chiamato** la cartella del progetto esattamente come richiesto?
- **Hai fatto un unico file ZIP (NON .7z, rar o altri formati) contenente l'intero progetto?**
In particolare, ti sei assicurato di aver incluso tutti i file .java (e non solo i .class)?
- **Hai consegnato DUE file distinti, ossia lo ZIP col progetto e il JAR eseguibile?**
- Su EOL, hai **premuto** il tasto “CONFERMA” per inviare il tuo elaborato?

ESAME DI FONDAMENTI DI INFORMATICA T-2 del 3/7/2024

Proff. E. Denti – R. Calegari – A. Molesini

Tempo a disposizione: 3h30

NOME PROGETTO ECLIPSE: CognomeNome-matricola (es. RossiMario-0000123456)

NOME CARTELLA PROGETTO: CognomeNome-matricola (es. RossiMario-0000123456)

NOME ZIP DA CONSEGNARE: CognomeNome-matricola.zip (es. RossiMario-0000123456.zip)

NOME JAR DA CONSEGNARE: CognomeNome-matricola.jar (es. RossiMario-0000123456.jar)

Si devono consegnare DUE FILE: l'intero progetto Eclipse e il JAR eseguibile

Si ricorda che compiti *non compilabili*, o che *non passino almeno 2/3 dei test* o siano *palesemente lontani da 18/30* NON SARANNO CORRETTI e causeranno la verbalizzazione del giudizio “RESPINTO**”.**

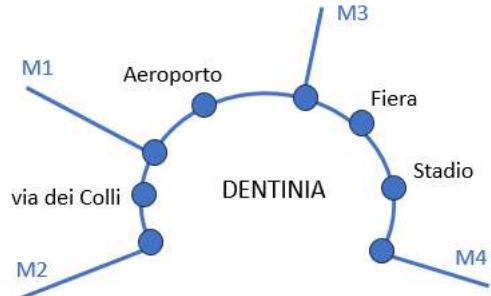
La rete autostradale di Dentinia è articolata in *quattro autostrade a pedaggio* che originano a raggiera dalla *tangenziale*, che avvolge buona parte la città (v. figura), senza comunque richiudersi su sé stessa. La tangenziale è utilizzata 1) dal traffico cittadino, che entra/esce dai caselli della tangenziale senza però utilizzare le autostrade; 2) dal traffico di puro transito fra un’autostrada e l’altra, che la usa solo come connessione fra tronchi autostradali; 3) dal traffico di entrata/uscita in/dalla città da/verso le autostrade, che esce/entra da un casello della tangenziale da/verso un’autostrada, dando luogo a un percorso misto.

Il comune di Dentinia ha ottenuto dal gestore autostradale che la tangenziale sia *gratuita per l’uso urbano* (caso 1) ma *a pagamento* per l’attraversamento (caso 2) e, entro i limiti sotto discussi, per l’uso ibrido (caso 3).

È quindi stato richiesto di sviluppare un’applicazione che consenta di determinare il percorso fra una qualsiasi coppia di caselli della rete, calcolandone il chilometraggio e il costo.

DESCRIZIONE DEL DOMINIO DEL PROBLEMA

Come illustrato a lato, la rete autostradale di Dentinia è costituita da quattro autostrade (M1, M2, M3, M4): esse convergono tutte sulla tangenziale, che avvolge la città da sud-ovest a sud-est. Sulla tangenziale sono presenti quattro caselli urbani (via dei Colli, Aeroporto, Fiera, Stadio) oltre, ovviamente, ai quattro allacciamenti per le quattro autostrade. Ogni autostrada prevede ovviamente una serie di caselli (non mostrati in figura).



Tutte le autostrade sono a pedaggio, mentre per la tangenziale vige un sistema misto, discusso in dettaglio sotto; in ogni caso, il pedaggio si calcola moltiplicando i *chilometri utili* per il costo chilometrico unitario (noto).

Per quanto riguarda la tangenziale, valgono le seguenti regole:

1. L’uso esclusivamente urbano è sempre gratuito: quindi, entrando/uscendo dai quattro caselli urbani, i km utili ai fini del pedaggio sono sempre zero.
2. L’uso come pura connessione fra autostrade è sempre a pagamento: in tal caso, quindi, i km compresi fra i due allacciamenti autostradali sono sempre tutti utili ai fini del pedaggio.
3. Nel caso ibrido i km percorsi in tangenziale sono a pagamento solo per i percorsi in cui la quota autostradale superi i 40 km: ciò consente di non penalizzare i pendolari che vivono entro i 40 km dalla città, per i quali il tratto in tangenziale resta quindi gratuito.

ESEMPI: entrando a via dei Colli e uscendo a Stadio, non si paga nulla; se, invece, da Fiera si va verso M1, il tratto in tangenziale fino all’allacciamento con la M1 è a pagamento solo se i km su M1 sono più di 40, altrimenti è gratuito; infine, se si proviene da M2 e si va verso M3, il tratto di tangenziale fra i due allacciamenti è sempre a pagamento.

Il file di testo [AutostradeDentinia.txt](#), nel formato descritto più oltre, contiene la descrizione delle varie autostrade e della tangenziale. Per ipotesi, nella rete autostradale esiste sempre una e una sola “Tangenziale”.

TEMPO STIMATO PER SVOLGERE L'INTERO COMPITO: **2h15 – 3h**

PARTE 1 – Modello dei dati:	Punti 13	[TEMPO STIMATO: 70-90 minuti]
PARTE 2 – Persistenza:	Punti 11	[TEMPO STIMATO: 45-60 minuti]
PARTE 3 – Grafica:	Punti 6	[TEMPO STIMATO: 20-30 minuti]

NUMERO MINIMO DI TEST CON SUCCESSO PERCHÉ IL COMPITO SIA CORRETTO

Considerando solo Percorso, MyPlanner e MyAutotradeReader	20 su 29
Considerandoli tutti:	32 su 48

JAVAFX – Parametri run configuration nei LAB

```
--module-path "C:\applicativi\moduli\javafx-sdk-21.0.2\lib"  
--add-modules javafx.controls
```

Cose da ricordare

- salva costantemente il tuo lavoro: l'informatica a volte può essere “subdolamente ostile”..
- in particolare: se ora compila e stai per fare modifiche, salva la versione attuale (non si sa mai)

Checklist di consegna

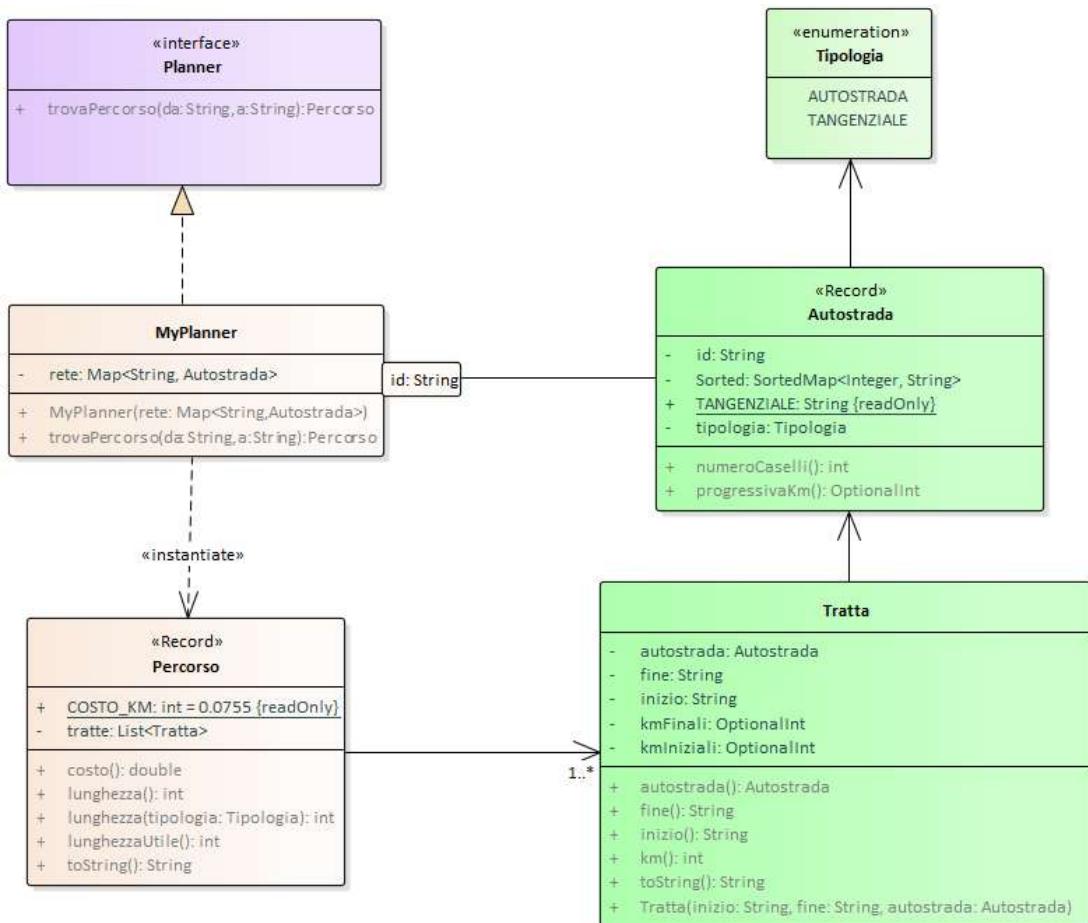
- Hai fatto un **JAR eseguibile**, che contenga cioè l'indicazione del main?
- Hai controllato che **si compili e ci sia tutto**? [NB: non includere il PDF del testo]
- Hai **rinominato IL PROGETTO**, lo ZIP e il JAR esattamente come richiesto?
- Hai **chiamato** la cartella del progetto esattamente come richiesto?
- **Hai fatto un unico file ZIP (NON .7z, rar o altri formati) contenente l'intero progetto?**
In particolare, ti sei assicurato di aver incluso tutti i file .java (e non solo i .class)?
- **Hai consegnato DUE file distinti, ossia lo ZIP col progetto e il JAR eseguibile?**
- Su EOL, hai premuto il tasto “CONFERMA” per inviare il tuo elaborato?

Parte 1 – Modello dei dati

(punti: 13)

Package: tangenziale.model

[TEMPO STIMATO: 70-90 minuti]



SEMANTICA:

- L'enumerativo **Tipologia** (fornito) definisce semplicemente le due costanti `AUTOSTRADA` e `TANGENZIALE`
- Il record **Autostrada** (fornito) rappresenta un'autostrada (o tangenziale), caratterizzata da **Tipologia**, identificativo univoco (stringa) e una mappa `<intero, stringa>` che elenca i caselli e gli allacciamenti, associando ogni casello (identificato dal suo nome) alla progressiva chilometrica lungo l'autostrada stessa.
 - Il costruttore verifica che la mappa ricevuta non sia null né vuota (lanciando rispettivamente `NullPointerException` o `IllegalArgumentException` in caso contrario) e che non ci siano né due caselli omonimi, né due caselli definiti alla stessa progressiva chilometrica (lanciando anche in questo caso `IllegalArgumentException` in caso di violazione)
 - il metodo `numeroCaselli` restituisce il numero dei caselli/allacciamenti previsti in questa autostrada (cioè il numero di righe della mappa)
 - il metodo `getProgressivaKm(String nomeCasello)` restituisce, sotto forma di `OptionalInt`, la progressiva chilometrica del casello o allacciamento specificato: se esso non esiste, restituisce `OptionalInt.empty`
 - è inoltre definita la costante `TANGENZIALE`, che esprime il nome della (unica) tangenziale
- La classe **Tratta** (fornita) rappresenta una tratta autostradale su una specifica autostrada o tangenziale, compresa fra due caselli di inizio e fine (due stringhe).
 - Il costruttore verifica che gli argomenti non sia null né vuoti (lanciando rispettivamente `NullPointerException` o `IllegalArgumentException` in caso contrario)
 - una serie di accessori consente di recuperare gli argomenti passati al costruttore

- il metodo *km* restituisce la lunghezza della tratta
 - un'apposita *toString* emette una rappresentazione stampabile della tratta stessa
- La classe **Percorso** (*da completare nella parte algoritmica*) esprime l'idea di percorso come sequenza di tratte e ne cattura le proprietà salienti. In particolare:
 - è definita la **costante pubblica COSTO_KM** che esprime il costo al chilometro del pedaggio sull'intera rete autostradale
 - il costruttore riceve la lista di tratte e verifica che non sia nulla, emettendo in caso contrario la classica **NullPointerException**; parimenti non è valido un percorso con più di tre tratte o che abbia due tratte sulla stessa autostrada: in questi casi viene lanciata **IllegalArgumentException**. È invece lecito il caso di lista vuota, che corrisponde a un percorso non trovato.
 - il metodo *lunghezza* restituisce la lunghezza totale del percorso
 - il metodo *lunghezza(Tipologia)* restituisce la lunghezza delle sole tratte della tipologia specificata
 - il metodo *costo* restituisce il costo (pedaggio) del percorso, ottenuto moltiplicando il costo chilometrico per i km utili
 - il metodo ***lunghezzaUtile (da realizzare)*** incorpora la logica di calcolo dei km utili ai fini del pedaggio, così come definita dal Dominio del Problema:
 1. l'uso urbano della Tangenziale è sempre gratuito: quindi, entrando/uscendo dai suoi quattro caselli urbani, i km utili ai fini del pedaggio sono sempre zero;
 2. l'uso della Tangenziale come pura connessione fra autostrade è sempre a pagamento;
 3. nel caso ibrido, i km percorsi in tangenziale sono a pagamento solo per i percorsi in cui la quota autostradale superi i 40 km.
- L'interfaccia **Planner** (fornita) rappresenta la vista esterna del pianificatore percorsi: essa dichiara il metodo *trovaPercorso(String da, String a)*, che calcola e restituisce il **Percorso** fra i due caselli o allacciamenti specificati secondo le specifiche e la tariffazione descritte nel Dominio del Problema. Si ricorda che:
 - se i caselli di entrata e di uscita sono sulla stessa autostrada (o tangenziale), il percorso è diretto (cioè, è composto da un'unica tratta);
 - se i caselli di entrata e di uscita sono su due autostrade diverse, il percorso è indiretto e comprende tre tratte, di cui senz'altro anche un tratto intermedio in Tangenziale;
 - se i caselli di entrata e di uscita sono uno su un'autostrada e l'altro sulla Tangenziale (o viceversa), il percorso è indiretto e composto di due tratte, una delle quali è senz'altro in Tangenziale.
- La classe **MyPlanner** (*da realizzare*) implementa il pianificatore sopra descritto. In particolare:
 - il costruttore riceve la mappa della rete autostradale, che non può essere né nulla, né vuota (altrimenti, viene lanciata rispettivamente **NullPointerException** o **IllegalArgumentException**) e deve necessariamente contenere almeno la tangenziale (altrimenti, **IllegalArgumentException**)
 - il metodo *trovaPercorso(String da, String a)* deve verificare accuratamente le precondizioni:
 - gli argomenti non possono essere nulli, altrimenti **NullPointerException**
 - i caselli dati devono esistere nella rete autostradale, altrimenti **IllegalArgumentException**

SEGUE PAGINA SUCCESSIVA

Parte 2 – Persistenza

(punti: 11)

Package: tangenziale.persistence

[TEMPO STIMATO: 45-60 minuti]

Il file di testo `AutotradeDentinia.txt` contiene la descrizione delle varie autostrade e della tangenziale. Per ipotesi, deve esistere sempre una e una sola “Tangenziale”. Il formato è descritto di seguito e illustrato nell'esempio in calce:

- Ogni autostrada (inclusa la tangenziale) è descritta da un blocco di righe, che inizia con una riga contenente solo il nome dell'autostrada e termina con una riga contenente la sola parola “END”.
- Le righe intermedie descrivono i caselli e gli allacciamenti di tale autostrada, e prevedono prima la progressiva chilometrica (un intero), poi, separato da una o più tabulazione, il nome del casello o allacciamento (che può contenere spazi).

```
M3
45      Petruvia Nord
40      Petruvia Sud
25      Aldino
1       Dentinia Ippodromo
0       Tangenziale
END

M4
0       Tangenziale
24     Castelbello
37     Molla
50     Forte
77     Ripalta
END

...
Tangenziale
0       M2
4       via dei colli
9       M1
13      Aeroporto
19      M3
23      Fiera
26      Stadio
32      M4
END
```

SEMANTICA:

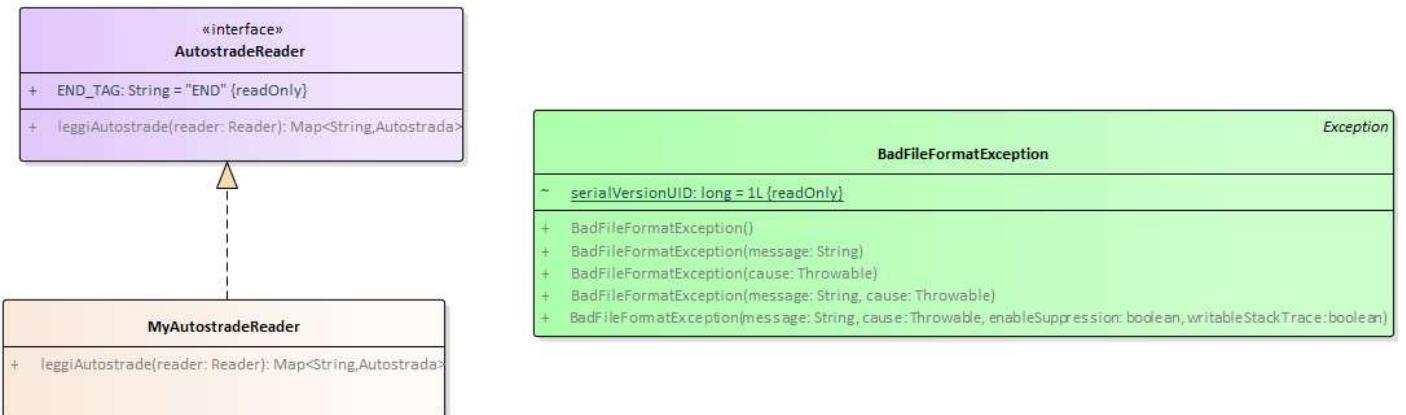
a) L'interfaccia `AutotradeReader` (fornita) dichiara:

- il metodo `leggiAutostrade`, che legge dal `Reader` (già aperto) fornito i dati necessari e restituisce la mappa della rete autostradale, di tipo `<String, Autostrada>`, che associa ogni `Autostrada` al suo identificativo univoc, che funge da chiave; esso deve lanciare `NullPointerException` in caso di reader nullo, `BadFormatException` con opportuno messaggio d'errore in caso di problemi di formato, o `IOException` in caso di problemi di I/O
- la costante stringa `END_TAG` che definisce la stringa corrispondente a "END" nel formato del file

b) La classe `MyAutotradeReader` (da completare) implementa `AutotradeReader`:

- non c'è alcun costruttore
- Il metodo `leggiAutostrade` legge le autostrade e le accumula via via nella mappa, che poi restituirà; esso deve effettuare due fondamentali controlli di consistenza:
 - i. Uno su ogni singola autostrada, verificando che in ogni autostrada letta non vi siano due caselli o allacciamenti alla stessa progressiva chilometrica

- ii. Un altro al termine della lettura, verificando che nella rete autostradale nel suo complesso ci sia sempre una e una sola tangenziale.



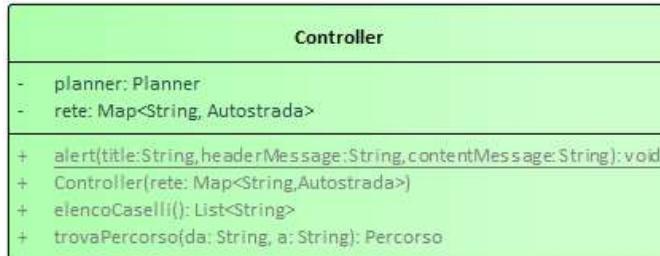
Parte 3

(punti: 6)

Package: tangenziale.controller

(punti 0)

Il Controller funge semplicemente da front-end verso l'analizzatore, ed è quindi è organizzato come segue:



SEMANTICA:

La classe **Controller** (fornita) funge da ponte fra grafica e model e definisce i seguenti metodi:

- Il costruttore riceve la mappa contenente la rete autostradale: verifica che essa non sia nulla né vuota, lanciando le opportune eccezioni in caso contrario; se tutto è ok, istanzia e mantiene al proprio interno il **Planner** che calcolerà i percorsi e i costi
- **trovaPercorso(String da, String a)** che banalmente delega all'omonimo metodo del **Planner**
- **elencoCaselli** che restituisce la lista dei soli caselli, esclusi gli allacciamenti fra autostrade e tangenziale

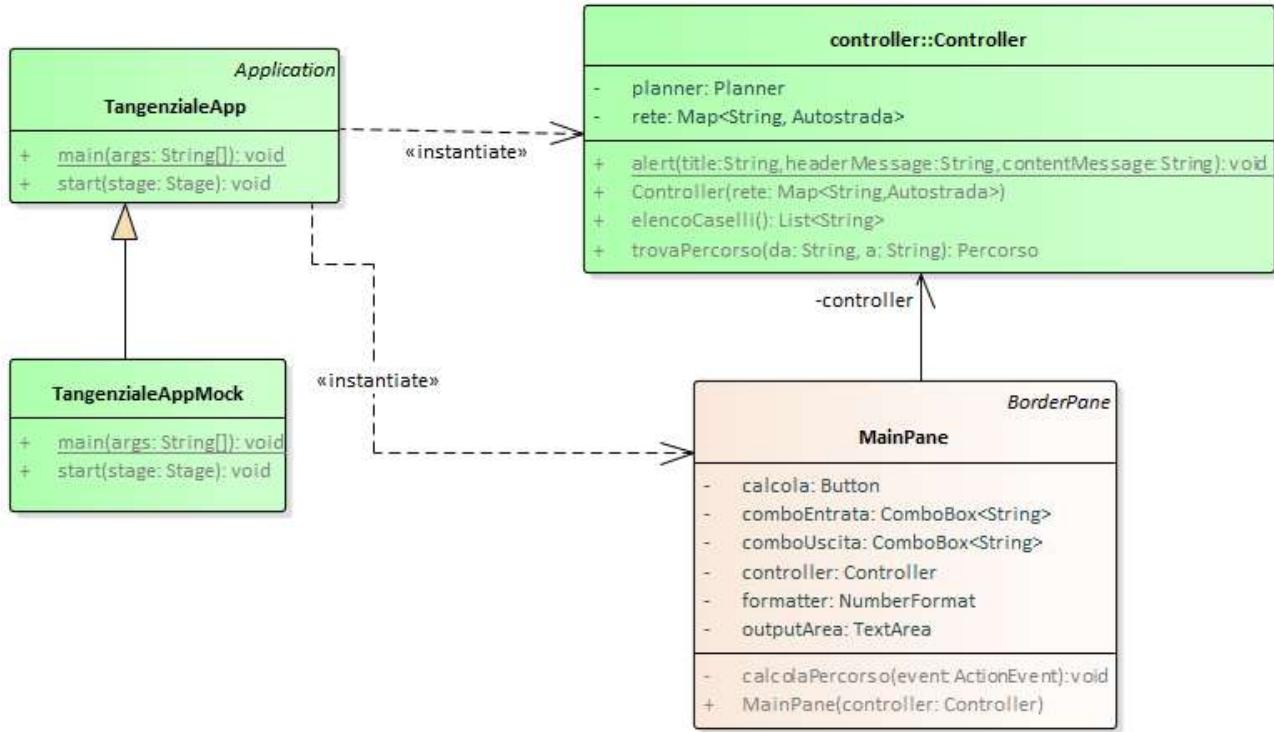
Il metodo statico **alert**, utilizzabile dal **MainPane** quando è attiva la grafica, consente di far comparire all'utente una finestra di dialogo che mostri il messaggio d'errore specificato.

Package: tangenziale.ui

[TEMPO STIMATO: 20-30 minuti] (punti 6)

La classe **TangenzialeApp** (fornita) costituisce l'applicazione JavaFX che si occupa di aprire i file, creare il controller e incorporare il **MainPane**. Per consentire di collaudare la GUI anche in assenza / in caso di malfunzionamento della parte di persistenza, è possibile avviare l'applicazione mediante la classe TangenzialeAppMock

L'interfaccia utente è illustrata nelle figure seguenti e segue il modello sotto illustrato:



- in alto, due combo popolate con l'elenco dei caselli consentono di scegliere i caselli di entrata e di uscita
- al centro, il pulsante “Calcola percorso” attiva il planner: se uno o entrambi i caselli non sono selezionati, dev’essere mostrato (tramite il metodo *alert* del **Controller**) un opportuno messaggio d’errore (Fig. 3)
- in basso, un’area di testo mostra il risultato, costituito dal dettaglio del percorso e dal relativo costo (pedaggio), formattato secondo lo standard italiano.

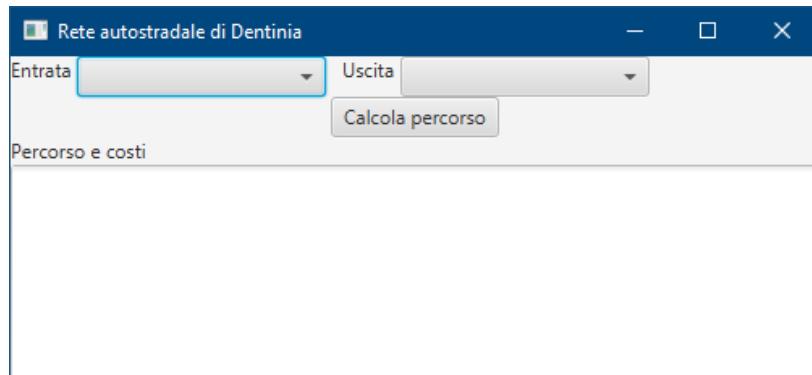


Fig. 1: vista generale della GUI: sopra le combo per scegliere i caselli di entrata e uscita, in mezzo il pulsante di attivazione del planner, in basso l’area destinata a mostrare i risultati.

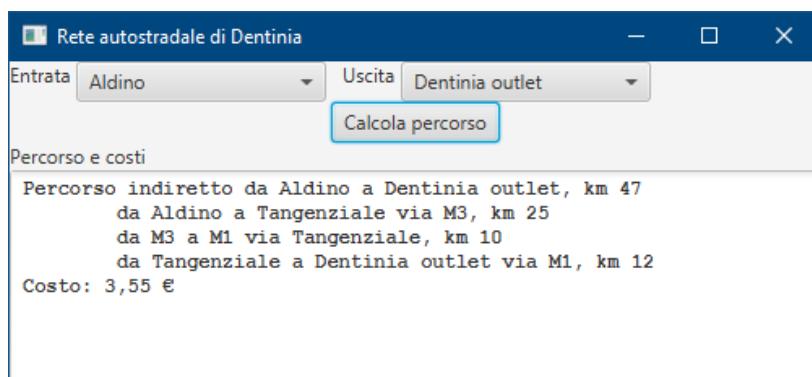


Fig. 2: la GUI dopo aver scelto due caselli e aver premuto il pulsante “Calcola percorso”.

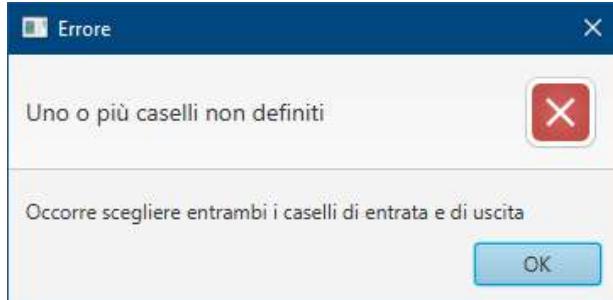


Fig. 3: il messaggio d'errore che compare se si preme il pulsante senza aver prima selezionato uno o entrambi i caselli di entrata e uscita.

Il MainPane è fornito *parzialmente realizzato*. Rimangono da realizzare le seguenti parti:

- il popolamento iniziale delle due combo
- l'aggancio del pulsante al gestore degli eventi (metodo privato `calcolaPercorso`)
- il gestore dell'evento `calcolaPercorso`, che deve innanzitutto verificare che entrambi i caselli di entrata e uscita siano selezionati: se così non è, deve emettere il messaggio d'errore sopra descritto e terminare senza fare nulla; se invece tutto è regolare, deve far calcolare il percorso al **Controller** ed emettere sull'area di testo il messaggio con il risultato, formattato come nelle figure, completo dell'indicazione di costo formattata secondo lo standard italiano

Cose da ricordare

- salva costantemente il tuo lavoro: l'informatica a volte può essere "subdolamente ostile"...
- in particolare: se ora compila e stai per fare modifiche, salva la versione attuale (non si sa mai)

Checklist di consegna

- Hai fatto un **JAR eseguibile**, che contenga cioè l'indicazione del main?
- Hai controllato che **si compili e ci sia tutto**? [NB: non includere il PDF del testo]
- Hai **rinominato IL PROGETTO**, lo ZIP e il JAR esattamente come richiesto?
- Hai **chiamato** la cartella del progetto esattamente come richiesto?
- **Hai fatto un unico file ZIP (NON .7z, rar o altri formati)** contenente l'intero progetto?
In particolare, ti sei assicurato di aver incluso tutti i file .java (e non solo i .class)?
- **Hai consegnato DUE file distinti, ossia lo ZIP col progetto e il JAR eseguibile?**
- Su EOL, hai premuto il tasto "CONFERMA" per inviare il tuo elaborato?

ESAME DI FONDAMENTI DI INFORMATICA T-2 del 14/2/2024

Proff. E. Denti – R. Calegari – A. Molesini

Tempo a disposizione: 3h30

NOME PROGETTO ECLIPSE: CognomeNome-matricola (es. RossiMario-0000123456)

NOME CARTELLA PROGETTO: CognomeNome-matricola (es. RossiMario-0000123456)

NOME ZIP DA CONSEGNARE: CognomeNome-matricola.zip (es. RossiMario-0000123456.zip)

NOME JAR DA CONSEGNARE: CognomeNome-matricola.jar (es. RossiMario-0000123456.jar)

Si devono consegnare DUE FILE: l'intero progetto Eclipse e il JAR eseguibile

Si ricorda che compiti *non compilabili* o *palesemente lontani* da 18/30 NON SARANNO CORRETTI e causeranno la verbalizzazione del giudizio “RESPINTO”

Dentinia Trasporti ha richiesto di sviluppare un'app per permettere agli utenti di sapere *l'orario di passaggio della prossima corsa* in una qualunque fermata della sua rete di tram.

DESCRIZIONE DEL DOMINIO DEL PROBLEMA

La rete di tram di Dentinia è costituita da varie linee “da punto a punto” (non esistono linee circolari): tutte le linee sono percorse nei due sensi e tutte le fermate sono servite sia dalle corse “di andata” (dal capolinea A al capolinea B) sia da quelle “di ritorno” (dal capolinea B al capolinea A).

Ogni fermata è caratterizzata dal suo nome e dalla distanza in minuti dal capolinea iniziale.

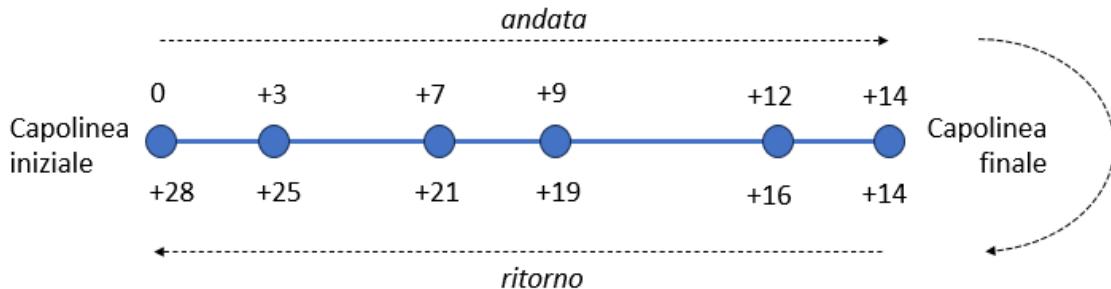
Nessuna linea effettua servizio a cavallo di mezzanotte: ogni linea effettua servizio da un certo orario iniziale (prima corsa), comunque non precedente le 00:00, a un certo orario finale (ultima corsa), comunque non successivo alle 23:59.

Per ipotesi, tutte le corse originano dal capolinea iniziale, effettuano l'intero tragitto fino al capolinea finale e poi – senza attesa – ripartono immediatamente per il capolinea iniziale. Pertanto, l'orario delle corse dal capolinea finale è traslato, rispetto al capolinea iniziale, di una durata pari alla lunghezza in minuti del tragitto e ogni corsa ritorna al capolinea iniziale dopo un tempo pari al doppio della durata del tragitto (v. figura).

La frequenza naturale del servizio, ossia quella che si avrebbe usando un solo bus che faccia avanti/indietro, è quindi pari al doppio della durata del tragitto. È possibile garantire corse più frequenti utilizzando più bus: in tal caso, la frequenza adottata dev'essere necessariamente sottomultiplo della frequenza naturale.

ESEMPIO: una linea che faccia servizio al capolinea iniziale dalle 6:30 alle 21:26, con durata del tragitto fino al capolinea finale di 14 minuti, avrà corse “di andata” dal capolinea iniziale fra le 6:30 e le 21:26 e corse “di ritorno” dal capolinea finale dalle 6:44 alle 21:40, che torneranno al capolinea iniziale fra le 6:58 e le 21:54, rispettivamente.

La frequenza naturale del servizio è quindi di 28 minuti, corrispondente al servizio di un solo bus. Volendo garantire corse più frequenti, le frequenze possibili sono i sottomultipli di 28 e dunque 14' (usando due bus), 7' (usandone quattro), 4' (usandone sette) o 1' (usandone ventotto).



Questo modello di funzionamento richiede perciò il rispetto di due vincoli:

1. La durata del servizio (differenza fra ultima e prima corsa) dev'essere multiplo pari della durata del tragitto
2. La frequenza operativa del servizio dev'essere uguale o sottomultiplo della frequenza naturale del tragitto

Nel caso sopra, la durata del servizio è di 14h56 = 896', che in effetti è multiplo pari della durata del tragitto (14'); inoltre, la frequenza naturale è 28', compatibile quindi con una frequenza operativa – ad esempio – di 7'.

Le linee sono descritte nei file di testo [Linee.txt](#), nel formato esplicitato più oltre.

TEMPO STIMATO PER SVOLGERE L'INTERO COMPITO: 2h15 – 3h

PARTE 1 – Modello dei dati:	Punti 13	[TEMPO STIMATO: 75-90 minuti]
PARTE 2 – Persistenza:	Punti 9	[TEMPO STIMATO: 35-50 minuti]
PARTE 3 – Grafica:	Punti 8	[TEMPO STIMATO: 25-40 minuti]

JAVAFX – Parametri run configuration nei LAB

```
--module-path "C:\applicativi\moduli\javafx-sdk-19\lib"  
--add-modules javafx.controls
```

Cose da ricordare

- salva costantemente il tuo lavoro: l'informatica a volte può essere “subdolamente ostile”..
- in particolare: se ora compila e stai per fare modifiche, salva la versione attuale (non si sa mai)

Checklist di consegna

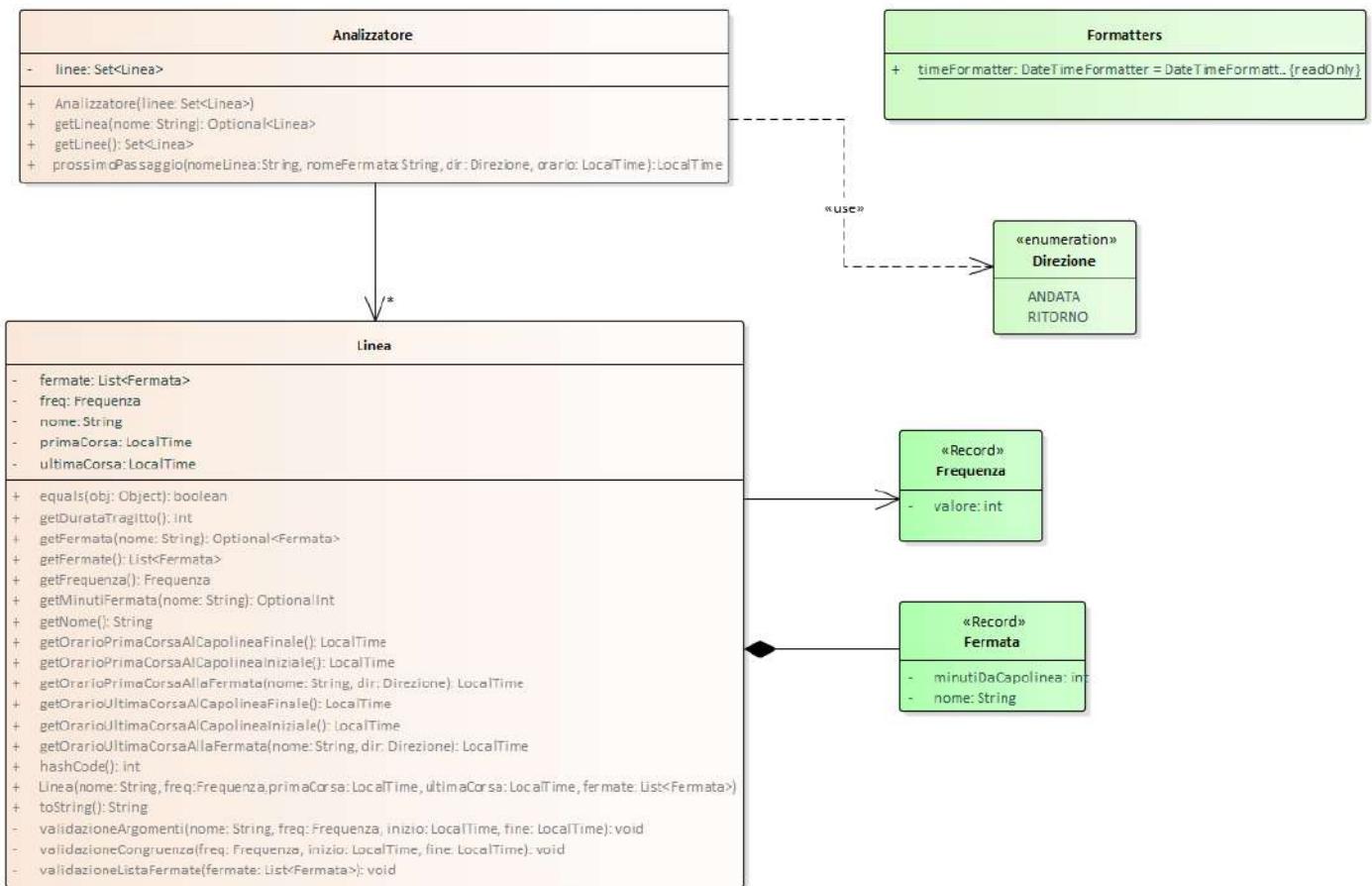
- Hai fatto un **JAR eseguibile**, che contenga cioè l'indicazione del main?
- Hai controllato che **si compili e ci sia tutto**? [NB: non includere il PDF del testo]
- Hai **rinominato IL PROGETTO**, lo ZIP e il JAR esattamente come richiesto?
- Hai **chiamato** la cartella del progetto esattamente come richiesto?
- **Hai fatto un unico file ZIP (NON .7z, rar o altri formati) contenente l'intero progetto?**
In particolare, ti sei assicurato di aver incluso tutti i file .java (e non solo i .class)?
- **Hai consegnato DUE file distinti, ossia lo ZIP col progetto e il JAR eseguibile?**
- Su EOL, hai premuto il tasto “CONFERMA” per inviare il tuo elaborato?

Parte 1 – Modello dei dati

(punti: 13)

Package: tram.model

[TEMPO STIMATO: 75-90 minuti]



SEMANTICA:

- L'enumerativo **Direzione** (fornito) definisce semplicemente le due costanti **ANDATA** e **RITORNO**
- Il record **Frequenza** (fornito) costituisce un *wrapper* per un valore intero al fine di garantire maggiore espressività rispetto al mero uso di un tipo primitivo: il suo costruttore verifica che il valore encapsulato sia compreso fra 1 e 60, estremi inclusi, lanciando **IllegalArgumentException** in caso contrario.
[NB: si ricorda che un record gode della generazione automatica di **equals**, **hashcode**, **toString** e di accessori di nome uguali agli argomenti, in questo caso quindi **valore**]
- Analogamente, il record **Fermata** (fornito) definisce una fermata come coppia (**nome**, **distanza temporale**), intendendosi con questo termine la distanza in minuti dal capolinea iniziale. Il costruttore verifica che il nome non sia nullo né blank e che il valore temporale encapsulato non sia negativo, lanciando **IllegalArgumentException** se necessario. La **toString** è definita in modo da restituire soltanto il nome della fermata.
- La classe **Formatters** (fornita) definisce il formattatore (**timeFormatter**) per formattare e fare parsing di orari secondo lo standard italiano.
- La classe **Linea** (da completare nella sola validazione) descrive una linea intesa come entità dotata di nome, orario di servizio (ossia orario della prima e dell'ultima corsa al capolinea iniziale), frequenza operativa ed elenco di fermate. Sono forniti già implementati tutti i metodi pubblici: rimane da realizzare solo quello, privato, utilizzato dal costruttore, che effettua le verifiche di congruenza relative al rispetto dei vincoli specificati dal Dominio del Problema (**validazioneCongruenza**). Sono quindi forniti già implementati:
 - **equals**, **hashcode**, **toString** tutte definite in modo da considerare soltanto il nome della linea

- `getNome`, `getFrequenza`, `getOrarioPrimaCorsaAlCapolineaIniziale`, `getOrarioUltimaCorsaAlCapolineaIniziale`, `getFermate` che restituiscono semplicemente gli argomenti ricevuti dal costruttore
- `getOrarioPrimaCorsaAlCapolineaFinale`, `getOrarioUltimaCorsaAlCapolineaFinale`, che calcolano e restituiscono i corrispondenti orari al capolinea finale, tenuto conto della durata del tragitto
- `getOrarioPrimaCorsaAllaFermata(String nome, Direzione dir)`, `getOrarioUltimaCorsaAllaFermata(String nome, Direzione dir)`, che calcolano e restituiscono gli orari della prima/ultima corsa alla fermata specificata, tenuto conto della direzione del viaggio (corsa di andata o di ritorno)
- `getDurataTragitto`, che calcola la durata del tragitto intesa come differenza fra gli orari della prima e dell'ultima corsa al capolinea iniziale
- `getFermata(String nome)`, che cerca e restituisce, se esiste, la fermata di nome uguale a quello passato come argomento: per questa ragione il tipo restituito è un ***Optional<Fermata>***
- `getMinutiFermata(String nome)`, che, analogamente a sopra, cerca, se esiste, la fermata di nome uguale a quello passato come argomento e nel caso restituisce la sua distanza in minuti dal capolinea iniziale: per questa ragione il tipo restituito è un ***OptionalInt***
- i due metodi privati `validazioneArgomenti` e `validazioneListaFermate`, che verificano, rispettivamente:
 - `validazioneArgomenti`, che gli argomenti siano non nulli (nel caso di stringhe, anche non-blank) e che gli orari di inizio e fine siano coerenti (ossia, il secondo non preceda il primo)
 - `validazioneListaFermate`, che non vi siano nomi di fermate duplicati, che non vi siano due fermate con la stessa distanza temporale dal capolinea iniziale, e che vi sia sempre una fermata con distanza temporale 0 dal capolinea iniziale (il capolinea iniziale stesso)

Deve invece essere implementato:

- `validazioneCongruenza`, che deve verificare sia che la durata del servizio sia coerente con la durata del tragitto (vincolo 1), sia che quest'ultima sia coerente con la frequenza del servizio (vincolo 2), lanciando ***IllegalArgumentException*** con opportuno messaggio d'errore in caso contrario
- f) La classe ***Analizzatore*** (da completare) costituisce il componente software fondamentale, in grado di calcolare l'orario del prossimo passaggio di una qualunque linea a una qualunque fermata. A tal fine:
- Il costruttore riceve il set di linee che costituiscono la rete, recuperabile tramite l'accessor `getLinee`; l'ulteriore accessor `getLinea(String nome)` restituisce, se esiste, la ***Linea*** di nome uguale all'argomento fornito: per questa ragione il tipo restituito è un ***Optional<Linea>***
 - il metodo ***prossimoPassaggio (da implementare)*** calcola l'orario della prossima corsa *di orario pari o successivo* a quello specificato come (quarto) argomento. Più in dettaglio, esso ricerca, nella linea ricevuta come primo argomento, la fermata ricevuta come secondo argomento e calcola l'orario del prossimo passaggio nella direzione (andata o ritorno) specificata dal terzo argomento. Il metodo deve preventivamente verificare, lanciando ***IllegalArgumentException*** in caso di violazione:
 - che gli argomenti non siano nulli, o, nel caso di stringhe, anche blank
 - che una *linea* di nome uguale a quello specificato esista realmente
- Non occorre invece verificare che una *fermata* di nome uguale a quello specificato esista realmente perché tale controllo è già svolto nei metodi `getOrarioXXX` di ***Linea***, che lanciano anch'essi ***IllegalArgumentException*** in tale evenienza.

Successivamente, il metodo agisce come segue:

- calcola l'orario di servizio (prima/ultima corsa) alla fermata richiesta *nella direzione indicata*
- se l'orario richiesto è esterno all'orario di servizio così calcolato, restituisce l'orario della prima corsa (che si svolgerà, sottinteso, nel giorno successivo)

- o se invece l'orario richiesto è ricompreso nell'orario di servizio odierno, calcola, tenendo conto della frequenza del servizio, l'orario della prima corsa di orario uguale o successivo a quello richiesto, e lo restituisce come risultato [NB: vedere i test nel codice per moltissimi esempi]
- SUGGERIMENTO:** calcolare quante corse ci stanno nell'intervallo temporale fra l'orario richiesto e quello della prima corsa, e sottrarne la durata da tale intervallo...

ESEMPIO:

- linea operante al capolinea iniziale dalle 06:30 alle 21:26, durata tragitto 14', frequenza 7';
- fermata richiesta "Giardino", a 10' dal capolinea iniziale (e quindi a 4' dal capolinea finale)

ESEMPI DI RICHIESTE:

- possibili orari richiesti in ANDATA: 06:30, 13:20, 13:26, 20:45, 21:45
- possibili orari richiesti in RITORNO: 06:30, 19:59, 20:00, 20:50, 21:50

COME OPERA IL METODO:

- preliminarmente calcola l'orario di servizio a tale fermata in ANDATA o RITORNO, secondo la direzione richiesta: nel primo caso otterrà 06:40-21:36, nel secondo 06:48-21:44
- confronta quindi l'orario richiesto per la corsa con l'orario di servizio sopra calcolato
- per orari esterni all'orario di servizio (quindi nell'esempio 06:30, 21:45 in andata, 06:30, 21:50 in ritorno) risponde indicando la prima corsa (eventualmente, da intendersi del giorno successivo per gli orari a tarda sera), ovvero 06:40 in andata o 06:48 in ritorno
- per quelli interni, calcola la prossima corsa a questa fermata tenendo conto della frequenza del servizio, ottenendo in questo caso, rispettivamente, per l'andata 13:26 (sia per la richiesta delle 13:20 che per quella delle 13:26), 20:47 (per quella delle 20:45), per il ritorno 19:59, 20:06, 20:55.

Parte 2 – Persistenza

Package: tram.persistence

(punti: 9)

[TEMPO STIMATO: 35-50 minuti]

Le linee di tram sono elencate nel file di testo **Linee.txt**: non si sa quante siano, ma ognuna è descritta su due righe, entrambe articolate in elementi separati da virgole. Più precisamente:

- la prima riga specifica il nome della linea, la frequenza e gli orari di servizio al capolinea iniziale: il nome è preceduto dalla parola "Linea" seguita da spazi, la frequenza è un valore intero non negativo e non superiore a 60, ricompreso fra le due parole "frequenza" e "minuti", separate da spazi, mentre gli orari della prima e dell'ultima corsa al capolinea iniziale sono espressi nel formato italiano short HH:MM, separati fra loro da un trattino (senza spazi intermedi)
- la seconda riga elenca invece le varie fermate, indicando per ciascuna:
 - o il nome, eventualmente seguito da spazi per comodità di lettura
 - o la distanza temporale in minuti dal capolinea iniziale (un intero non negativo), indicata fra parentesi tonde.

NB: le fermate sono elencate senza alcun particolare ordine, ma devono avere, in ciascuna linea, nomi distinti e distanze temporali tutte distinte: ciò è verificato, come già spiegato, dal costruttore di **Linea**.

È ovviamente invece senz'altro possibile che una fermata di egual nome serva più linee (con distanze temporali probabilmente diverse da una linea all'altra).

ESEMPIO

Linea Rossa, frequenza 5 minuti, 06:30-21:30

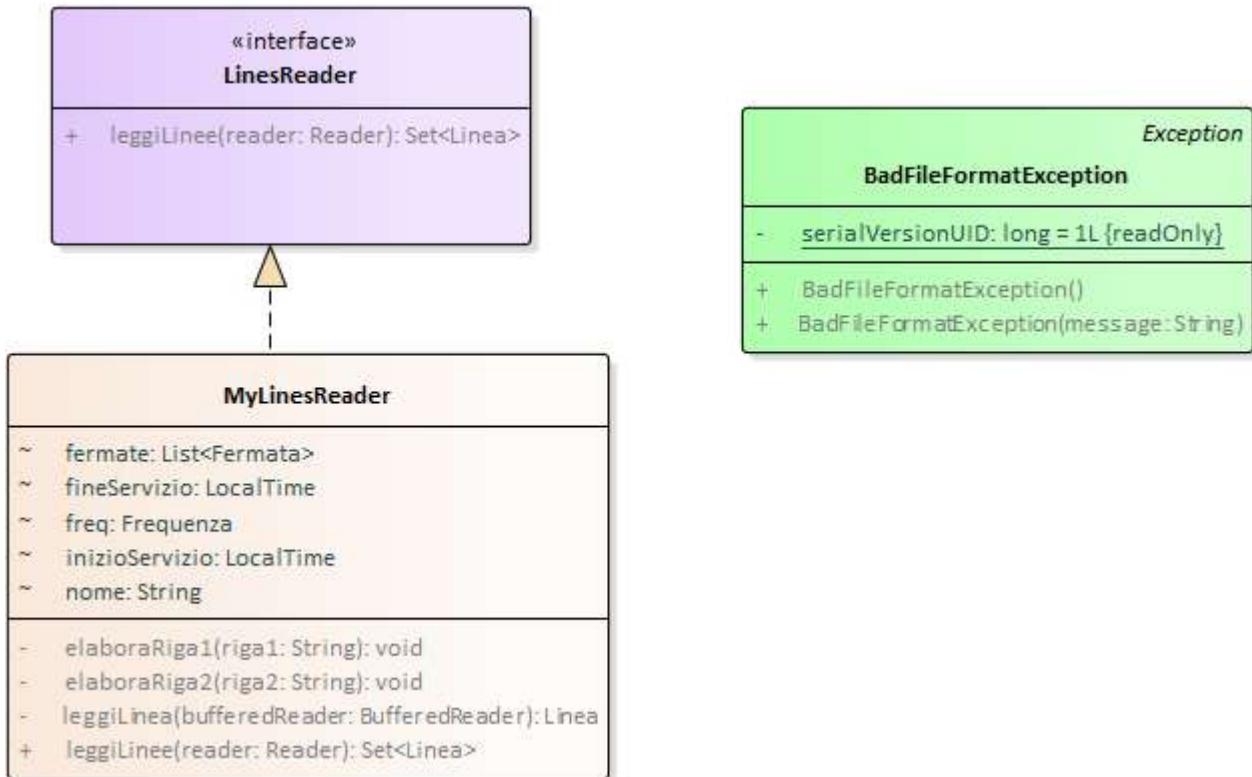
Piazza Celestini (0), via Amendola (3), ..., porta Fiorentina (25)

Linea Verde, frequenza 8 minuti, 06:00-22:00

Porta Romana (0), via Garibaldi (4), via Morandi (7), ..., Stazione FS (24)

Linea Gialla, frequenza 10 minuti, 06:00-23:20

Autostazione (0), via dei Frati (2), via delle Querce (5), ..., largo Nuvolari (20)



SEMANTICA:

- L'interfaccia **LinesReader** (fornita) dichiara il metodo `leggiLinee` che carica da un apposito Reader (già aperto) i dati necessari e restituisce un **Set** di **Linea** opportunamente popolato. Devono essere lanciate:
 - IllegalArgumentException** con opportuno messaggio d'errore in caso di argomento (reader) nullo;
 - BadFormatException** con messaggio d'errore appropriato in caso di problemi nel formato del file (mancanza/eccesso di elementi, errori nel formato dei numeri o degli orari, etc.)
 - una **IOException** in caso di altri problemi di I/O.
- La classe **LinesReader** (da completare) implementa **LinesReader** secondo le specifiche sopra descritte. Il metodo pubblico `leggiLinee` è fornito già implementato e si avvale del metodo ausiliario privato `leggiLinee` (anch'esso fornito) per leggere dal BufferedReader una singola **Linea**. A sua volta, quest'ultimo delega l'elaborazione delle due righe ad altrettanti metodi ausiliari privati **elaboraRiga1** ed **elaboraRiga2** (da implementare).

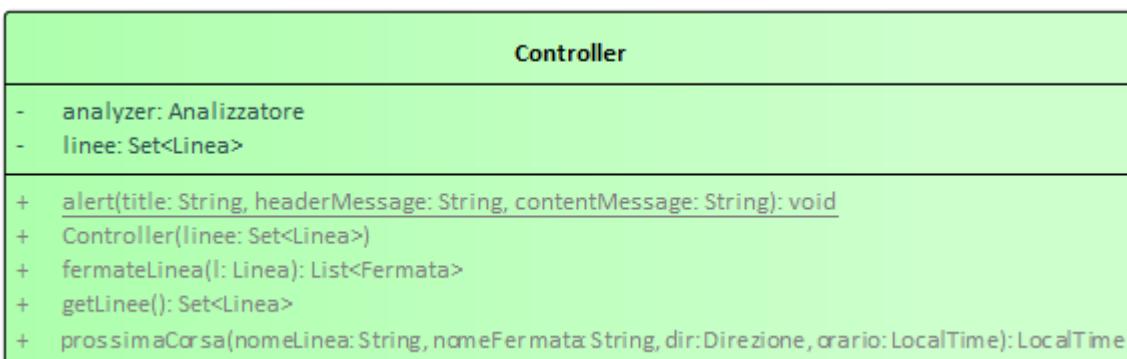
Parte 3

(punti: 8)

Package: **tram.controller**

(punti 0)

Il Controller funge semplicemente da front-end verso l'analizzatore, ed è quindi è organizzato come segue:



SEMANTICA:

La classe **Controller** (fornita) riceve in fase di costruzione l'insieme di linee della rete, che usa poi – previa validazione di coerenza – per costruire internamente l'analizzatore, usato poi per rispondere alle interrogazioni dell'utente.

I metodi si limitano a richiamare quelli di **Analizzatore**, talora con minime differenze di naming. In particolare:

- `getLinee` restituisce l'insieme delle linee fornito al costruttore
- `fermateLinea` restituisce la lista delle fermate di una data Linea: se essa non esiste, lancia **IllegalArgument-Exception** con apposito messaggio.
- `prossimaCorsa` è un semplice front-end verso il metodo `prossimoPassaggio` di **Analizzatore**

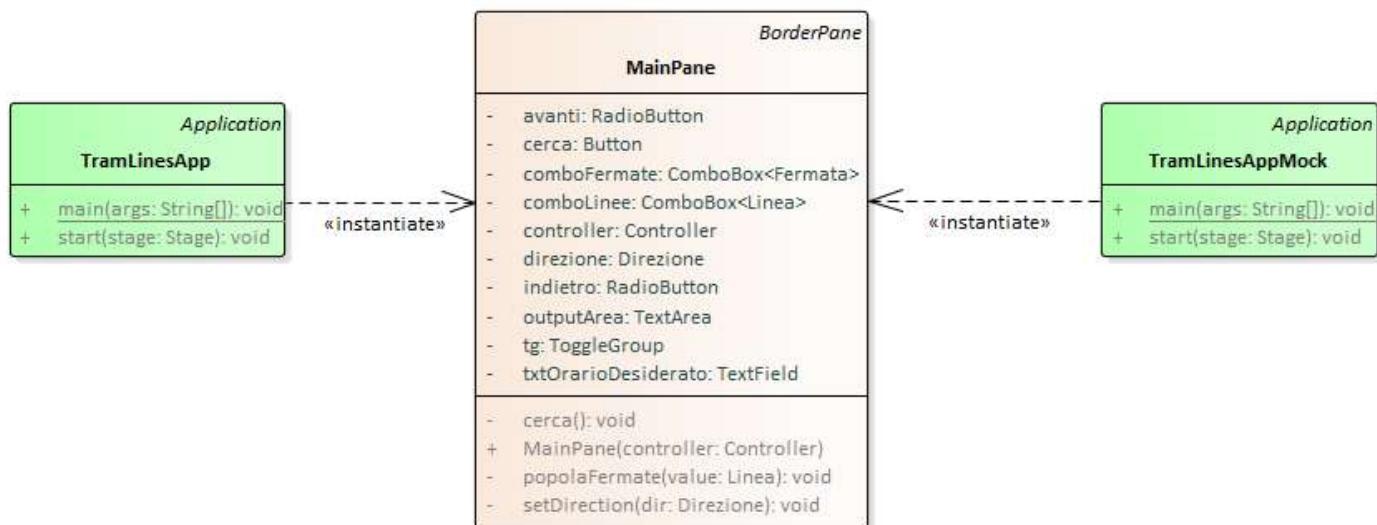
Infine, il metodo statico `alert`, utilizzabile anche dal **MainPane**, consente di far comparire all'utente, ove occorra, una finestra di dialogo con opportuno messaggio d'errore.

Package: tram.ui

[TEMPO STIMATO: 25-40 minuti] (punti 8)

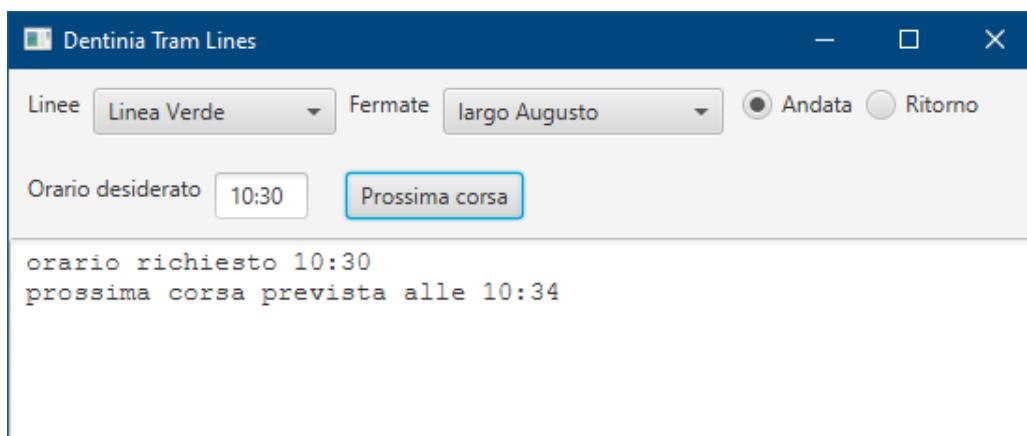
La classe **TramLinesApp** (fornita) costituisce l'applicazione JavaFX che si occupa di aprire i file, creare il controller e incorporare il **MainPane**. Per consentire di collaudare la GUI anche in assenza / in caso di malfunzionamento della parte di persistenza, è possibile avviare l'applicazione mediante la classe **TramLinesAppMock** (NB: essa utilizza linee diverse da quelle elencate nel file)

L'interfaccia utente è illustrata nelle figure seguenti e segue il modello sotto illustrato:



L'interfaccia grafica si presenta come segue:

- in alto, le due combo con l'elenco delle linee e delle fermate, seguite dai radiobutton per la scelta della direzione
- al centro, il campo di testo in cui scrivere l'orario desiderato e, a fianco, il bottone per attivare la ricerca
- in basso, un'area di testo in cui vengono mostrati i risultati.



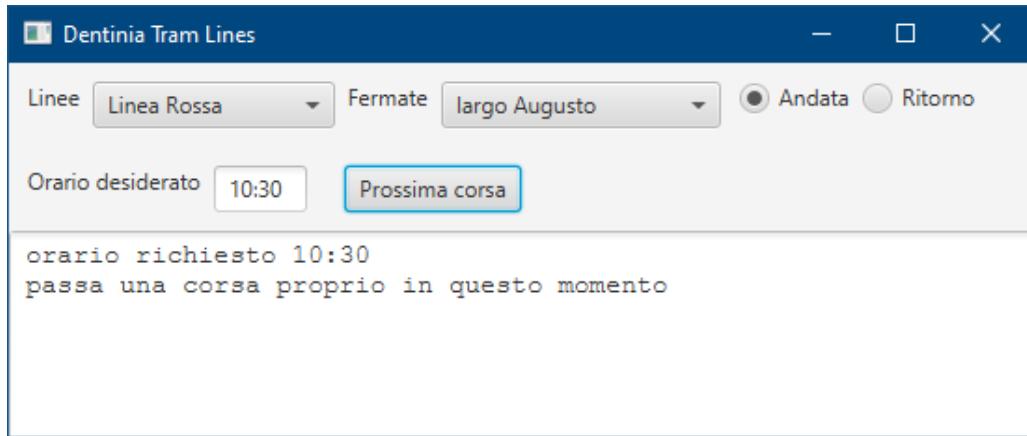
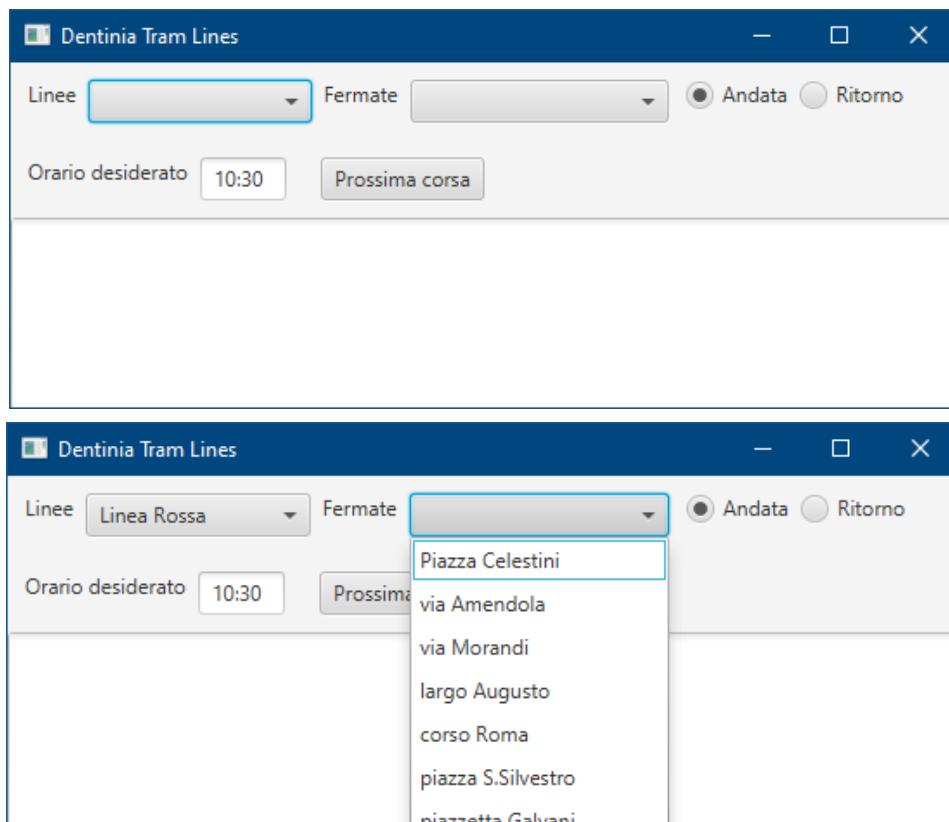


Fig. 1: vista generale della GUI. In alto le combo per scegliere linea e fermata, e i radiobutton per la scelta della direzione; a seguire, il campo per inserire l'orario e il pulsante per attivare la ricerca. In basso, l'area risultati.

Comportamento:

- preliminarmente l'utente deve selezionare la linea di suo interesse dalla combo in alto a sinistra: ciò determina il caricamento, nella combo a fianco, delle fermate della linea prescelta
- l'utente deve poi scegliere la fermata di suo interesse dalla seconda combo (non accade ancora nulla) e selezionare la direzione della corsa di suo interesse tramite gli appositi radiobutton (anche qui non accade nulla)
- a questo punto l'utente deve digitare nel campo di testo l'orario desiderato, nel formato italiano HH:MM (non accade ancora nulla) e infine premere il pulsante "Prossima corsa"
- la pressione del pulsante attiva la ricerca e determina l'apparizione del risultato nell'area di testo sottostante.

Il testo visualizzato deve riportare, come nell'esempio, sia l'orario richiesto, sia quello della prossima corsa: nel caso essa sia proprio all'orario indicato, ciò va adeguatamente evidenziato con messaggistica ad hoc.



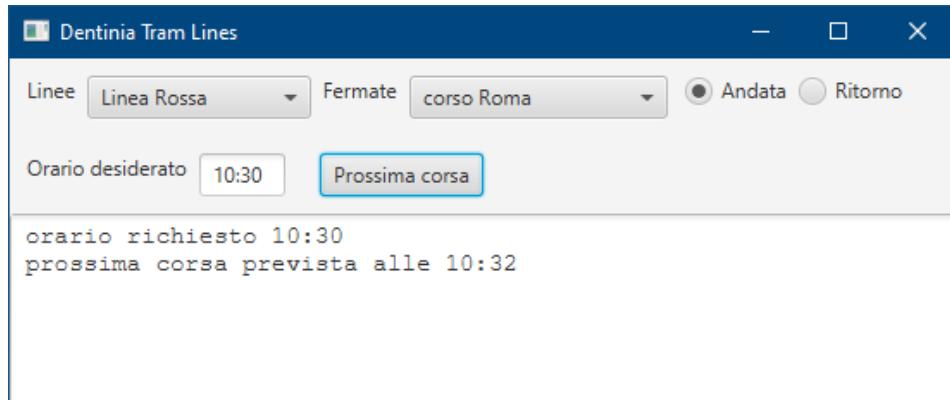


Fig. 2: in alto, la situazione iniziale; al centro, la GUI dopo aver scelto la linea: la seconda combo è stata popolata con l'elenco delle corrispondenti fermate. In basso, dopo aver scelto la fermata (nonché direzione e orario), la pressione del pulsante “Prossima corsa” determina la produzione nell’area di output del risultato richiesto.

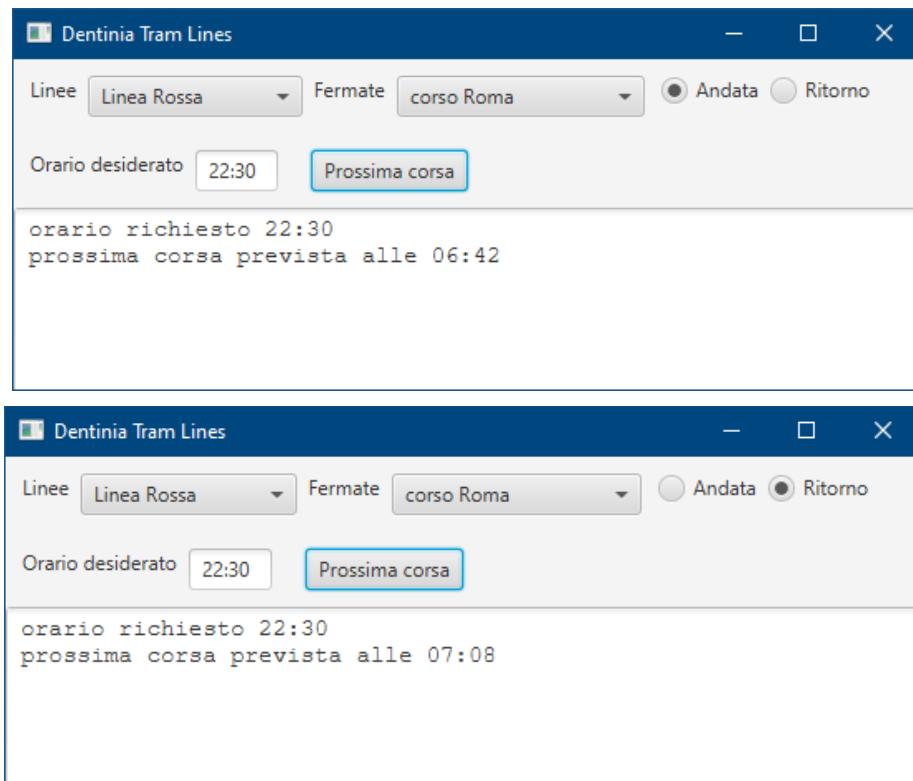
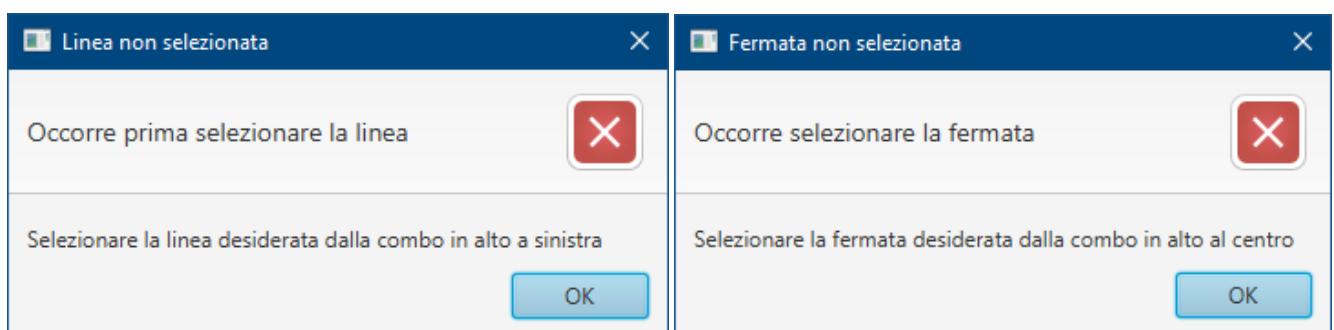


Fig. 3: sopra: se l’orario indicato è successivo al termine del servizio, viene proposta la prima corsa del giorno successivo. Sotto: stesso scenario ma relativamente alla corsa di ritorno..



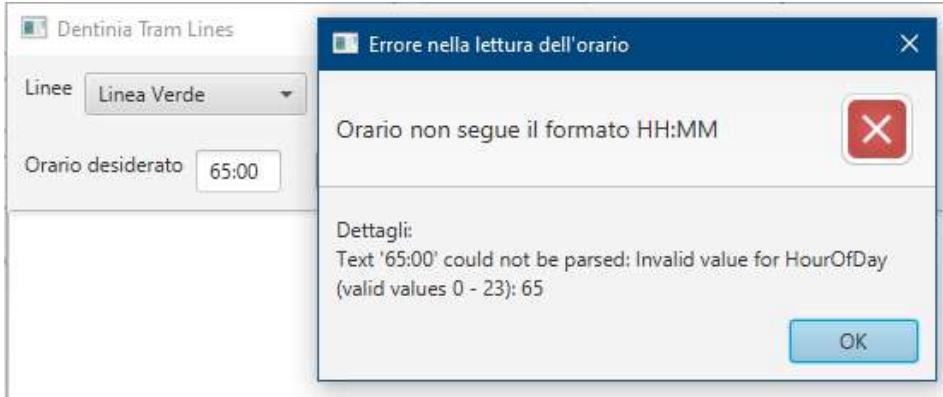


Fig. 4: in alto: messaggi d'errore nel caso si attivi la ricerca rispettivamente senza aver prima selezionato la linea (a sinistra) o la fermata (a destra); sotto, nel caso in cui l'orario sia scorretto o non rispetti il formato richiesto.

Il MainPane è fornito *parzialmente realizzato*: è presente quasi tutta la parte strutturale ed è già implementata la gran parte dei metodi. Rimangono da realizzare:

- 1) il popolamento iniziale della combo delle linee
- 2) il relativo gestore degli eventi (metodo privato `popolaFermate`) che popola l'altra combo
- 3) il metodo privato `cerca`, che gestisce il pulsante “Prossima corsa”

In particolare, quest'ultimo deve:

- preliminarmente, recuperare la linea selezionata dalla combo linee: se questa è nulla, si emette un messaggio d'errore (tramite il metodo `alert` del **Controller**) e si ritorna senza fare nulla
- recuperare la fermata selezionata dalla combo fermate: anche in questo caso, se essa è nulla si deve emettere un messaggio d'errore (tramite il metodo `alert` del **Controller**) e ritornare senza fare nulla
- recuperare l'orario dal campo di testo apposito e farne il parsing: ancora una volta, in caso di errore occorre emettere un apposito messaggio (tramite il metodo `alert` del **Controller**) e ritornare
- infine, tramite il controller, provvedere al calcolo dell'orario della prossima corsa e sintetizzare gli appositi messaggi da emettere sull'area di testo.

Cose da ricordare

- salva costantemente il tuo lavoro: l'informatica a volte può essere “subdolamente ostile”..
- in particolare: se ora compila e stai per fare modifiche, salva la versione attuale (non si sa mai)

Checklist di consegna

- Hai fatto un **JAR eseguibile**, che contenga cioè l'indicazione del main?
- Hai controllato che **si compili e ci sia tutto**? [NB: non includere il PDF del testo]
- Hai **rinominato IL PROGETTO**, lo ZIP e il JAR esattamente come richiesto?
- Hai **chiamato** la cartella del progetto esattamente come richiesto?
- **Hai fatto un unico file ZIP (NON .7z, rar o altri formati) contenente l'intero progetto?**
In particolare, ti sei assicurato di aver incluso tutti i file .java (e non solo i .class)?
- **Hai consegnato DUE file distinti, ossia lo ZIP col progetto e il JAR eseguibile?**
- Su EOL, hai premuto il tasto “CONFERMA” per inviare il tuo elaborato?

ESAME DI FONDAMENTI DI INFORMATICA T-2 del 17/1/2024

Proff. E. Denti – R. Calegari – A. Molesini

Tempo a disposizione: 3h30

NOME PROGETTO ECLIPSE: CognomeNome-matricola (es. RossiMario-0000123456)

NOME CARTELLA PROGETTO: CognomeNome-matricola (es. RossiMario-0000123456)

NOME ZIP DA CONSEGNARE: CognomeNome-matricola.zip (es. RossiMario-0000123456.zip)

NOME JAR DA CONSEGNARE: CognomeNome-matricola.jar (es. RossiMario-0000123456.jar)

Si devono consegnare DUE FILE: *l'intero progetto Eclipse e il JAR eseguibile*

Si ricorda che compiti non compilabili o palesemente lontani da 18/30 NON SARANNO CORRETTI e causeranno la verbalizzazione del giudizio “RESPINTO”

DentiniaTV ha deciso di lanciare un nuovo programma di intrattenimento basato sul noto format del Gioco dei Pacchi. A tal fine ha richiesto lo sviluppo di un'applicazione che supporti tutta l'evoluzione del gioco. *Rispetto al format televisivo, questa versione NON prevede l'offerta del “cambio pacco”, ma solo offerte in denaro.*

DESCRIZIONE DEL DOMINIO DEL PROBLEMA

Il Gioco dei Pacchi prevede preliminarmente l'associazione fra N “territori” (solitamente “Regioni”, ma potrebbero essere anche province, comuni, o altro), altrettanti pacchi numerati da 1 a N, e altrettanti premi, di cui alcuni di basso valore e altri di valore elevato. L'associazione Territorio/Numero pacco/Premio contenuto è sorteggiata segretamente prima dall'inizio del gioco e non è nota ai partecipanti.

Uno dei partecipanti assume il ruolo di “concorrente”, gli altri fungono da attori passivi per aprire i rispettivi pacchi quando il conduttore glielo richiede. **In questa simulazione software, gli attori passivi saranno simulati dal computer, mentre l'utente/giocatore assumerà il ruolo di concorrente.**

Il gioco si basa sull' interazione fra il concorrente e il “Dottore”, personaggio dietro le quinte che, volta per volta:

- dice al concorrente quanti pacchi aprire: ogni pacco aperto è “perso” dal concorrente, nel senso che il premio in esso contenuto non è più disponibile e viene tolto dal tabellone
- dopo che il concorrente ha aperto la quantità di pacchi richiesti, propone al concorrente un'offerta in denaro, “vagamente” proporzionata ai premi ancora disponibili: se il concorrente l'accetta, il gioco termina; altrimenti si passa alla manche successiva, ripetendo le stesse operazioni.

Il numero di pacchi da aprire a ogni manche viene stabilito dal Dottore volta per volta, con l'unico vincolo di garantire che nella manche finale (se raggiunta) rimanga in gioco almeno un pacco, oltre a quello già in possesso del concorrente.

In questa simulazione software:

- **il numero di pacchi da aprire ogni volta è sorteggiato fra 1 e N/3**
- **l'offerta del Dottore è sorteggiata fra 1/4 della media dei premi ancora disponibili e la media stessa.**

Territori e premi sono elencati nei file di testo **Territori.txt** e **Premi.txt**, nel formato descritto più oltre.

TEMPO STIMATO PER SVOLGERE L'INTERO COMPITO: 2h15 – 3h

PARTE 1 – Modello dei dati: Punti 14 [TEMPO STIMATO: 80-95 minuti]

PARTE 2 – Persistenza: Punti 8 [TEMPO STIMATO: 35-50 minuti]

PARTE 3 – Grafica: Punti 8 [TEMPO STIMATO: 20-35 minuti]

JAVAFX – Parametri run configuration nei LAB

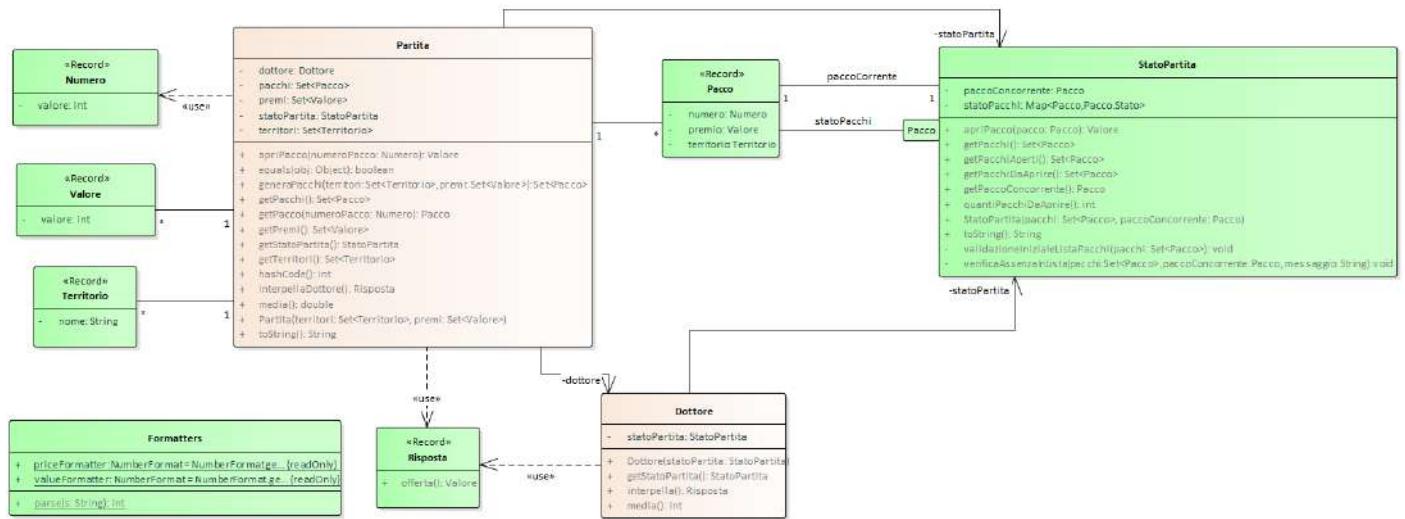
```
--module-path "C:\applicativi\moduli\javafx-sdk-19\lib"  
--add-modules javafx.controls
```

Parte 1 – Modello dei dati

(punti: 14)

Package: pacchi.model

[TEMPO STIMATO: 80-95 minuti]



SEMANTICA:

- I record **Numero**, **Territorio**, **Valore** (forniti) costituiscono altrettanti *wrapper* per un valore (intero o stringa) al fine di garantire maggiore espressività rispetto al mero uso di un tipo primitivo e di una generica stringa. [NB: si ricorda che l'uso di record implica la generazione automatica di *equals*, *hashCode* e di tanti accessori quanti gli argomenti del record, ognuno di nome uguale all'argomento a cui accede]
- Analogamente, il record **Pacco** (fornito) esprime un pacco come associazione (Territorio, Numero, Valore). [NB: al suo interno è definito l'enumerativo di servizio **Stato**, non descritto in questa sede perché utilizzato unicamente da **StatoPartita** per registrare quali pacchi siano aperti e chiusi]
- La classe **Formatters** (fornita) definisce due formattatori, da usare in tutta l'applicazione per formattare e fare parsing di numeri reali (*valueFormatter*) e valori in Euro (*priceFormatter*) secondo lo standard italiano
- La classe **StatoPartita** (fornita) rappresenta lo stato della **Partita** (vedere oltre): il suo ruolo è registrare lo stato dei pacchi aperti/chiusi, nonché verificare la coerenza fra l'insieme dei pacchi dei partecipanti e il pacco del concorrente. Tali aspetti non vengono qui descritti in dettaglio in quanto non è previsto che questa classe debba essere manipolata direttamente dal candidato. Si evidenzia soltanto la presenza di alcuni metodi utili, richiamati comunque più oltre: *getPacchiDaAprire*, *getPaccoConcorrente* che restituiscono rispettivamente i soli pacchi ancora da aprire in mano ai partecipanti (escluso quindi quello del concorrente) e il solo pacco del concorrente. Tali metodi sono utili per implementare la classe **Dottore**, più oltre specificata.
- La classe **Partita** (da completare) ha due compiti: gestire la configurazione del gioco a partire dagli insiemi di territori e premi, e fungere da front-end verso la funzionalità di apertura pacchi gestita dalla classe di back-end **StatoPartita**. Per quanto riguarda il primo compito, controlla in primis la consistenza dei due set ricevuti come argomenti, poi genera tutti gli N pacchi, sorteggia quello del concorrente e lo rimuove dal set dei pacchi, che così facendo contiene d'ora in avanti tutti e soli i pacchi dei partecipanti. Espone molti metodi per accedere a tutti i set e le proprietà rilevanti, ma gli unici da implementare sono i due metodi:
 - apriPacco*, che apre il pacco con il numero specificato, restituendone il premio. Più in dettaglio, preliminariamente verifica che il numero del pacco sia nel range 1..N, altrimenti lancia **IllegalArgument-Exception** con opportuno messaggio d'errore; se tutto è ok, recupera dall'insieme dei pacchi quello con il numero richiesto, lo apre tramite il metodo *apriPacco* di **StatoPartita** e restituisce il **Valore** del premio in esso racchiuso.
 - generaPacchi*, che costruisce gli N pacchi sorteggiando i numeri e accoppiandoli a territori e premi. Più in dettaglio, itera sui due insiemi ricevuti accoppiando un territorio, un premio e un numero di pacco –

quest'ultimo sorvegliato fra 1 e N, senza ripetizioni e senza escluderne alcuno – usandoli per costruire un nuovo **Pacco**, che viene poi aggiunto all'insieme dei pacchi da restituire.

- f) Il record **Risposta** (fornito) incorpora un **Valore** inteso come risposta del Dottore
- g) La classe **Dottore** (da completare nell'implementazione) fornisce anch'essa fondamentalmente due metodi:
 - *interpella*, che sintetizza la **Risposta** con l'offerta del dottore, calcolata sorteggiando un valore casuale compreso fra $M/4$ ed M , dove M è la media dei pacchi ancora da aprire, ottenuta dal metodo **media**
 - *media*, che calcola e restituisce, sotto forma di intero, la media dei pacchi ancora da aprire incluso quello del concorrente, per recuperare i quali si possono sfruttare i metodi **getPacchiDaAprire** e **getPaccoConcorrente** di **StatoPartita**

A tale scopo la classe **Dottore** riceve lo **StatoPartita** come argomento (e, per soli fini di test, lo rende anche accessibile tramite un apposito accessorio).

Parte 2 – Persistenza

(punti: 8)

Package: pacchi.persistence

[TEMPO STIMATO: 35-50 minuti]

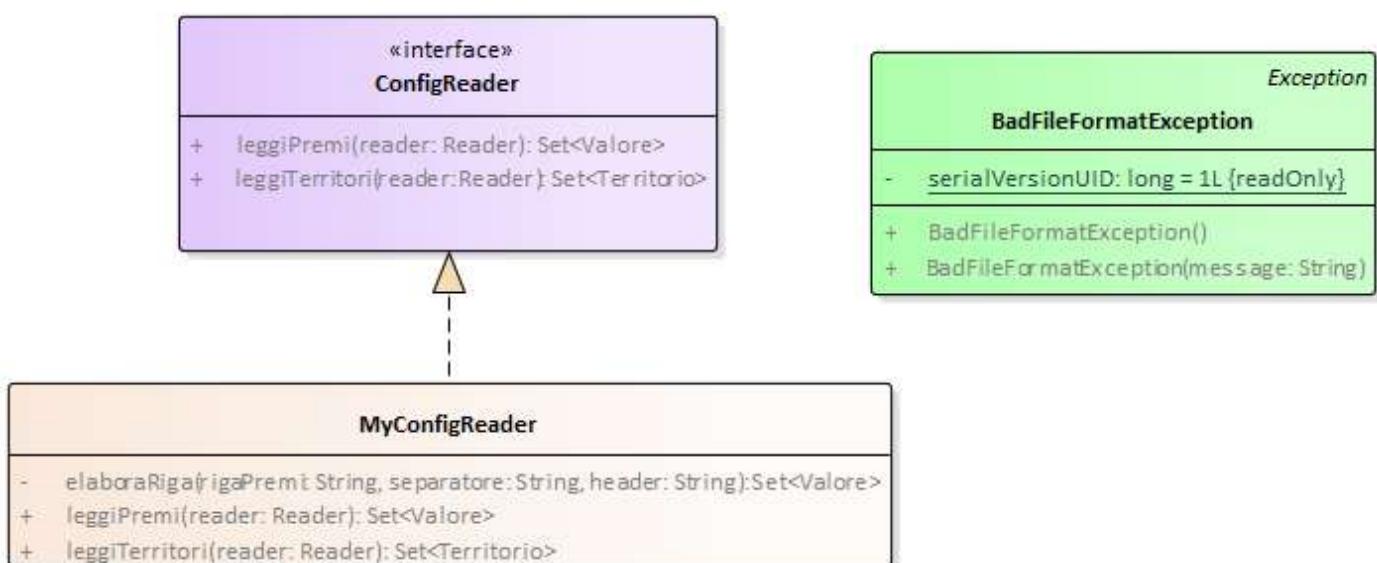
Territori e premi sono elencati nei due file di testo **Territori.txt** e **Premi.txt**, rispettivamente.

- i *territori* sono semplicemente elencati uno per riga: il numero delle righe non è ovviamente noto a priori;
 - i *premi* sono invece elencati in due sole righe, una per i premi bassi (<1000€) e una per i premi alti ($\geq 1000\text{€}$), che devono contenere lo stesso numero di valori, nel seguente formato:
 - intestazione “PREMI ALTI:” o “PREMI BASSI:”, rispettivamente
 - elenco dei rispettivi valori, separati da virgole, formattati come numeri in formato italiano, utilizzando ove appropriato il simbolo delle migliaia (SENZA simbolo di valuta)
- Premi bassi e premi alti devono essere presenti in egual quantità.

ESEMPIO

PREMI BASSI: 0, 1, 5, 10, 20, 50, 75, 100, 200, 500

PREMI ALTI: 5.000, 10.000, 15.000, 20.000, 30.000, ..., 200.000, 300.000



SEMANTICA:

- a) L'interfaccia **ConfigReader** (fornita) dichiara i due metodi *leggiTerritori* e *leggiPremi* che caricano da un apposito Reader (già aperto) i dati necessari, restituendo un **Set** dell'opportuno tipo (**Territorio** o **Valore**) popolata. Entrambi lanciano:
- **IllegalArgumentException** con opportuno messaggio d'errore in caso di argomento (reader) nullo;
 - **BadFormatException** con messaggio d'errore appropriato in caso di problemi nel formato del file (mancanza/eccesso di elementi, errori nel formato dei numeri, etc.), *ivi incluso il caso in cui premi bassi e premi alti non siano presenti in egual quantità*
 - una **IOException** in caso di altri problemi di I/O.
- b) La classe **MyConfigReader** (da completare) implementa **ConfigReader** secondo le specifiche sopra descritte. Il metodo *leggiTerritori* è fornito già implementato: si deve quindi implementare soltanto *leggiPremi*. A tal fine, si conviene di delegare l'elaborazione della singola riga al metodo ausiliario privato *elaboraRiga*.

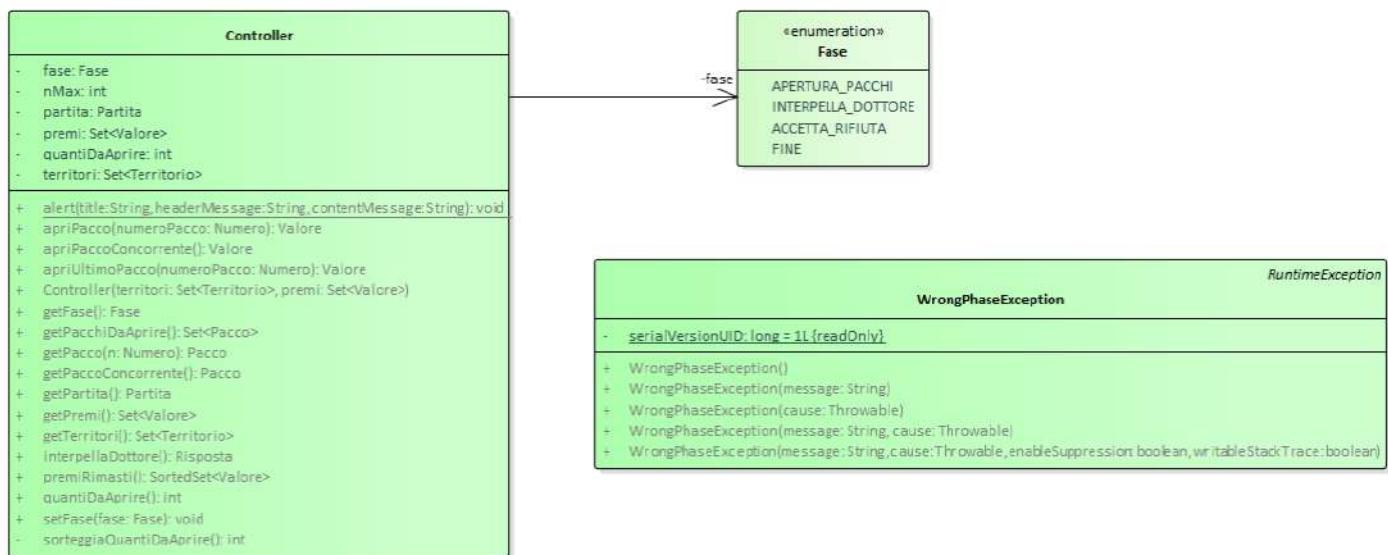
Parte 3

(punti: 8)

Package: pacchi.controller

(punti 0)

Il Controller è organizzato secondo il diagramma UML seguente:



SEMANTICA:

L'enumerativo **Fase** (fornito) descrive le quattro fasi in cui il gioco, e quindi il controller, può trovarsi:

APERTURA_PACCHI, **INTERPELLA_DOTTORE**, **ACCETTA_RIFIUTA**, **FINE**.

La classe **Controller** (fornita) riceve in fase di costruzione gli stessi due insiemi di territori e premi già descritti nella classe **Partita**: se uno o entrambi sono nulli viene lanciata **IllegalArgumentException** con idoneo messaggio d'errore. Se invece tutto è regolare, il costruttore provvede a istanziare internamente la **Partita**, precalcolare il numero massimo di pacchi da aprire a ogni manche (da specifica, N/3) e impostare la fase ad **APERTURA_PACCHI**.

Molti metodi si limitano a richiamare quelli omonimi di **Partita**, altri forniscono invece servizi specifici.

Fra questi:

- **getPacchiDaAprire** restituisce l'insieme di pacchi ancora da aprire in un dato momento

- *apriPacco* apre il pacco di numero specificato, restituendone il valore contenuto. Il metodo agisce solo se la fase corrente è APERTURA_PACCHI, altrimenti lancia **WrongPhaseException** con apposito messaggio. Se il pacco da aprire era l'ultimo di questa manche, commuta la fase alla successiva INTERPELLA_DOTTORE.
- *apriUltimoPacco* è la versione specializzata per aprire l'ultimo pacco rimasto nell'ultima manche, quando il gioco ormai sta per terminare e resta solo da svelare cos'ha vinto il concorrente: per questo agisce solo se la fase attuale è FINE, altrimenti lancia **WrongPhaseException** con apposito messaggio.
- *apriPaccoConcorrente* apre il pacco del concorrente, mentre il metodo simile *getPaccoConcorrente* lo recupera e restituisce ma senza aprirlo
- *getFase/setFase* accedono alla fase corrente e permettono di modificarla
- *quantiDaAprire* restituisce il numero di pacchi da aprire in questa manche, preventivamente sorteggiato all'inizio della fase di apertura pacchi (tramite il metodo privato *sorreggiaQuantiDaAprire*)
- *premiRimasti* restituisce l'insieme ordinato di tutti i premi rimasti in gioco, ossia tutti i premi contenuti nei pacchi ancora da aprire più quello contenuto nel pacco del concorrente (il metodo è pensato per fornire i contenuti da mostrare sul tabellone di gioco).

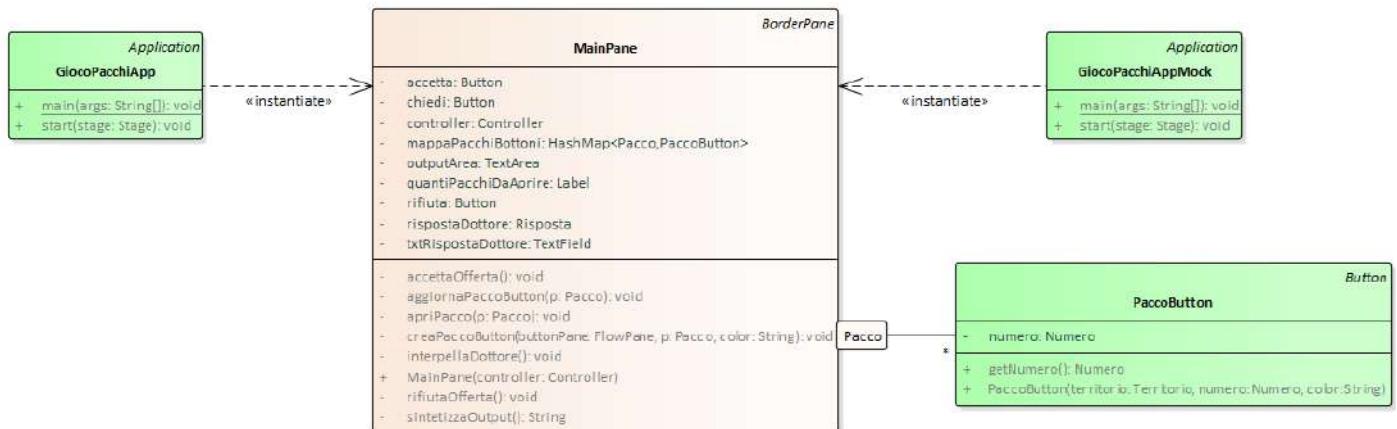
Infine, il metodo statico *alert*, utilizzabile anche dal MainPane, consente di far comparire all'utente, ove occorra, una finestra di dialogo con opportuno messaggio d'errore.

Package: pacchi.ui

[TEMPO STIMATO: 20-35 minuti] (punti 8)

La classe **GioocoPacchiApp** (fornita) costituisce l'applicazione JavaFX che si occupa di aprire i file, creare il controller e incorporare il **MainPane**. Per consentire di collaudare la GUI anche in assenza / in caso di malfunzionamento della parte di persistenza, è possibile avviare l'applicazione mediante la classe **GioocoPacchiAppMock**: essa utilizza un minor numero di territori e può quindi essere utile anche per sveltire i collaudi.

L'interfaccia utente è illustrata nelle figure seguenti e segue il modello sotto illustrato:



L'interfaccia grafica si presenta come segue:

- a sinistra, tutta l'area di gioco, suddivisa in tre zone disposte verticalmente una sotto l'altra;
- a destra, un'area di testo che funge da tabellone, con l'elenco dei premi ancora in gioco e l'indicazione (al top) del pacco in mano al concorrente.

L'area di gioco si articola come segue:

- in alto, il pannello con i pulsanti che rappresentano i pacchi dei partecipanti;
- di seguito, il bottone arancione che rappresenta il pacco del concorrente;
- sotto, l'area di competenza del Dottore, ulteriormente distinta in due parti:
 - o il numero di pacchi ancora da aprire, decrescente fino a 0 ad ogni clic sui pulsanti del pannello;

- l'area più propriamente destinata all'interazione col Dottore, costituita dai pulsanti *Chiedi*, *Accetta*, *Rifiuta* e dal campo di testo (non scrivibile dall'utente) in cui compare l'offerta del Dottore.

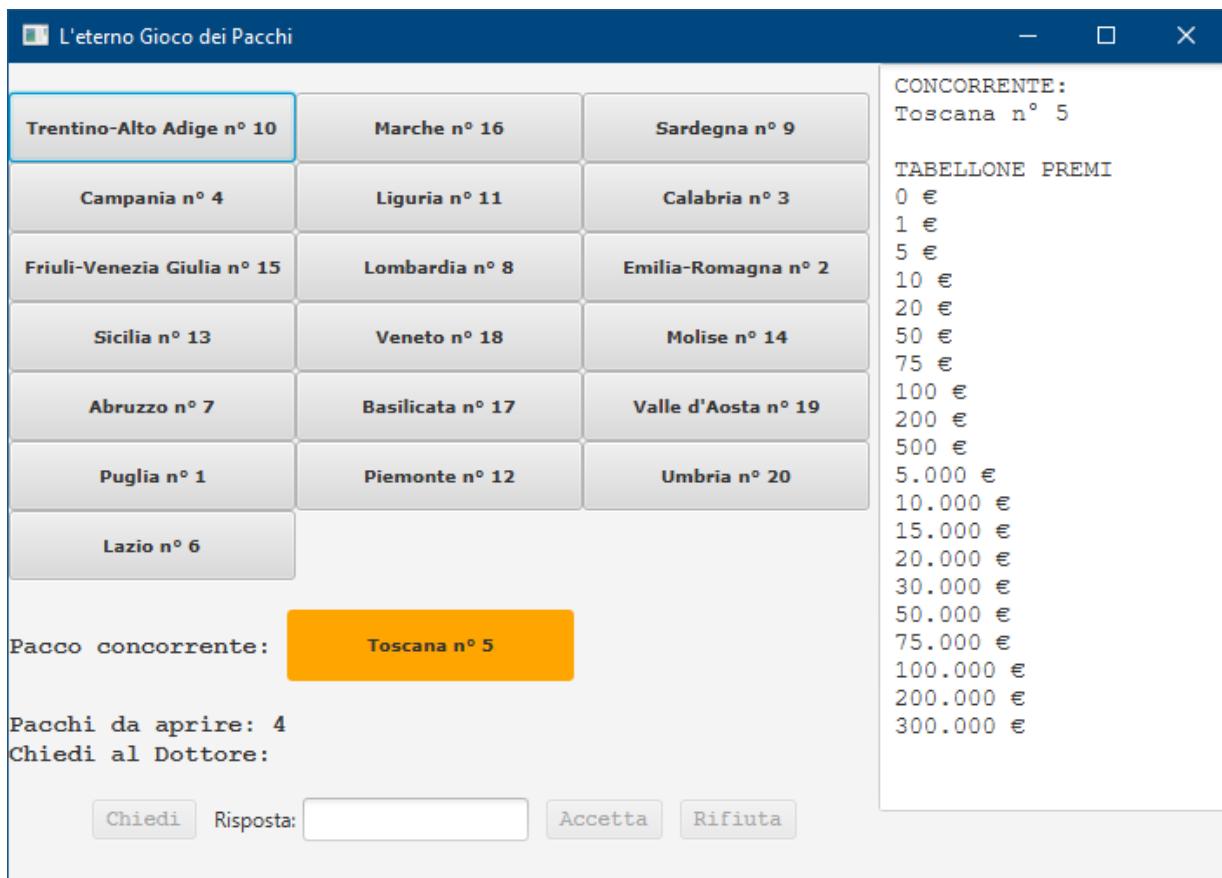


Fig. 1: situazione iniziale. Territori e numeri dei pacchi variano ogni volta, così come l'associazione (ovviamente non visibile in figura) fra Pacco e Premio in esso contenuto. Anche il Pacco concorrente è estratto a sorte ogni volta.

Comportamento:

- il gioco inizia sempre dalla fase di apertura pacchi: il concorrente deve quindi aprire tanti pacchi quanti ne indica il Dottore nella riga apposita (in Fig. 1 sono quattro). Per farlo, basta che clicchi via via sui pulsanti corrispondenti: ogni pacco aperto viene svelato e il corrispondente pulsante disabilitato (Fig. 2). Contestualmente, il relativo premio viene tolto dal tabellone.
- una volta aperti tutti i pacchi richiesti, il gioco passa automaticamente alla fase INTERPELLA_DOTTORE, abilitando il pulsante "Chiedi" (Fig. 2, destra): in questa fase, ogni tentativo di aprire altri pacchi è impedito e causerebbe la comparsa di un alert (non mostrato) che spieghi la ragione del non potersi procedere
- Premendo il pulsante "Chiedi", il dottore formula l'offerta (Fig. 3, sinistra), che può essere accettata o rifiutata premendo gli appositi pulsanti "Accetta" o "Rifiuta", all'uopo abilitati: in contemporanea viene disabilitato invece il pulsante "Chiedi", per evitare che l'utente possa inavvertitamente chiedere un'altra offerta anzitempo.
- se l'offerta viene accettata (Fig. 5, destra), compare un dialogo che svela il contenuto del pacco del concorrente e il gioco termina: in contemporanea, il pulsante arancione che rappresenta tale pacco viene svelato e disabilitato.
- se invece l'offerta è rifiutata (Fig. 3, destra), il gioco torna alla fase iniziale di apertura pacchi (Fig. 4) e si procede come sopra, manche dopo manche.
- se, rifiuto dopo rifiuto, si giunge alla manche finale, il sistema propone un'ultima offerta: rifiutando anche quella, il concorrente vince il contenuto del proprio pacco, che viene quindi svelato (Fig. 5, sinistra).

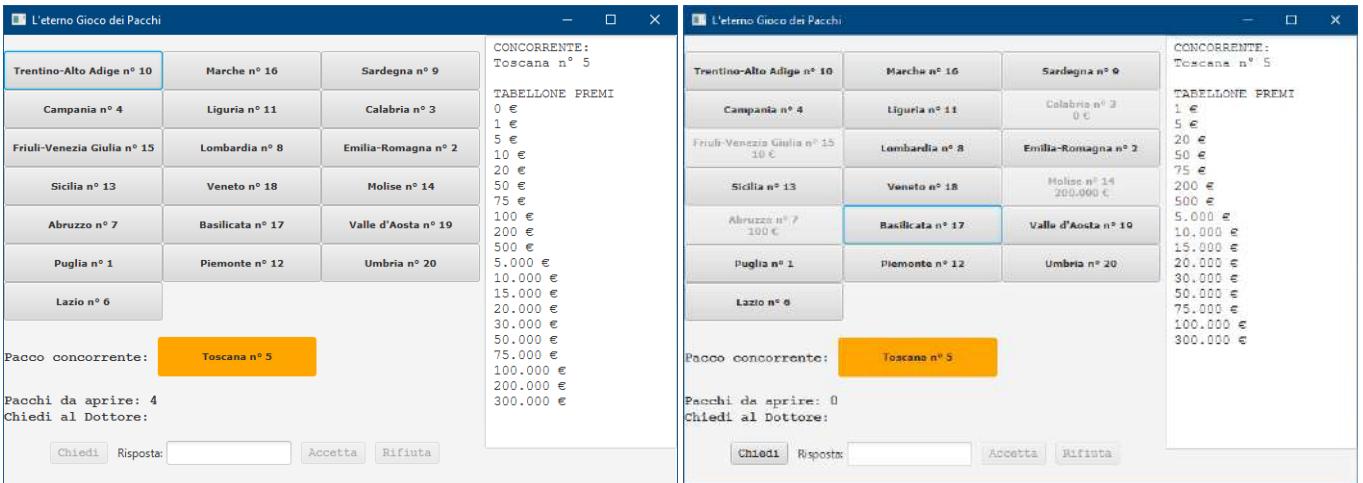


Fig. 2: a sinistra, la situazione iniziale; a destra, la GUI dopo aver aperto i 4 pacchi richiesti dal Dottore. Notare come ciò abiliti il pulsante “Chiedi” per interpellare il Dottore e attendere la sua offerta.

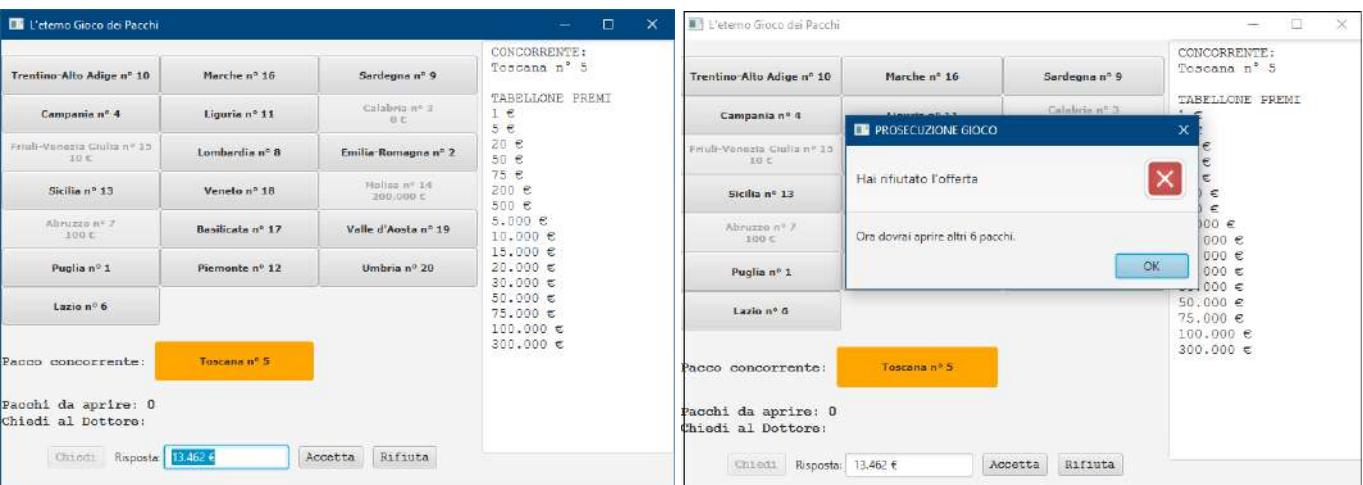


Fig. 3: a sinistra: l'offerta del Dottore è mostrata nell'apposito campo; in parallelo vengono abilitati i due pulsanti “Accetta” e “Rifiuta”, disabilitando il precedente pulsante “Chiedi”. A destra, rifiutando l'offerta appare il dialogo che conferma che il gioco prosegue e informa il concorrente di quanti pacchi dovrà aprire nella prossima manche.

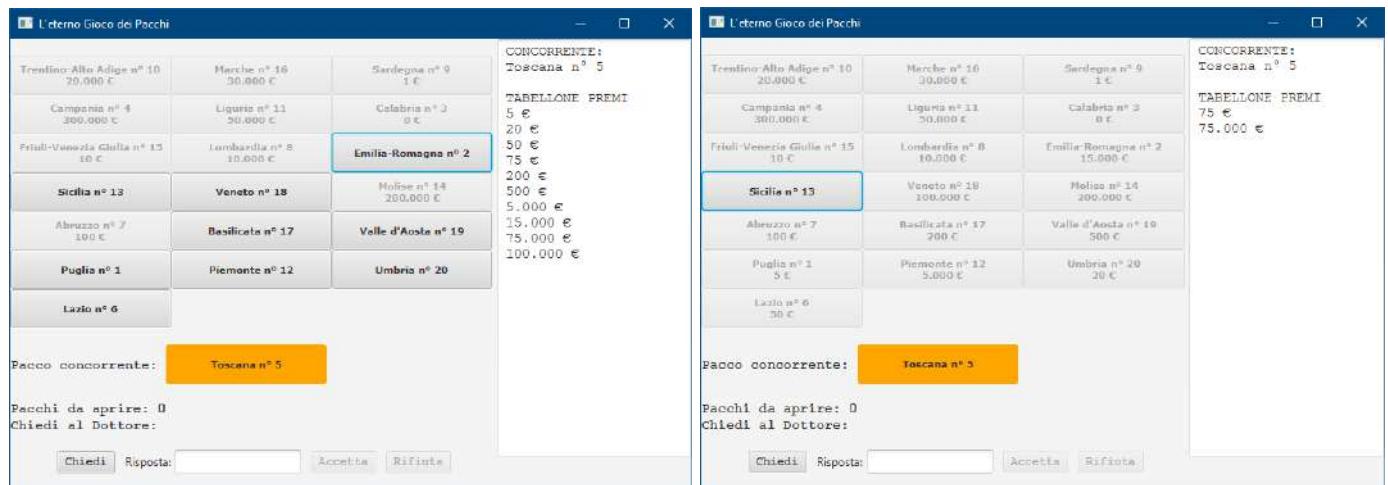


Fig. 4: a sinistra: il gioco prosegue ora come nel caso precedente, aprendo altri pacchi e chiedendo poi la successiva offerta; a destra: continuando a rifiutare tutte le offerte, si giunge infine alla manche finale.

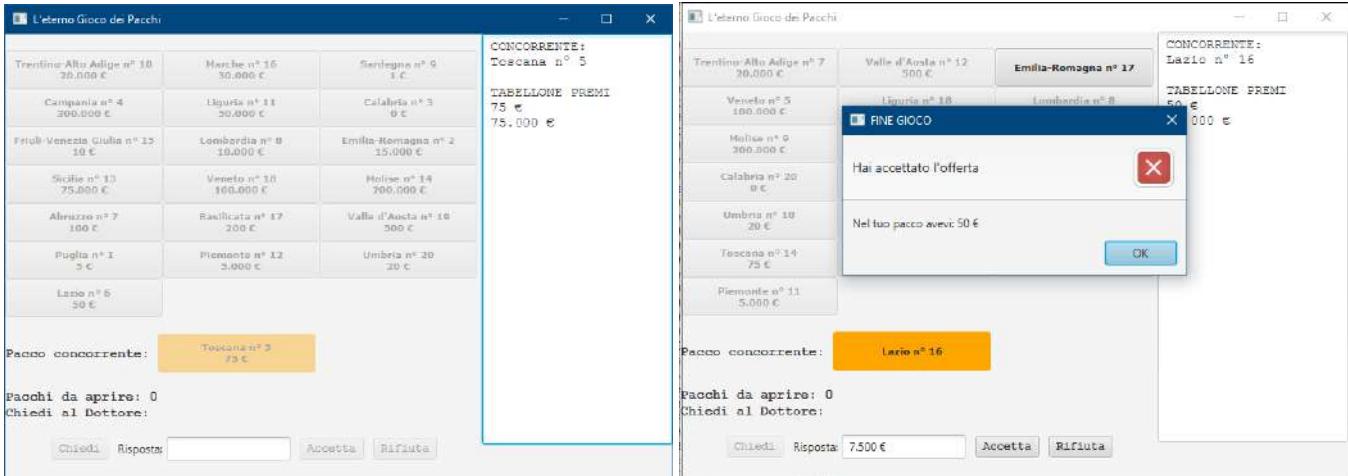


Fig. 5: a sinistra: rifiutando anche l'ultima offerta si vince il contenuto del proprio pacco (in questo caso, 75€); a destra: in alternativa (in un'altra partita) si può in qualsiasi momento accettare un'offerta. Il dialogo mostra quanto c'era nel pacco concorrente, poi la GUI viene aggiornata (non mostrato in figura).

Il MainPane è fornito *parzialmente realizzato*: è presente quasi tutta la parte strutturale ed è già implementata la gran parte dei metodi. Rimangono da realizzare solo tre funzionalità:

- 1) il popolamento iniziale del pannello con gli N-1 button che rappresentano i pacchi dei partecipanti: il compito è facilitato dal metodo privato *creaPaccoButton* (fornito) che provvede anche ad agganciare già ai button il relativo gestore degli eventi, costituito dal metodo privato *apriPacco* (anch'esso fornito)
- 2) il metodo privato *interpellaDottore*, che gestisce il pulsante "Chiedi"
- 3) il metodo privato *sintetizzaOutput*, che produce la stringa (articolata in righe come da figure) che popola il tabellone di gioco (sulla destra nelle figure).

In particolare, il metodo privato *interpellaDottore* deve:

- preliminarmente, verificare che il controller sia nella fase INTERPELLA_DOTTORE: se così non è, deve emettere (tramite il metodo alert del Controller) un messaggio d'errore e ritornare senza fare nulla
- se la fase di gioco è quella corretta, interpellare il Dottore e tramite l'omonimo metodo del controller, e mostrare la risposta nel campo di testo all'uopo previsto
- disabilitare il tasto "Chiedi" e ri-abilitare i due tasti "Accetta" e "Rifiuta"
- reimpostare il controller sulla fase ACCETTA_RIFIUTA, così da predisporsi al giro successivo

Invece, il metodo privato *sintetizzaOutput* deve sintetizzare la stringa di seguito specificata:

- in alto, dopo un'intestazione come "CONCORRENTE", deve indicare territorio e numero del pacco in mano al concorrente (senza ovviamente svelarne il premio!)
- di seguito, dopo una riga bianca e un'intestazione come "TABELLONE PREMI", deve elencare i premi ancora disponibili (cioè quelli contenuti nei pacchi ancora da aprire, incluso ovviamente quello del concorrente), in ordine di valore crescente, utilizzando il formato valuta italiano standard: ovviamente, non deve dire in che pacchi siano contenuti! ☺

IMPORTANTE:

LEGGERE BENE LA CHECKLIST DI CONSEGNA ALLA PAGINA SEGUENTE!

Cose da ricordare

- salva costantemente il tuo lavoro: l'informatica a volte può essere “subdolamente ostile”..
- in particolare: se ora compila e stai per fare modifiche, salva la versione attuale (non si sa mai)

Checklist di consegna

- Hai fatto un **JAR eseguibile**, che contenga cioè l'indicazione del main?
- Hai controllato che **si compili e ci sia tutto**? [NB: non includere il PDF del testo]
- Hai **rinominato IL PROGETTO**, lo ZIP e il JAR esattamente come richiesto?
- Hai **chiamato** la cartella del progetto esattamente come richiesto?
- **Hai fatto un unico file ZIP (NON .7z, rar o altri formati) contenente l'intero progetto?**
In particolare, ti sei assicurato di aver incluso tutti i file .java (e non solo i .class)?
- **Hai consegnato DUE file distinti, ossia lo ZIP col progetto e il JAR eseguibile?**
- Su EOL, hai **premuto** il tasto “CONFERMA” per inviare il tuo elaborato?

ESAME DI FONDAMENTI DI INFORMATICA T-2 del 13/9/2023

Proff. E. Denti – R. Calegari – A. Molesini

Tempo a disposizione: 3h30

NOME PROGETTO ECLIPSE: CognomeNome-matricola (es. RossiMario-0000123456)

NOME CARTELLA PROGETTO: CognomeNome-matricola (es. RossiMario-0000123456)

NOME ZIP DA CONSEGNARE: CognomeNome-matricola.zip (es. RossiMario-0000123456.zip)

NOME JAR DA CONSEGNARE: CognomeNome-matricola.jar (es. RossiMario-0000123456.jar)

Si devono consegnare DUE FILE: *l'intero progetto Eclipse e il JAR eseguibile*

Si ricorda che compiti *non compilabili o palesemente lontani da 18/30* NON SARANNO CORRETTI e causeranno la verbalizzazione del giudizio “RESPINTO”

Nella ridente città di Dentinia è festa grande: ha appena aperto i battenti un nuovo mirabile fast food, DentBurger! L'applicazione deve quindi consentire agli utenti di ordinare il proprio pasto scegliendo dal vasto menù.

DESCRIZIONE DEL DOMINIO DEL PROBLEMA

I prodotti offerti da DentBurger si distinguono in diverse *categorie*: bevande, piatti, contorni, condimenti e dessert.

Per ogni categoria, il menù offre uno o più *generi*: ad esempio, come piatti può offrire panini, insalate, snack; come condimenti, patatine, chips, o altro; e così via.

Per ogni genere sono presenti nel menù uno o più *prodotti*, ognuno caratterizzato quindi da:

- categoria
- genere (denominazione)
- specifica dettagliata
- prezzo

Il cliente compone il proprio ordine scegliendo prodotti dal menù tramite la GUI: in ogni istante la GUI mostra la composizione dell'ordine e il prezzo complessivo, consentendo di aggiungere/rimuovere elementi.

Il file di testo [DentBurgerList.txt](#), descritto più oltre, contiene i prodotti offerti a menù.

TEMPO STIMATO PER SVOLGERE L'INTERO COMPITO: **2h15 – 3h**

PARTE 1 – Modello dei dati: Punti 12 [TEMPO STIMATO: 55-70 minuti]

PARTE 2 – Persistenza: Punti 8 [TEMPO STIMATO: 40-50 minuti]

PARTE 3 – Grafica: Punti 10 [TEMPO STIMATO: 45-60 minuti]

JAVAFX – Parametri run configuration nei LAB

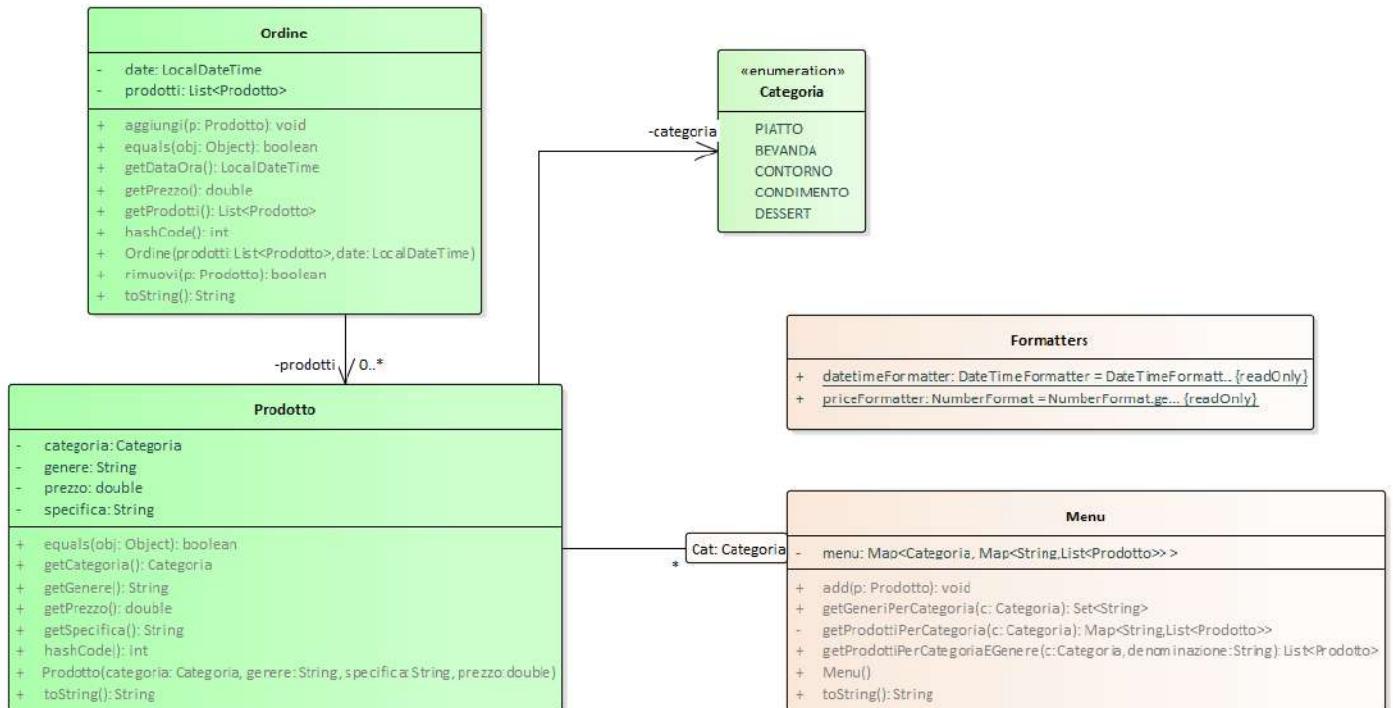
```
--module-path "C:\applicativi\moduli\javafx-sdk-19\lib"  
--add-modules javafx.controls
```

Parte 1 – Modello dei dati

(punti: 12)

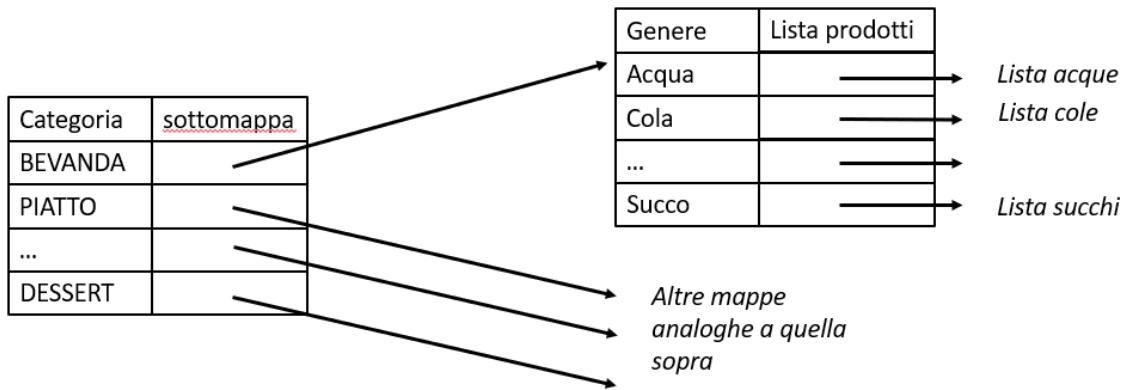
Package: dentburger.model

[TEMPO STIMATO: 55-70 minuti]



SEMANTICA:

- L'enumerativo **Categoria** (fornito) definisce le costanti corrispondenti alle categorie di prodotti previste
- La classe **Formatters** (da completare) definisce i due formattatori personalizzati usati da tutta l'applicazione: in particolare, **datetimeFormatter** è un formattatore per data e ora secondo lo standard italiano SHORT, mentre **priceFormatter** è un formattatore per prezzi in Euro secondo lo standard italiano (quindi col simbolo di valuta posizionato dopo l'importo e da esso separato da spazio hard)
- La classe **Prodotto** (fornita) rappresenta un singolo prodotto caratterizzato da **Categoria**, genere (stringa), specifica (stringa) e prezzo. Il costruttore riceve e verifica gli argomenti in ingresso, lanciando apposite **IllegalArgumentException** con adeguato messaggio d'errore in caso di incoerenze. Appositi accessori consentono di recuperare le singole proprietà. Sono incluse opportune implementazioni di **equals**, **toString** ed **hashcode**. Da notare che, intenzionalmente, due prodotti sono considerati uguali se hanno le stesse proprietà a meno del prezzo: ciò al fine di prevenire che lo stesso prodotto possa comparire a menù più volte con prezzi diversi.
- La classe **Ordine** (fornita) rappresenta un ordine inteso come lista di prodotti, effettuato in una certa data e ora. Il costruttore riceve e verifica gli argomenti in ingresso, lanciando due **IllegalArgumentException** distinte con adeguato messaggio d'errore in caso di argomenti nulli. Appositi accessori consentono di recuperare le singole proprietà. Sono incluse opportune implementazioni di **equals**, **toString** ed **hashcode**.
- La classe **Menu** (da completare nell'implementazione) rappresenta il menù del ristorante, strutturato come mappa di mappe che gestiscono liste di prodotti. Più precisamente, il menù è organizzato come mappa indicizzata per **Categoria**: a ogni categoria è associata una ulteriore mappa, indicizzata per genere (stringa), che associa a ogni genere la lista dei **Prodotto** di quella categoria e genere (vedere figura).



Il costruttore istanzia la mappa vuota: è compito del metodo `add` popolarla via via che vengono aggiunti prodotti, curando la costruzione delle mappe di secondo livello e delle liste quando necessario.

Devono essere implementati:

- Il metodo `add`, che aggiunge un nuovo prodotto nell'opportuna lista dell'opportuna mappa, se non già presente: se presente, deve lanciare **`IllegalArgumentException`** con apposito messaggio d'errore
- I due metodi `getGeneriPerCategoria` e `getProdottiPerCategoriaEGenere`, che restituiscono rispettivamente l'insieme (eventualmente vuoto) dei generi associati a una data categoria e la lista (eventualmente vuota) dei prodotti di una data categoria e genere.
NB: a questo fine può essere utile sfruttare il metodo privato `getProdottiPerCategoria` (fornito), che restituisce la sottomappa relativa alla categoria fornita come argomento.

La classe definisce i vari accessori, nonché un'opportuna `toString`.

Parte 2 – Persistenza

(punti: 8)

Package: `dentburger.persistence`

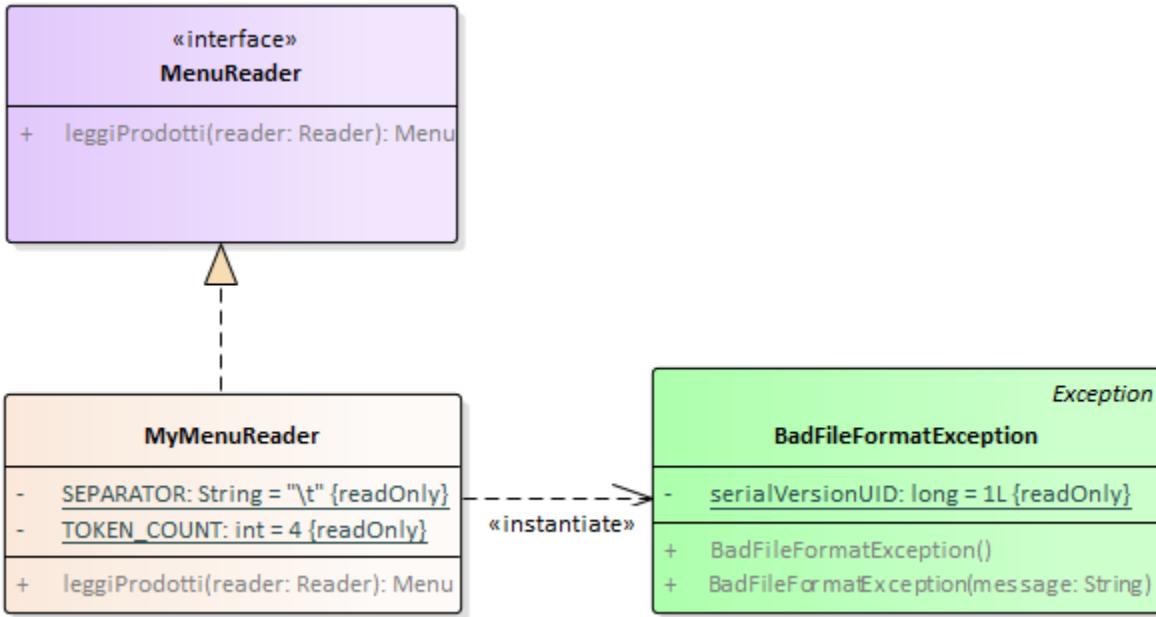
[TEMPO STIMATO: 40-50 minuti]

Il file di testo `DentBurgerList.txt` contiene i prodotti offerti a menù dal ristorante, uno per riga. Ogni riga è composta di quattro elementi separati da una o più tabulazioni:

- Categoria del prodotto (stringa scritta con un qualunque insieme di caratteri maiuscoli e/o minuscoli)
- Genere del prodotto (stringa)
- Specifica del prodotto (stringa)
- Prezzo in Euro del prodotto, formattato secondo lo standard italiano col simbolo di valuta dietro al valore numerico, separato da uno spazio hard.

ESEMPIO

Piatto	Burger	Basic	4,90 €
Piatto	Pollo	Alette	3,90 €
Piatto	Insalata	Verde	3,50 €
Piatto	Insalata	Mista	3,80 €
Piatto	Insalata	con pollo	5,20 €
Contorno	Patatine	Medie	2,70 €
Contorno	Patatine	Grandi	3,60 €
Contorno	Patatine	Formaggio	5,50 €
Bevanda	Cola	Media	3,50 €
Bevanda	Cola	Grande	4,50 €
Bevanda	Cola0	Media	3,80 €
Bevanda	Cola0	Grande	4,80 €
Bevanda	Acqua	Media	1,50 €
Bevanda	Aranciata	Media	3,60 €
Dessert	Gelato	Panna	1,30 €
Dessert	Gelato	Panna e cioccolato	1,80 €
...			



SEMANTICA:

- L'interfaccia **MenuReader** (fornita) dichiara il metodo `leggiProdotti` che carica da un apposito Reader (già aperto) i dati necessari, restituendo un'istanza di **Menu** opportunamente popolata. Il metodo lancia:
 - IllegalArgumentException** con opportuno messaggio d'errore in caso di argomento (reader) nullo;
 - BadFormatException** con messaggio d'errore appropriato in caso di problemi nel formato del file (mancanza/eccesso di elementi, categorie inesistenti, errori nel formato dei prezzi, etc.); in particolare, è richiesto di controllare con cura il formato del prezzo, verificando:
 - che non siano presenti punti decimali
 - che il numero sia letto interamente e non vi siano anomalie di sorta.
 - una **IOException** in caso di altri problemi di I/O.
- La classe **MyMenuReader** (**da realizzare**) implementa **MenuReader** secondo le specifiche sopra descritte.

Parte 3

(punti: 10)

(punti 0)

Package: dentburger.controller

Il Controller è organizzato secondo il diagramma UML seguente:



SEMANTICA:

La classe **Controller** (fornita) riceve in fase di costruzione il **Menu** del ristorante, mantenuto nel proprio stato interno unitamente all'**Ordine** del cliente, che viene gestito interamente dal **Controller** stesso.

Sono forniti metodi per recuperare tutte le informazioni utili, molti dei quali delegano l'operazione al **Menu** o all'**Ordine** interni. In particolare i tre metodi `getCategorie`, `getGeneriPerCategoria` `getProdottiPerGenereCategoria` restituiscono tre liste

utili per il successivo popolamento delle combo nella GUI, mentre i due metodi *aggiungi/rimuovi* consentono di modificare dinamicamente l'**Ordine** del cliente aggiungendo/togliendo prodotti.

Infine, il metodo statico *alert*, utilizzabile anche dal MainPane, consente di far comparire all'utente, ove occorra, una finestra di dialogo con opportuno messaggio d'errore.

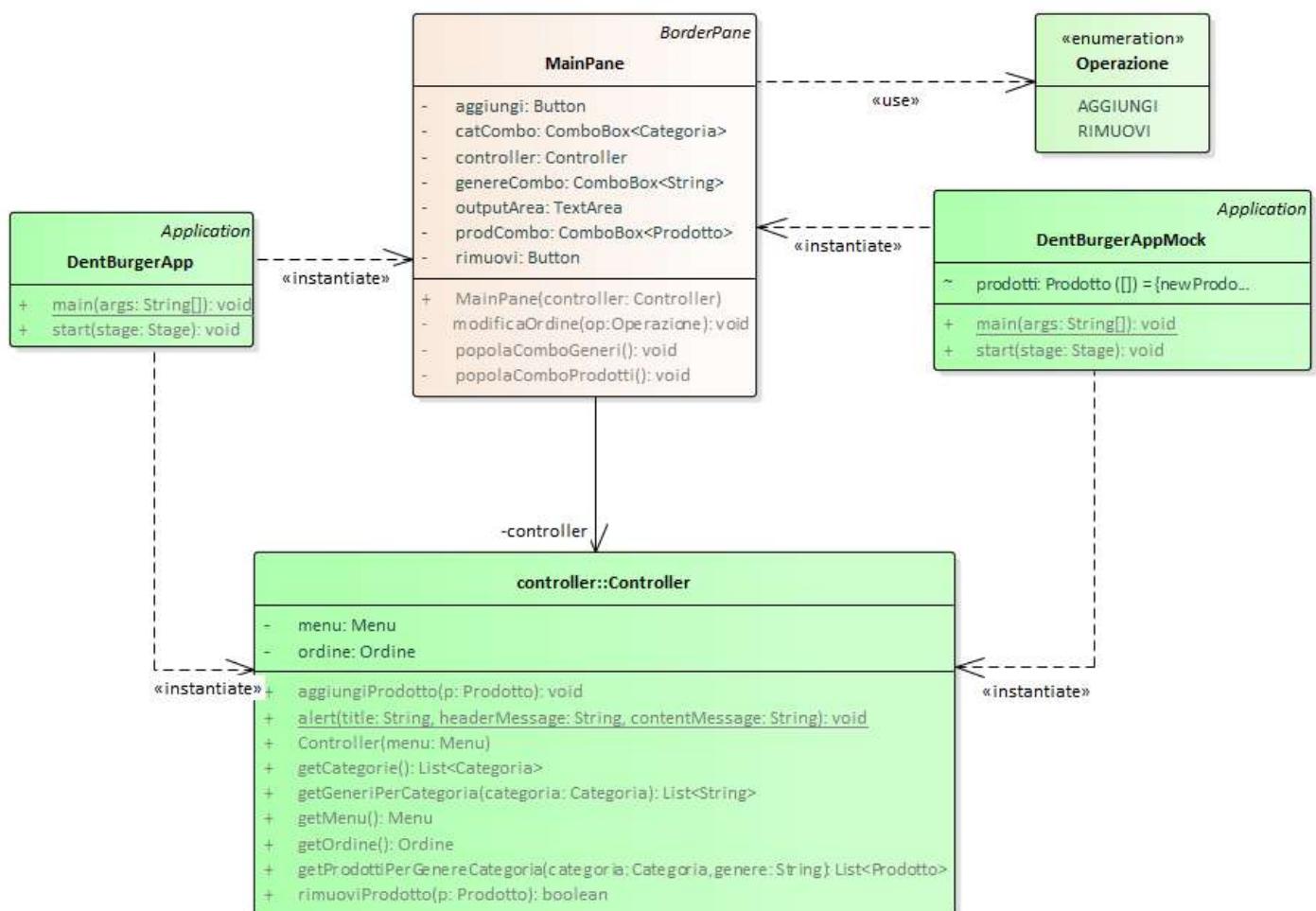
Package: dentburger.ui

[TEMPO STIMATO: 45-60 minuti] (punti 10)

La classe **DentBurgerApp** (fornita) costituisce l'applicazione JavaFX che si occupa di aprire i file, creare il controller e incorporare il **MainPane**. Per consentire di collaudare la GUI anche in assenza / in caso di malfunzionamento della parte di persistenza, è possibile avviare l'applicazione mediante la classe **DentBurgerAppMock**.

L'enumerativo ausiliario **Operazione** definisce le due costanti AGGIUNGI / RIMUOVI utili per parametrizzare l'ascoltatore degli eventi dei pulsanti.

L'interfaccia utente è illustrata nelle figure seguenti e segue il modello sotto illustrato:



L'interfaccia grafica si presenta come segue:

- a sinistra tre combo elencano rispettivamente le **Categoria** disponibili, i *generi* disponibili per la categoria selezionata e i prodotti disponibili per la categoria e il genere selezionati: pertanto, a parte la **combo delle categorie** che ha contenuto fisso, le altre due **combo devono essere popolate / ripopolate dinamicamente, in base alle selezioni delle combo precedenti**. Sotto, due pulsanti “Aggiungi” / “Rimuovi” consentono di aggiungere / togliere dall’ordine il prodotto attualmente selezionato.
- a destra, un’area di testo (inizialmente vuota e non scrivibile dall’utente) mostra in ogni momento lo stato dell’**Ordine** dell’utente, unitamente al prezzo complessivo e alla data dell’ordine stesso (quella corrente), opportunamente formattati.

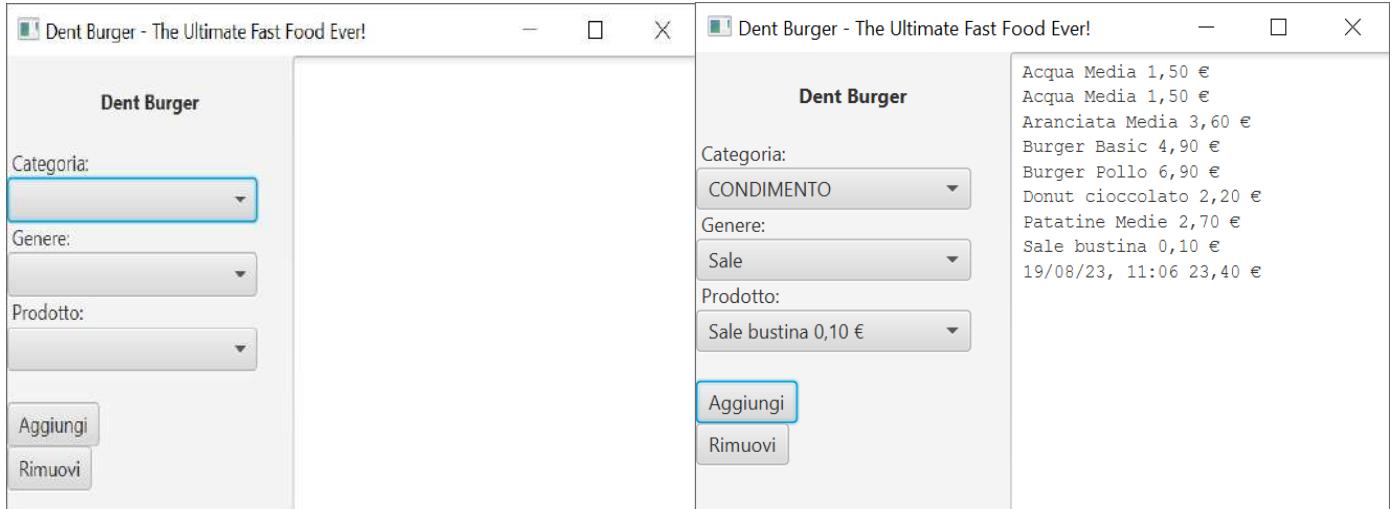


Fig. 1: a sinistra, la situazione iniziale della GUI, con tutte le combo ancora non selezionate, prima di qualunque interazione con l'utente; a destra, dopo la selezione e aggiunta di diversi prodotti.

Comportamento:

- la selezione di una **Categoria** dalla prima combo causa il popolamento automatico della seconda combo, quella dei generi, nonché la selezione automatica del primo elemento di essi; in cascata, ciò causa il popolamento della terza combo, quella dei prodotti, anche in questo caso con la selezione automatica del primo di essi.

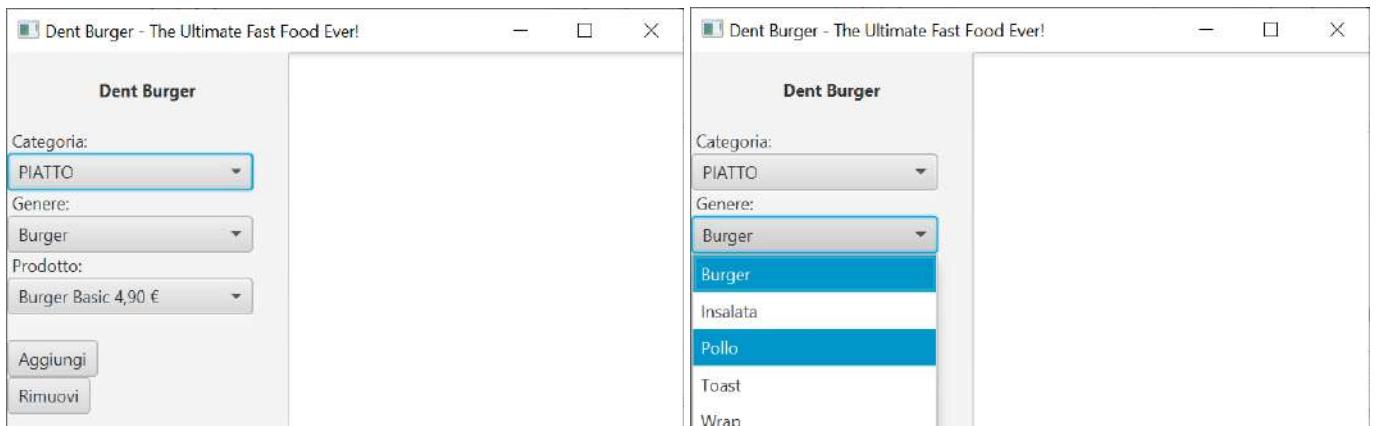


Fig. 2: a sinistra, la GUI dopo la selezione della categoria PIATTO dalla prima combo; come mostrato a destra, indipendentemente dalla selezione automatica è comunque possibile selezionare altri generi.

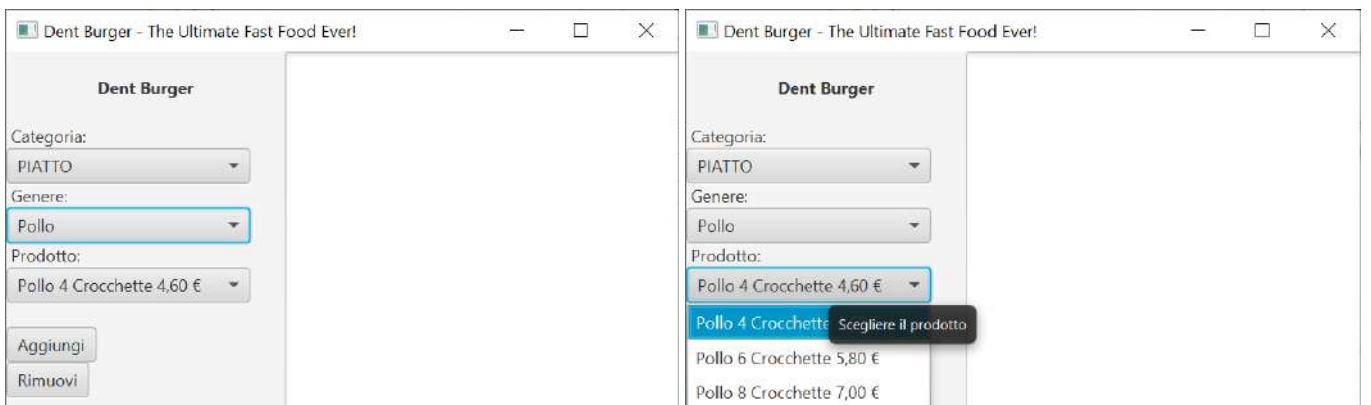


Fig. 3: a sinistra: se si seleziona un diverso genere (es. Pollo) la terza combo viene ripopolata di conseguenza con tutti i prodotti di quel genere (mostrati a destra).

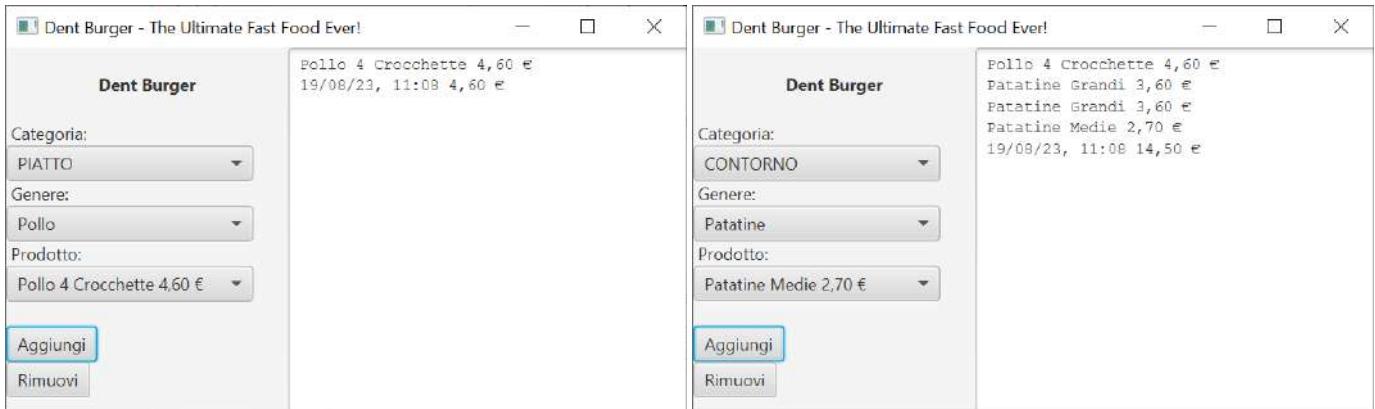


Fig. 4: a sinistra: premendo il pulsante Aggiungi, il prodotto selezionato viene incluso nell'ordine; procedendo in questo modo si possono aggiungere tutti i prodotti desiderati, anche più di una volta (a destra).

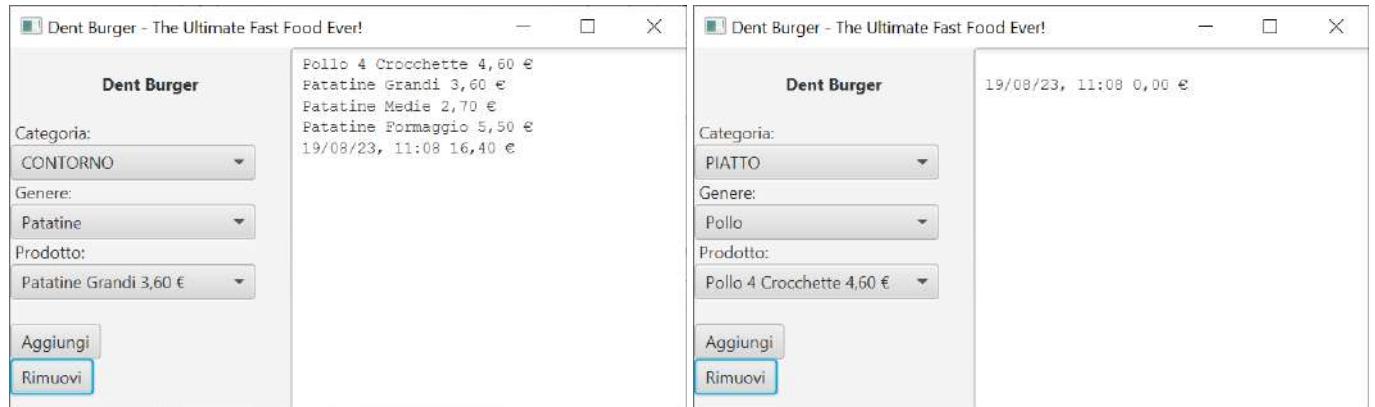


Fig. 5: a sinistra: premendo il pulsante Rimuovi è anche possibile togliere dall'ordine un prodotto precedentemente inserito (nell'esempio sopra, una delle due patatine grandi): nel caso esso non sia presente non accade nulla (non viene mostrato alcun messaggio d'errore). Da notare che il totale dell'ordine è sempre aggiornato dinamicamente.

Come caso limite (a destra), se vengono rimossi tutti i prodotti, l'ordine si azzera.

Il MainPane è fornito parzialmente realizzato: è presente quasi tutta l'impostazione strutturale, mentre sono da completare la configurazione delle combo e la gestione degli eventi.

In particolare, **MainPane** estende **BorderPane** e prevede:

- 1) a sinistra, una **VBox** per le varie combo, i pulsanti e le etichette ausiliarie
- 2) a destra, una **VBox** con la sola area di output.

La **parte da completare** riguarda:

- 1) la configurazione iniziale delle tre combo
- 2) l'aggancio dei tre ascoltatori degli eventi, rispettivamente *popolaComboGeneri* per la combo delle categorie, *popolaComboProdotti* per la combo dei generi e *modificaOrdine* per i due pulsanti Aggiungi/Rimuovi.
- 3) la logica di gestione degli eventi, incapsulata nei tre metodi privati sopra citati.

La logica di gestione degli eventi è la seguente:

- *popolaComboGeneri* recupera dalla combo delle categorie la **Categoria** selezionata (se esiste, altrimenti non fa nulla) e, sulla base di quella, recupera dal **Controller** la lista dei soli generi di interesse: se tale lista ha almeno un elemento, provvede a selezionare il primo come default nella combo. Infine, richiama *popolaComboProdotti* per propagare il popolamento conseguente alla terza combo.
- *popolaComboProdotti* recupera dalle due combo delle categorie e dei generi la **Categoria** e il genere selezionati (se esistono: altrimenti non fa nulla) e, sulla base di questi, recupera dal **Controller** la lista dei soli prodotti di interesse: come sopra, se tale lista ha almeno un elemento provvede a selezionare il primo come default nella combo.

- *modificaOrdine* recupera dalla combo prodotti il **Prodotto** attualmente selezionato (se esiste: altrimenti emette un apposito messaggio d'errore tramite la funzione *alert* del **Controller**), quindi distingue il pulsante premuto tramite l'argomento **Operazione** ricevuto e procede a invocare l'appropriato metodo del **Controller** per aggiornare l'ordine. Infine, aggiorna la visualizzazione dell'ordine – recuperato tramite l'apposito metodo del **Controller** – sull'area di output.

Cose da ricordare

- salva costantemente il tuo lavoro: l'informatica a volte può essere “subdolamente ostile”..
- in particolare: se ora compila e stai per fare modifiche, salva la versione attuale (non si sa mai)

Checklist di consegna

- Hai fatto un **JAR eseguibile**, che contenga cioè l'indicazione del main?
- Hai controllato che **si compili e ci sia tutto**? [NB: non includere il PDF del testo]
- Hai **rinominato IL PROGETTO**, lo ZIP e il JAR esattamente come richiesto?
- Hai **chiamato** la cartella del progetto esattamente come richiesto?
- **Hai fatto un unico file ZIP (NON .7z, rar o altri formati) contenente l'intero progetto?**
In particolare, ti sei assicurato di aver incluso tutti i file .java (e non solo i .class)?
- **Hai consegnato DUE file distinti, ossia lo ZIP col progetto e il JAR eseguibile?**
- Su EOL, hai **premuto** il tasto “CONFERMA” per inviare il tuo elaborato?

ESAME DI FONDAMENTI DI INFORMATICA T-2 del 26/7/2023

Proff. E. Denti – R. Calegari – A. Molesini

Tempo a disposizione: 3h30

NOME PROGETTO ECLIPSE: CognomeNome-matricola (es. RossiMario-0000123456)

NOME CARTELLA PROGETTO: CognomeNome-matricola (es. RossiMario-0000123456)

NOME ZIP DA CONSEGNARE: CognomeNome-matricola.zip (es. RossiMario-0000123456.zip)

NOME JAR DA CONSEGNARE: CognomeNome-matricola.jar (es. RossiMario-0000123456.jar)

Si devono consegnare DUE FILE: l'intero progetto Eclipse e il JAR eseguibile

Si ricorda che compiti non compilabili o palesemente lontani da 18/30 NON SARANNO CORRETTI e causeranno la verbalizzazione del giudizio “RESPINTO”

L'Ente del Turismo di Dentinia ha richiesto lo sviluppo di un'applicazione per la generazione delle previsioni meteo per una serie di località turistiche che, a partire da una serie di dati grezzi (previsioni per una specifica località e ora), produca il *bollettino meteorologico* che descriva la giornata nel suo complesso. Il bollettino dev'essere offerto in due possibili formati: *sintetico* o *dettagliato*. Il primo fa una sintesi della giornata attesa tramite una breve frase, mentre il secondo elenca anche temperatura e probabilità di pioggia a diverse ore della giornata (si vedano esempi più oltre).

DESCRIZIONE DEL DOMINIO DEL PROBLEMA

Una *previsione* indica *temperatura e probabilità di pioggia* previste per un certo luogo in una certa *data* a una certa *ora* (ad esempio: Ferrara, 18/07/22, 05:00, 18°, 91%).

Un *bollettino meteo* sintetizza *l'andamento complessivo* di una giornata per un certo *luogo* in una certa *data*: a tal fine, in formato *sintetico* riporta un breve *testo riassuntivo, temperatura media e probabilità di pioggia* previste globalmente per quel dato luogo in quella giornata, mentre nel formato *dettagliato* a queste informazioni si aggiungono anche le *previsioni disponibili per una serie di orari* (che non sono prefissati ma dipendono in generale dai dati disponibili per uno specifico luogo).

ESEMPIO DI BOLLETTINO SINTETICO

Previsioni per il giorno 12/07/22 a Bologna

Giornata con piogge diffuse, con probabilità di pioggia del 77% e temperatura media di 20°C

ESEMPIO DI BOLLETTINO DETTAGLIATO

Previsioni per il giorno 12/07/22 a Bologna

12/07/22, 00:00, Temperatura 18°, Prob. pioggia 91%

12/07/22, 02:00, Temperatura 18°, Prob. pioggia 91%

12/07/22, 05:00, Temperatura 16°, Prob. pioggia 92%

12/07/22, 08:00, Temperatura 20°, Prob. pioggia 87%

12/07/22, 11:00, Temperatura 24°, Prob. pioggia 64%

12/07/22, 14:00, Temperatura 27°, Prob. pioggia 78%

12/07/22, 17:00, Temperatura 15°, Prob. pioggia 58%

12/07/22, 20:00, Temperatura 16°, Prob. pioggia 58%

12/07/22, 23:00, Temperatura 16°, Prob. pioggia 58%

12/07/22, 23:59, Temperatura 16°, Prob. pioggia 58%

Giornata con piogge, con probabilità di pioggia del 77% e temperatura media di 20°C

Il testo riassuntivo utilizza per la giornata una delle espressioni standard “serena”, “quasi serena”, “con possibili piogge sparse”, “variabile”, “con piogge diffuse”, “con piogge” o “con piogge insistenti e generalizzate”, scegliendola in base alla probabilità di pioggia globale secondo la seguente scaletta:

- serena → probabilità di pioggia non superiore al 5%
- quasi serena → probabilità di pioggia non superiore al 10%
- con possibili piogge sparse → probabilità di pioggia non superiore al 25%
- variabile → probabilità di pioggia non superiore al 50%

- con piogge diffuse → probabilità di pioggia non superiore al 65%
- con piogge → probabilità di pioggia non superiore all'80%
- con piogge insistenti → probabilità di pioggia superiore all'80%

Temperatura media e probabilità di pioggia globali di una data giornata si ottengono calcolando una *opportuna media pesata* delle singole previsioni orarie. Più precisamente:

- se non esiste una previsione per le ore 00:00, la si crea uguale alla prima previsione disponibile per la giornata
- se non esiste una previsione per le ore 23:59, la si crea uguale all'ultima previsione disponibile per la giornata
- per ogni coppia di previsioni consecutive, si determinano temperatura e probabilità di pioggia medie: tali valori si pesano con un peso pari alla durata dell'intervallo in questione, ossia alla distanza temporale fra le due previsioni considerate.

ESEMPIO DI CALCOLO

Data la sequenza di previsioni per il giorno 12/07/22 a Bologna:

12/07/22, 02:00, Temperatura 18°, Prob. pioggia 91%

12/07/22, 05:00, Temperatura 16°, Prob. pioggia 92%

12/07/22, 08:00, Temperatura 20°, Prob. pioggia 87%

12/07/22, 11:00, Temperatura 24°, Prob. pioggia 64%

12/07/22, 14:00, Temperatura 27°, Prob. pioggia 78%

12/07/22, 17:00, Temperatura 15°, Prob. pioggia 58%

12/07/22, 20:00, Temperatura 16°, Prob. pioggia 58%

12/07/22, 23:00, Temperatura 16°, Prob. pioggia 58%

si introducono in primo luogo due previsioni extra per le ore 00:00 e 23:59, rispettivamente uguali alla prima e all'ultima della sequenza. Dopo di che si calcolano le seguenti medie:

dalle ore 00:00 alle ore 02:00 (ovvero per 02:00 ore), Temperatura 18°, Prob. pioggia 91%

dalle ore 02:00 alle ore 05:00 (ovvero per 03:00 ore), Temperatura 17°, Prob. pioggia 91.5%

dalle ore 05:00 alle ore 08:00 (ovvero per 03:00 ore), Temperatura 18°, Prob. pioggia 90.5%

dalle ore 08:00 alle ore 11:00 (ovvero per 03:00 ore), Temperatura 22°, Prob. pioggia 75.5%

dalle ore 11:00 alle ore 14:00 (ovvero per 03:00 ore), Temperatura 25.5°, Prob. pioggia 71%

dalle ore 14:00 alle ore 17:00 (ovvero per 03:00 ore), Temperatura 21°, Prob. pioggia 68%

dalle ore 17:00 alle ore 20:00 (ovvero per 03:00 ore), Temperatura 15.5°, Prob. pioggia 58%

dalle ore 20:00 alle ore 23:00 (ovvero per 03:00 ore), Temperatura 16°, Prob. pioggia 58%

dalle ore 23:00 alle ore 23:59 (ovvero per 00:59 ore), Temperatura 16°, Prob. pioggia 58%

La temperatura media e le probabilità di pioggia globali si ottengono infine semplicemente facendo la media pesata di tali valori medi, utilizzando come peso la durata dei rispettivi intervalli – ovvero:

$$T = (18^\circ \times 2\text{h} + 17^\circ \times 3\text{h} + 18^\circ \times 3\text{h} + \dots + 16^\circ \times 0\text{h}59\text{m}) / 23\text{h}59\text{m} = 19.9^\circ \rightarrow 20^\circ \text{ (arrotondamento al più vicino)}$$

$$P = (91\% \times 2\text{h} + 91.5\% \times 3\text{h} + 90.5\% \times 3\text{h} + \dots + 58\% \times 0\text{h}59\text{m}) / 23\text{h}59\text{m} = 77.1\% \rightarrow 77\% \text{ (arrotondamento al più vicino)}$$

Il file di testo [Previsioni.txt](#), descritto più oltre, contiene i dati di una serie di previsioni puntuali per diverse città.

TEMPO STIMATO PER SVOLGERE L'INTERO COMITO: 2h15 – 3h

PARTE 1 – Modello dei dati: Punti 11 [TEMPO STIMATO: 55-65 minuti]

PARTE 2 – Persistenza: Punti 13 [TEMPO STIMATO: 60-75 minuti]

PARTE 3 – Grafica: Punti 6 [TEMPO STIMATO: 20-40 minuti]

JAVAFX – Parametri run configuration nei LAB

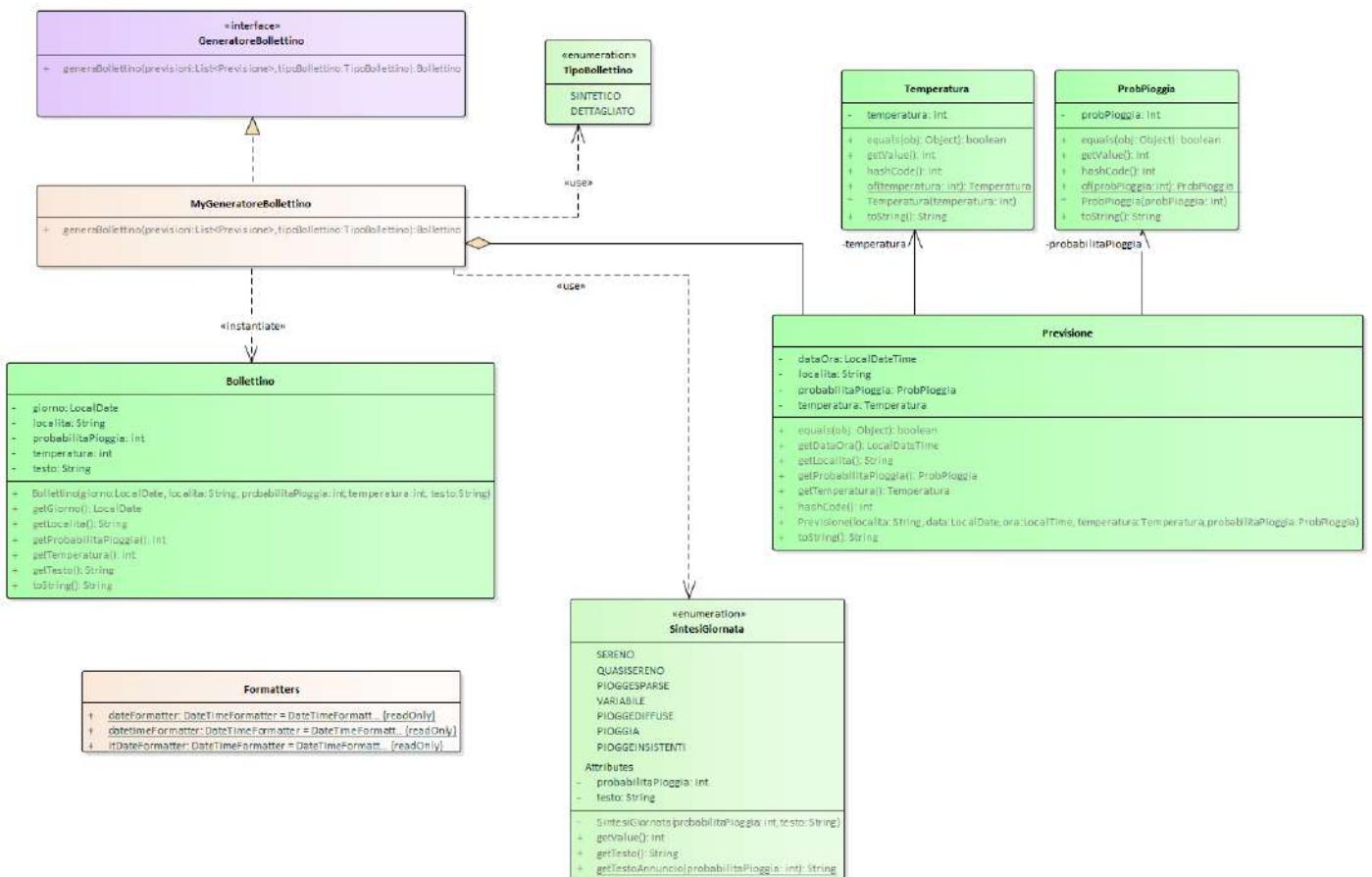
```
--module-path="C:\applicativi\moduli\javafx-sdk-19\lib"
--add-modules=javafx.controls
```

Parte 1 – Modello dei dati

(punti: 11)

Package: meteodent.model

[TEMPO STIMATO: 60-75 minuti]



SEMANTICA:

- a) L'enumerativo ***TipoBollettino*** (fornito) definisce semplicemente le due costanti DETTAGLIATO e SINTETICO
 - b) L'enumerativo ***SintesiGiornata*** (fornito) definisce le sette costanti da SERENO a PIOGGEINSISTENTI, accoppiandole ai rispettivi valori-soglia come definiti nel Dominio del Problema e alla corrispondente stringa descrittiva, recuperabili tramite i due accessori `getValue` e `getTesto`. Il metodo `getTestoAnnuncio` completa l'implementazione, restituendo una stringa compiutamente descrittiva.
 - c) Le due classi ***Temperatura*** e ***ProbPioggia*** (fornite) encapsulano ciascuna un valore intero, dandogli semantica. In ossequio al pattern factory, il costruttore non è pubblico: la costruzione deve quindi avvenire attraverso il metodo statico `of`. L'accessori `getValue` e implementazioni standard per `toString`, `equals` e `hashCode` completano l'implementazione.
 - d) La classe ***Previsione*** (fornita) rappresenta una previsione intesa come specificato nel Dominio del Problema (i suoi dati corrispondono a quelli di una riga del file da leggere nella persistenza). Il costruttore verifica gli argomenti in ingresso, lanciando ***IllegalArgumentException*** con adeguato messaggio d'errore in caso di incoerenze. Appositi accessori consentono di recuperare le singole proprietà, *ivi inclusa la data e l'ora sotto forma di LocalDateTime*. Sono incluse opportune implementazioni di `equals`, `toString` e `hashCode`.
 - e) La classe ***Bollettino*** (fornita) rappresenta un bollettino inteso come specificato nel Dominio del Problema. Il costruttore verifica gli argomenti in ingresso, lanciando ***IllegalArgumentException*** con adeguato messaggio d'errore in caso di incoerenze. Appositi accessori consentono di recuperare le singole proprietà. È inclusa un'idonea implementazione di `toString`.
 - f) La classe ***Formatters*** (da realizzare) definisce i tre formattatori usati in tutta l'applicazione: in particolare, `itDateFormatter` è un formattatore per data secondo lo standard italiano ma con l'anno su quattro cifre,

mentre `dateFormatter` e `datetimeFormatter` sono, rispettivamente, formattatori per data/ data e ora secondo lo standard italiano in formato SHORT.

- g) L'interfaccia **GeneratoreBollettino** (fornita) dichiara il metodo `generaBollettino`, che genera un **Bollettino** a partire da una lista di **Previsione** e dal **TipoBollettino** desiderato. Da specifica di progetto, il metodo deve lanciare **IllegalArgumentException** con adeguato messaggio d'errore nei seguenti casi:
- Lista previsioni vuota o nulla
 - Le previsioni nella lista non si riferiscono tutte alla stessa località
 - Le previsioni nella lista non si riferiscono tutte alla stessa data
 - Vi sono due o più previsioni per lo stesso orario
- h) La classe **MyGeneratoreBollettino** (da realizzare) implementa **GeneratoreBollettino** in ossequio alle specifiche sopra elencate e a quanto previsto dal Dominio del Problema. NB: l'arrotondamento finale della temperatura e probabilità di pioggia deve avvenire all'intero più vicino (metodo `Math.round`).

Parte 2 – Persistenza

(punti: 13)

Package: `meteodent.persistence`

[TEMPO STIMATO: 55-65 minuti]

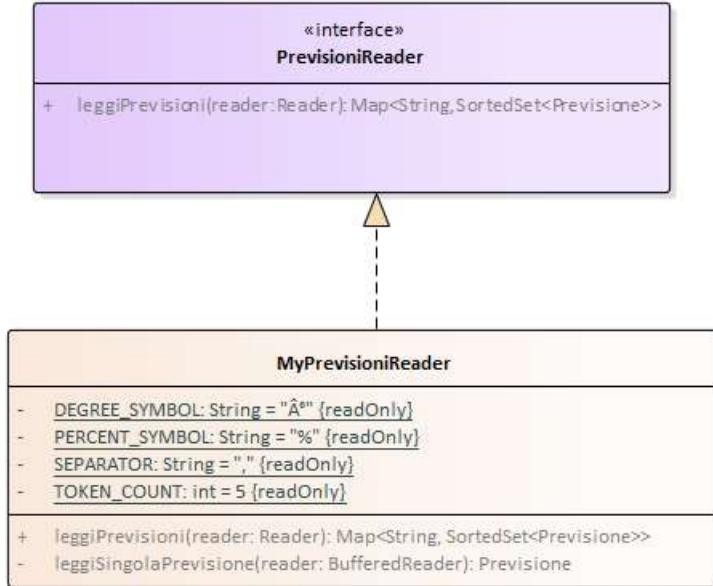
Il file di testo `previsioni.txt` contiene una serie di previsioni, una per riga, riferite a più località ed eventualmente per diverse date: in ogni riga, gli elementi sono separati da virgole.

Ogni riga specifica cinque elementi:

- la località (una stringa)
- la data a cui la previsione si riferisce, nel classico formato GG/MM/AA
- l'orario a cui la previsione si riferisce, nel classico formato HH:MM
- la temperatura prevista, costituita da un numero intero (eventualmente anche negativo) subito seguito, senza spazi intermedi, dal carattere “°”
- la probabilità di pioggia prevista, costituita da un numero intero (ovviamente compreso fra 0 e 100) subito seguito, senza spazi intermedi, dal carattere ‘%’

ESEMPIO

```
Bologna, 12/07/22, 02:00, 18°, 91%
Bologna, 12/07/22, 05:00, 16°, 92%
Bologna, 12/07/22, 08:00, 20°, 87%
Bologna, 12/07/22, 11:00, 24°, 64%
...
Ferrara, 18/07/22, 05:00, 18°, 91%
Ferrara, 18/07/22, 15:00, 16°, 92%
Ferrara, 18/07/22, 22:00, 20°, 87%
Reggio Emilia, 11/07/22, 02:00, -1°, 21%
Reggio Emilia, 11/07/22, 10:00, -1°, 32%
Reggio Emilia, 11/07/22, 18:00, 6°, 50%
...
```



SEMANTICA:

- L'interfaccia **PrevisioniReader** (fornita) dichiara il metodo *leggiPrevisioni* che carica da un apposito Reader (già aperto) i dati necessari, restituendo una mappa che associa a ogni località (*stringa*) l'insieme ordinato delle previsioni che la riguardano (*SortedSet<Previsione>*), ordinato in senso crescente per data e ora. Il metodo lancia:
 - IllegalArgumentException** con opportuno messaggio d'errore in caso di argomento (reader) nullo;
 - BadFormatException** con messaggio d'errore appropriato in caso di problemi nel formato del file (mancanza/eccesso di elementi, errori nel formato delle date/orari, percentuali assurde, formato errato nelle temperature e nelle probabilità di pioggia, etc.);
 - una **IOException** in caso di altri problemi di I/O.
- La classe **MyPrevisioniReader** (**da realizzare**) implementa **PrevisioniReader** secondo le specifiche sopra descritte.

Parte 3

(punti: 6)

Package: meteodent.controller

(punti 0)

Il Controller è organizzato secondo il diagramma UML seguente:



SEMANTICA:

La classe **Controller** (fornita) riceve in fase di costruzione la mappa delle previsioni, che viene mantenuta nel suo stato interno. È fornita un'ampia serie di metodi per recuperare tutte le informazioni utili:

- La mappa previsioni (*getMappaPrevisioni*)

- La lista delle località (`getListaLocalita`)
- l'insieme delle date per le quali esistono previsioni per una data località (`getDatePerLocalita`)
- La lista delle previsioni per una data località e un certo giorno (`getPrevisioni`)
- Il bollettino per un certo giorno, a un certo orario, in una data località (`getBollettino`)

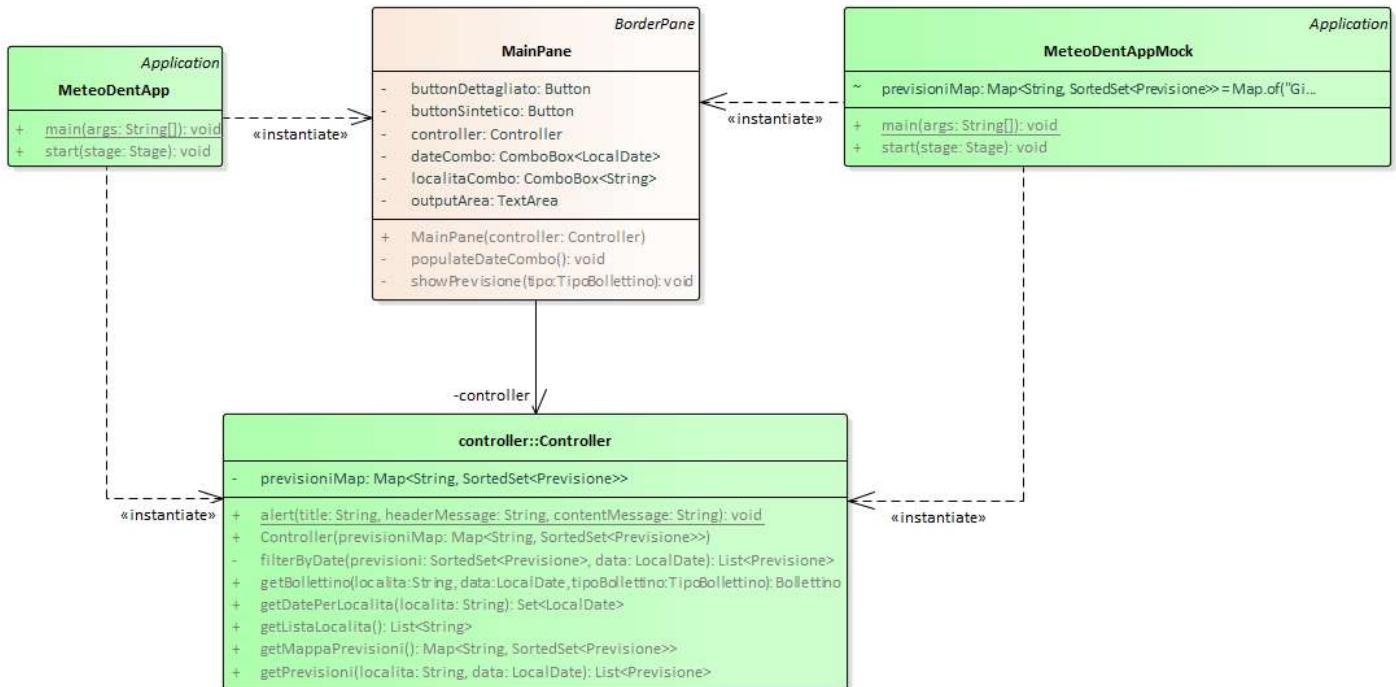
Infine, il metodo statico `alert`, utilizzabile anche dal MainPane, consente di far comparire all'utente, ove occorra, una finestra di dialogo con opportuno messaggio d'errore.

Package: meteodent.ui

[TEMPO STIMATO: 20-40 minuti] (punti 6)

La classe **MeteodentApp** (fornita) costituisce l'applicazione JavaFX che si occupa di aprire i file, creare il controller e incorporare il **MainPane**. Per consentire di collaudare la GUI anche in assenza / in caso di malfunzionamento della parte di persistenza, è possibile avviare l'applicazione mediante la classe **MeteodentAppMock**.

L'interfaccia utente è illustrata nelle figure seguenti e segue il modello sotto illustrato:



L'interfaccia grafica si presenta come segue:

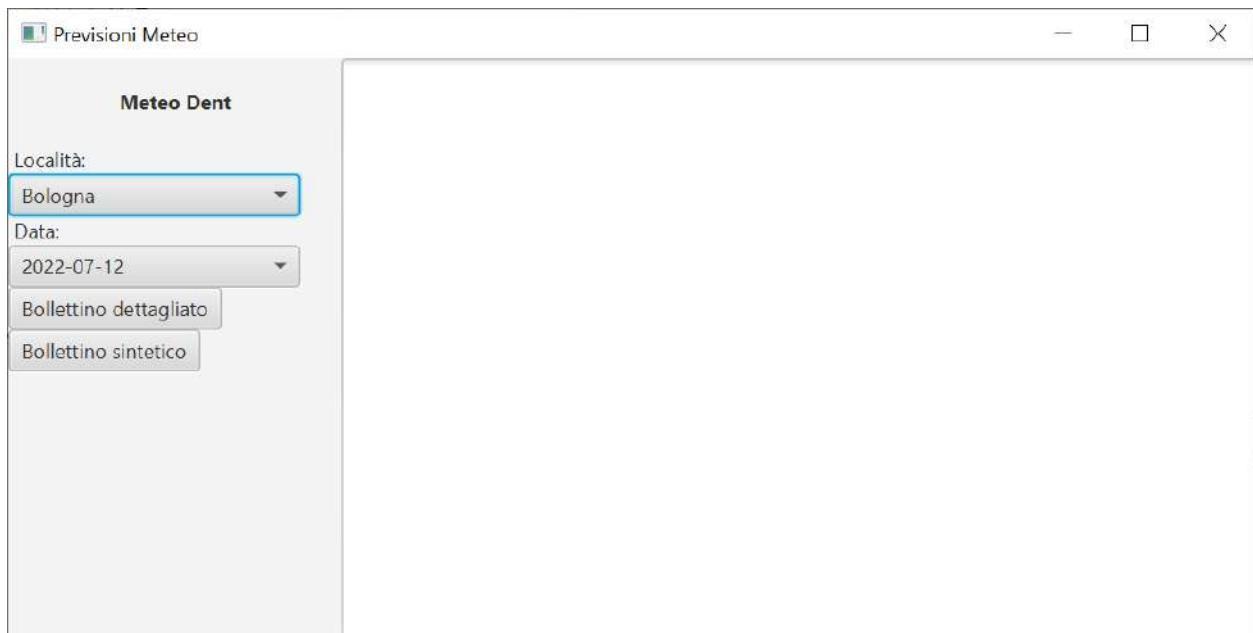
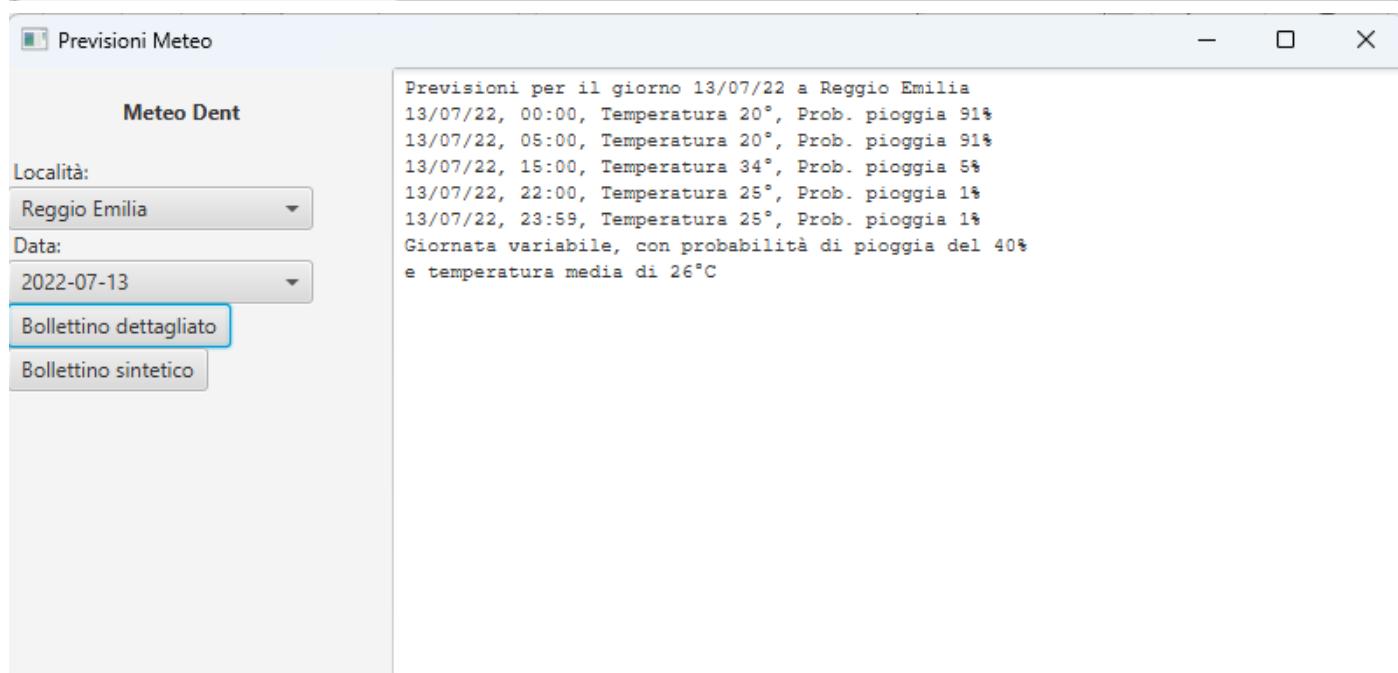
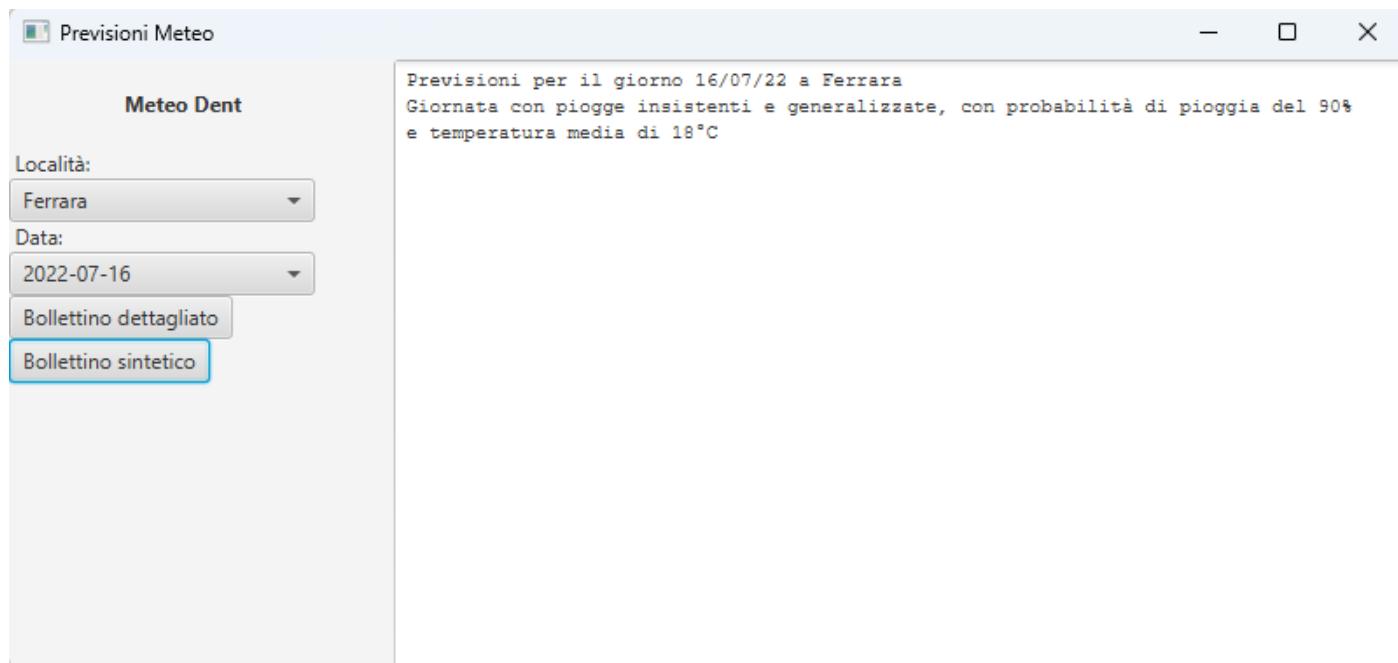


Fig. 1: la situazione iniziale della GUI, con combo tipologie vuota, prima di qualunque interazione con l'utente.

- a sinistra una combo elenca tutte le località disponibili: di seguito, un'altra combo, *popolata dinamicamente in base alla località selezionata nella prima*, elenca le date per le quali esistono previsioni per la località scelta; subito sotto, due pulsanti consentono di produrre e visualizzare il bollettino, nei due formati possibili
- a destra, un'area di testo (inizialmente vuota e non scrivibile dall'utente) mostra i risultati.

Comportamento:

- la selezione di una località dalla prima combo deve causare il ri-popolamento della seconda combo con tutte e sole le date per le quali esistono previsioni per la località scelta
- la scelta di una di tali date dalla seconda combo NON produce effetti immediati
- la pressione di uno dei due pulsanti causa invece immediatamente la generazione del bollettino richiesto e la sua visualizzazione nell'area a lato.



Figg. 2 / 3.

Il MainPane è fornito parzialmente realizzato: è presente quasi tutta l'impostazione strutturale, mentre sono da completare la configurazione di alcuni componenti e la gestione degli eventi.

In particolare, **MainPane** estende **BorderPane** e prevede:

- 1) a sinistra, una **VBox** per le varie combo, etichette e pulsanti
- 2) a destra, una **VBox** con la sola area di output.

La **parte da completare** riguarda:

- 1) l'aggancio dei gestori eventi rispettivamente alla prima combo e ai due pulsanti
- 2) il popolamento della combo delle date (metodo privato *populateDateCombo*), che deve recuperare la località di interesse, recuperare tramite il controller le date che la riguardano, e preimpostare il primo item come selezione di default
- 3) la logica di gestione dell'evento (metodo privato *showPrevisione*), che deve recuperare dalle due combo i dati necessari e far generare al **Controller** il **Bollettino** del tipo richiesto, mostrandone l'output sull'area.

Cose da ricordare

- salva costantemente il tuo lavoro: l'informatica a volte può essere "subdolamente ostile"..\
- in particolare: se ora compila e stai per fare modifiche, salva la versione attuale (non si sa mai)

Checklist di consegna

- Hai fatto un **JAR eseguibile**, che contenga cioè l'indicazione del main?
- Hai controllato che **si compili e ci sia tutto**? [NB: non includere il PDF del testo]
- Hai **rinominato IL PROGETTO**, lo ZIP e il JAR esattamente come richiesto?
- Hai **chiamato** la cartella del progetto esattamente come richiesto?
- **Hai fatto un unico file ZIP (NON .7z, rar o altri formati) contenente l'intero progetto?**
In particolare, ti sei assicurato di aver incluso tutti i file .java (e non solo i .class)?
- **Hai consegnato DUE file distinti, ossia lo ZIP col progetto e il JAR eseguibile?**
- Su EOL, hai **premuto** il tasto "CONFERMA" per inviare il tuo elaborato?

ESAME DI FONDAMENTI DI INFORMATICA T-2 del 5/7/2023

Proff. E. Denti – R. Calegari – A. Molesini

Tempo a disposizione: 3h30

NOME PROGETTO ECLIPSE: CognomeNome-matricola (es. RossiMario-0000123456)

NOME CARTELLA PROGETTO: CognomeNome-matricola (es. RossiMario-0000123456)

NOME ZIP DA CONSEGNARE: CognomeNome-matricola.zip (es. RossiMario-0000123456.zip)

NOME JAR DA CONSEGNARE: CognomeNome-matricola.jar (es. RossiMario-0000123456.jar)

Si devono consegnare DUE FILE: *l'intero progetto Eclipse e il JAR eseguibile*

Si ricorda che compiti *non compilabili o palesemente lontani da 18/30* NON SARANNO CORRETTI e causeranno la verbalizzazione del giudizio “RESPINTO”

L'Associazione Consumatori di Dentinia ha richiesto lo sviluppo di un'applicazione che agevoli l'analisi delle spese sanitarie, separandole per tipologia e indicando quelle detraibili e quelle non detraibili.

DESCRIZIONE DEL DOMINIO DEL PROBLEMA

Fra i suoi molti servizi, l'Associazione Consumatori di Dentinia assiste anche gli utenti nell'analisi delle loro spese sanitarie, che si distinguono in diverse *tipologie*:

- 1) spese per farmaci
- 2) spese per dispositivi medici
- 3) spese per ticket
- 4) spese per libera professione (visite mediche, interventi di specialistica, etc.)
- 5) altre spese, diverse dalle precedenti.

Le prime quattro tipologie sono *dutraibili ai fini fiscali*, mentre l'ultima non lo è.

Un *documento di spesa* può comprendere più *voci di spesa*, ciascuna con la propria tipologia.

Alla fine dell'anno, il Governatorato di Dentinia rende disponibile a ciascun cittadino, su un apposito file di testo, l'elenco di tutte le spese fatte nell'anno, specificando per ciascuna:

- la data in cui ogni documento di spesa è stato emesso
- l'indicazione dell'emittente (farmacia, medico, AUSL, etc.) di tale documento
- l'importo totale di tale documento
- l'elenco delle singole voci di spesa dettagliate in tale documento, ciascuna con la relativa tipologia e importo

Dovrà quindi essere possibile analizzare tali informazioni, precisamente:

- mostrando il totale delle spese effettuate
- calcolando separatamente il totale delle spese detraibili e di quelle non detraibili
- mostrando, a richiesta, il dettaglio delle singole spese di una data categoria a scelta dell'utente (ad esempio, solo quella per farmaci, solo quella per ticket, etc.)

Il file di testo [spesesanitarie.txt](#), descritto più oltre, contiene i dati di una serie di documenti di spesa.

TEMPO STIMATO PER SVOLGERE L'INTERO COMPITO: 2h15 – 3h

PARTE 1 – Modello dei dati: Punti 13 [TEMPO STIMATO: 60-75 minuti]

PARTE 2 – Persistenza: Punti 11 [TEMPO STIMATO: 55-65 minuti]

PARTE 3 – Grafica: Punti 6 [TEMPO STIMATO: 20-40 minuti]

JAVAFX – Parametri run configuration nei LAB

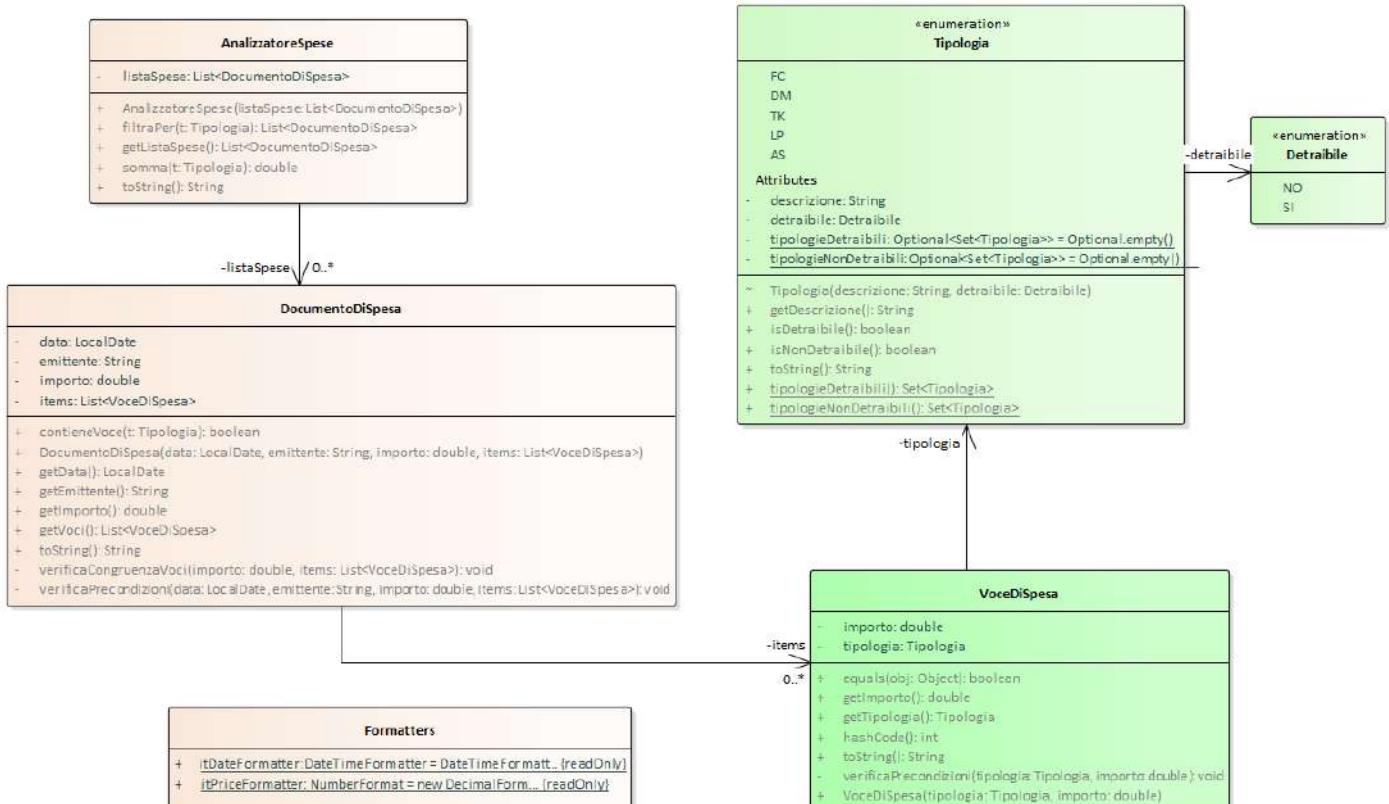
```
--module-path "C:\applicativi\moduli\javafx-sdk-19\lib"  
--add-modules javafx.controls
```

Parte 1 – Modello dei dati

Package: spesesanitarie.model

(punti: 13)

[TEMPO STIMATO: 60-75 minuti]



SEMANTICA:

- L'enumerativo **Detraibile** (fornito) definisce semplicemente due costanti Sì/No
- L'enumerativo **Tipologia** (fornito) definisce le costanti corrispondenti ai tipi di spese definiti nel dominio del problema: a ciascuna sono associate una descrizione testuale e una proprietà booleana che specifica se tale tipologia di spesa sia detraibile o meno. La coppia di metodi **isDetraibile** / **isNonDetraibile** consente di verificare tale proprietà, mentre la coppia di metodi **statici tipologieDetraibili** / **tipologieNonDetraibili** restituisce l'insieme delle tipologie che, rispettivamente, sono / non sono detraibili.
- La classe **Formatters** (da realizzare) definisce i due formattatori personalizzati usati da tutta l'applicazione: in particolare, **itDateFormatter** è un formattatore per data secondo lo standard italiano ma con l'anno su quattro cifre, mentre **itPriceFormatter** è un formattatore per prezzi in Euro col simbolo di valuta davanti al prezzo anziché dietro, che usa sempre esattamente due decimali dopo la virgola.
- La classe **VoceDiSpesa** (fornita) definisce una voce di spesa di uno scontrino o fattura, caratterizzata da tipologia e importo. Il costruttore verifica gli argomenti in ingresso, lanciando **IllegalArgumentException** con adeguato messaggio d'errore in caso di incoerenze. Appositi accessori consentono di recuperare le singole proprietà. Sono incluse opportune implementazioni di **equals**, **toString** ed **hashcode**.
- La classe **DocumentoDiSpesa** (da completare nelle verifiche di precondizioni del costruttore e nel metodo **toString**) rappresenta un documento di spesa (scontrino, fattura, ricevuta fiscale, etc.), costituito da un elenco di una o più **VoceDiSpesa**; il documento è caratterizzato altresì dalla data di emissione, dalla descrizione dell'emittente e dall'importo totale del documento stesso. Il costruttore riceve perciò tali argomenti (nell'ordine: data, emittente, importo totale e lista di voci), recuperabili tramite appositi accessori. Devono essere implementate le seguenti verifiche nel costruttore:

- Verifiche di precondizioni: che nessun argomento sia null, che la lista contenga almeno un elemento, e che l'importo sia un numero reale finito e non negativo
- Verifiche di congruenza: che la somma delle voci di spesa sia uguale all'importo

La classe definisce i vari accessori, nonché un'opportuna `toString`

- f) La classe ***AnalizzatoreSpese*** (da completare) analizza una collezione di documenti di spesa. Il costruttore riceve una lista di ***DocumentoDiSpesa***, recuperabile tramite appositi accessori. **Devono essere implementati:**
- il metodo `somma`, che restituisce il totale delle spese effettuate per la ***Tipologia*** specificata;
 - il metodo `filtraPer`, che restituisce la lista dei soli ***DocumentoDiSpesa*** che contengono almeno una spesa della ***Tipologia*** specificata.

Parte 2 – Persistenza

(punti: 11)

Package: `spesesanitarie.persistence`

[TEMPO STIMATO: 55-65 minuti]

Il file di testo `spesesanitarie.txt` contiene i dati di una serie di documenti di spesa, uno dietro l'altro. Ogni documento di spesa è organizzato su più righe: in tutte, gli elementi sono separati dal carattere “punto e virgola” (‘;’).

La prima riga specifica quattro elementi:

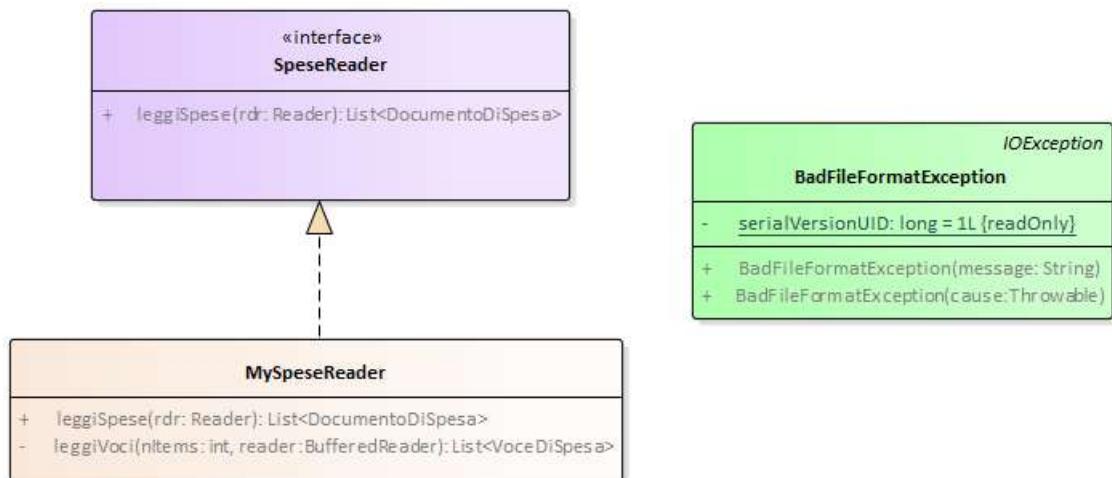
- la data di emissione del documento, nel formato GG/MM/AAAA;
- l'emittente della prestazione (una stringa);
- il numero (intero) di voci di cui il documento stesso è composto, non inferiore a 1;
- il totale in Euro del documento stesso, formattato col simbolo di valuta *davanti* al valore numerico, separato da uno spazio; a sua volta, il valore numerico è espresso secondo le convenzioni italiane.

Le righe successive (in numero uguale al numero di voci precedentemente specificato) riportano ciascuna:

- la tipologia della spesa (una sigla di due lettere identica alle costanti dell'enumerativo ***Tipologia***);
- il relativo importo in Euro, formattato come sopra.

ESEMPIO

```
25/01/2022;Dentista;1;€ 300,00
LP;€ 300,00
08/02/2022;Farmacia;4;€ 43,98
FC;€ 10,77
FC;€ 6,03
FC;€ 19,62
FC;€ 7,56
...
```



SEMANTICA:

- a) L'interfaccia **SpeseReader** (fornita) dichiara il metodo *leggiSpese* che carica da un apposito Reader (già aperto) i dati necessari, restituendo una lista di **DocumentoDiSpesa** che non è mai nulla. Il metodo lancia:
- **IllegalArgumentException** con opportuno messaggio d'errore in caso di argomento (reader) nullo;
 - **BadFormatException** con messaggio d'errore appropriato in caso di problemi nel formato del file (mancanza/eccesso di elementi, errori nel formato delle date o dei prezzi, etc.);
 - una **IOException** in caso di altri problemi di I/O.
- b) La classe **MySpeseReader** (da realizzare) implementa **SpeseReader** secondo le specifiche sopra descritte.

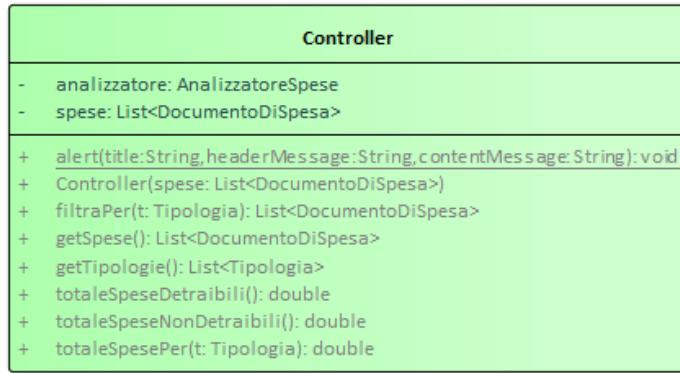
Parte 3

(punti: 6)

Package: spesesanitarie.controller

(punti 0)

Il Controller è organizzato secondo il diagramma UML seguente:



SEMANTICA:

La classe **Controller** (fornita) riceve in fase di costruzione la lista dei **DocumentoDiSpesa**, che viene mantenuta nel suo stato interno unitamente all'**AnalizzatoreSpese** che si occuperà di analizzarla.

È fornita un'ampia serie di metodi per recuperare tutte le informazioni utili, molti dei quali delegano

l'operazione all'analizzatore interno. In particolare *getSpese* restituisce la lista ricevuta dal costruttore, *getTipologie* restituisce la lista delle costanti definite nell'enumerativo **Tipologia**, *getTotaleSpesePer* restituisce il totale delle spese della **Tipologia** specificata, *filtraPer* restituisce la lista dei soli **DocumentoDiSpesa** che contengono almeno una voce di spesa della **Tipologia** specificata.

Inoltre, i due metodi *totaleSpeseDetraibili* / *totaleSpeseNonDetraibili* restituiscono, rispettivamente, il totale delle spese di tipologie detraibili / non detraibili.

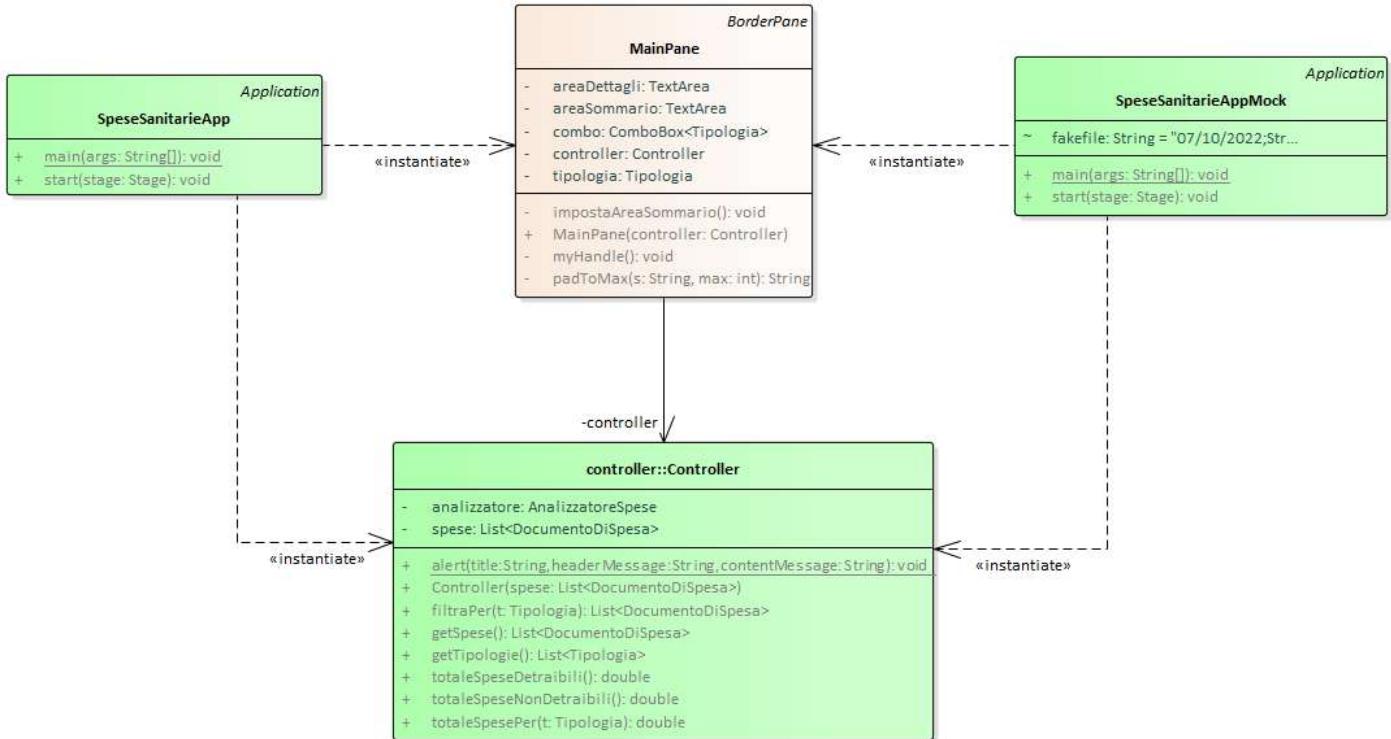
Infine, il metodo statico *alert*, utilizzabile anche dal **MainPane**, consente di far comparire all'utente, ove occorra, una finestra di dialogo con opportuno messaggio d'errore.

Package: spesesanitarie.ui

[TEMPO STIMATO: 20-40 minuti] (punti 6)

La classe **SpeseSanitarieApp** (fornita) costituisce l'applicazione JavaFX che si occupa di aprire i file, creare il controller e incorporare il **MainPane**. Per consentire di collaudare la GUI anche in assenza / in caso di malfunzionamento della parte di persistenza, è possibile avviare l'applicazione mediante la classe **SpeseSanitarieAppMock**.

L'interfaccia utente è illustrata nelle figure seguenti e segue il modello sotto illustrato:



L'interfaccia grafica si presenta come segue:

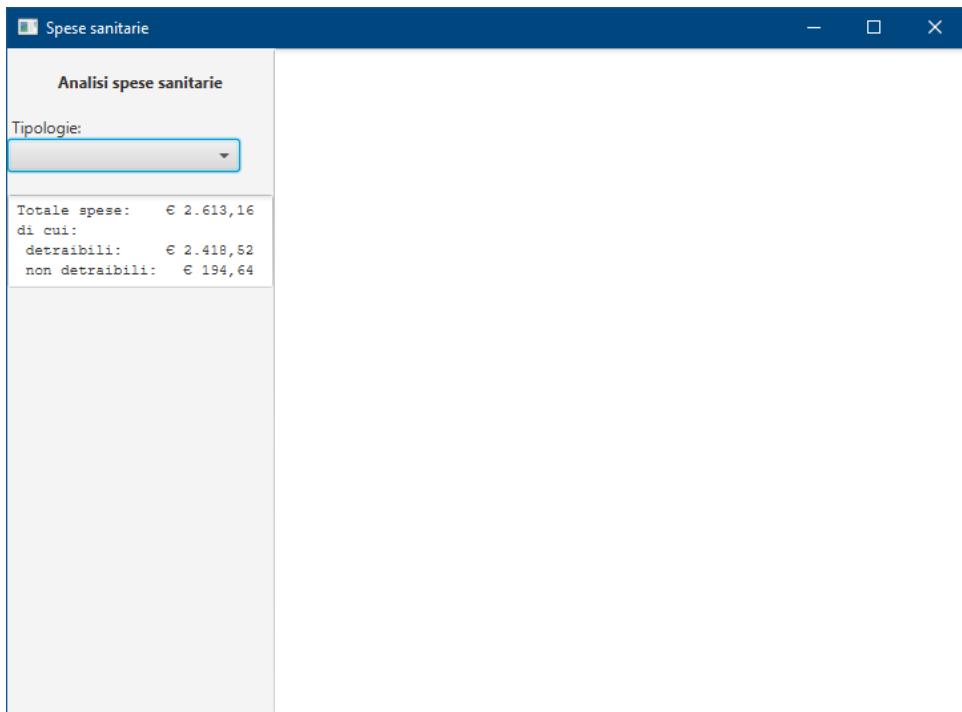


Fig. 1: la situazione iniziale della GUI, con combo tipologie vuota, prima di qualunque interazione con l'utente.

- a sinistra una combo elenca tutte le **Tipologia** disponibili; subito sotto, una piccola area di testo (non scrivibile dall'utente) riporta il totale delle spese, nonché i sub-totali di quelle detraibili/non detraibili;
- a destra, un'area di testo (inizialmente vuota e non scrivibile dall'utente) filtra i soli documenti di spesa che contengono almeno una voce della **Tipologia** selezionata.

Comportamento:

- la selezione di una **Tipologia** dalla combo deve causare immediatamente la visualizzazione dei soli documenti di spesa richiesti (Figg. 2,3), preceduta dal totale delle relative spese, con opportuni messaggi

Spese sanitarie

Analisi spese sanitarie	
Tipologie:	Totale spese per questa tipologia: € 748,64
FC - Farmaco	Dettaglio spese che contengono voci per questa tipologia:
Total spese: € 2.613,16	03/01/2022 Farmacia € 25,71, di cui: Farmaco € 4,98 Farmaco € 8,04 Farmaco € 4,41 Farmaco € 8,28
di cui: detraibili: € 2.418,52 non detraibili: € 194,64	18/01/2022 Farmacia € 16,65, di cui: Farmaco € 16,65
	08/02/2022 Farmacia € 43,98, di cui: Farmaco € 10,77 Farmaco € 6,03 Farmaco € 19,62 Farmaco € 7,56
	16/02/2022 Farmacia € 44,95, di cui: Farmaco € 3,15 Farmaco € 8,64 Farmaco € 4,41 Farmaco € 16,65 Altre spese € 12,10
	23/02/2022 Farmacia € 37,31, di cui: Farmaco € 9,83 Ticket € 3,00 Farmaco € 9,81 Farmaco € 8,64 Farmaco € 6,03
	26/02/2022 Farmacia € 11,00, di cui: Farmaco € 4,30 Farmaco € 6,70
	01/03/2022 Farmacia € 20,86, di cui: Altre spese € 0,76

Spese sanitarie

Analisi spese sanitarie	
Tipologie:	Totale spese per questa tipologia: € 1.465,00
LP - Prestaz. medica	Dettaglio spese che contengono voci per questa tipologia:
Total spese: € 2.613,16	25/01/2022 Dentista € 300,00, di cui: Prestaz. medica € 300,00
di cui: detraibili: € 2.418,52 non detraibili: € 194,64	25/02/2022 AUSL € 132,00, di cui: Prestaz. medica € 132,00
	01/03/2022 AUSL € 142,00, di cui: Prestaz. medica € 142,00
	19/04/2022 AUSL € 122,00, di cui: Prestaz. medica € 122,00
	25/05/2022 Struttura privata € 140,00, di cui: Prestaz. medica € 140,00
	28/06/2022 Medico € 175,00, di cui: Prestaz. medica € 175,00
	19/08/2022 AUSL € 122,00, di cui: Prestaz. medica € 122,00
	06/10/2022 Medico € 30,00, di cui: Prestaz. medica € 30,00
	07/10/2022 Struttura privata € 130,00, di cui: Prestaz. medica € 130,00
	21/10/2022 Medico € 100,00, di cui: Prestaz. medica € 100,00
	22/11/2022 AUSL € 72,00, di cui: Prestaz. medica € 72,00

Figg. 2 / 3.

Il MainPane è fornito parzialmente realizzato: è presente quasi tutta l'impostazione strutturale, mentre sono da completare la configurazione di alcuni componenti e la gestione degli eventi.

In particolare, **MainPane** estende **BorderPane** e prevede:

- 1) a sinistra, una **VBox** per le varie combo, aree di testo ed etichette ausiliarie
- 2) a destra, una **VBox** con la sola area di output.

La parte da completare riguarda:

- 1) la configurazione iniziale della combo
- 2) la configurazione dell'area di testo riassuntiva (metodo privato *impostaAreaSommario*)
- 3) l'aggancio dell'ascoltatore degli eventi, rappresentato dal metodo *myHandle*
- 4) la logica di gestione dell'evento, incapsulata nel metodo privato *myHandle*

In particolare, la gestione dell'evento principale, tramite *myHandle*, deve:

- recuperare dalla combo la tipologia selezionata e, sulla base di quella, recuperare l'elenco filtrato dei soli documenti di spesa di interesse, per le successive operazioni
- calcolare, tramite il **Controller**, il totale delle spese per tale tipologia
- emettere nell'area di testo il totale ora calcolato (con adeguato messaggio) e, subito sotto, l'elenco dei documenti di spesa corrispondenti, *opportunamente formattati*.

Cose da ricordare

- salva costantemente il tuo lavoro: l'informatica a volte può essere “subdolmente ostile”..
- in particolare: se ora compila e stai per fare modifiche, salva la versione attuale (non si sa mai)

Checklist di consegna

- Hai fatto un **JAR eseguibile**, che contenga cioè l'indicazione del main?
- Hai controllato che **si compili e ci sia tutto?** [NB: non includere il PDF del testo]
- Hai **rinominato IL PROGETTO**, lo ZIP e il JAR esattamente come richiesto?
- Hai **chiamato** la cartella del progetto esattamente come richiesto?
- **Hai fatto un unico file ZIP (NON .7z, rar o altri formati)** contenente l'intero progetto?
In particolare, ti sei assicurato di aver incluso tutti i file .java (e non solo i .class)?
- **Hai consegnato DUE file distinti, ossia lo ZIP col progetto e il JAR eseguibile?**
- Su EOL, hai premuto il tasto “CONFERMA” per inviare il tuo elaborato?

ESAME DI FONDAMENTI DI INFORMATICA T-2 del 12/6/2023

Proff. E. Denti – R. Calegari – A. Molesini

Tempo a disposizione: 3h30

NOME PROGETTO ECLIPSE: CognomeNome-matricola (es. RossiMario-0000123456)

NOME CARTELLA PROGETTO: CognomeNome-matricola (es. RossiMario-0000123456)

NOME ZIP DA CONSEGNARE: CognomeNome-matricola.zip (es. RossiMario-0000123456.zip)

NOME JAR DA CONSEGNARE: CognomeNome-matricola.jar (es. RossiMario-0000123456.jar)

Si devono consegnare DUE FILE: *l'intero progetto Eclipse e il JAR eseguibile*

Si ricorda che compiti *non compilabili o palesemente lontani da 18/30* NON SARANNO CORRETTI e causeranno la verbalizzazione del giudizio “RESPINTO”

L’Associazione Consumatori di Dentinia ha richiesto lo sviluppo di un’applicazione che calcoli il ritardo di un treno secondo diversi criteri, per poter poi scegliere il più adeguato alle proprie rilevazioni di qualità del servizio.

L’applicazione deve offrire all’utente la possibilità di operare secondo tre diversi criteri:

- considerare solo il ritardo (in minuti) alla stazione finale del percorso del treno;
- considerare la media dei ritardi (in minuti) in ogni stazione del percorso del treno;
- calcolare la percentuale di stazioni in cui il treno è stato “puntuale” lungo il suo percorso.

DESCRIZIONE DEL DOMINIO DEL PROBLEMA

A Dentinia, purtroppo, a volte i treni viaggiano in ritardo: per questo motivo, l’Associazione Consumatori desidera poter confrontare diversi possibili criteri per valutare la qualità del servizio offerto. A tal fine, occorre innanzitutto etichettare ogni treno con una “misura” della sua qualità, che può essere definita in modi diversi:

- 1) considerando solo il ritardo (in minuti) alla stazione finale del percorso del treno;
- 2) considerando la media dei ritardi (in minuti) in ogni stazione del percorso del treno;
- 3) calcolando, invece, la percentuale di stazioni in cui il treno è stato “puntuale” lungo il suo percorso, intendendosi per “puntuale” che è giunto in quella stazione *entro una certa soglia* (in minuti) rispetto all’orario previsto.

Il viaggio di ogni treno è monitorato da un sistema automatico, che salva su un apposito file (di testo) l’elenco delle stazioni servite, riportando per ciascuna sia l’orario previsto, sia quello effettivo, di arrivo e partenza del treno.

ESEMPIO: la tabella seguente riporta i dati di monitoraggio di un possibile treno

Secondo il criterio 1), il treno risulta giunto a destinazione con 14 minuti di ritardo.

Secondo il criterio 2), calcolando i ritardi in arrivo in ogni stazione e facendo la media, risultano invece ben 24 minuti di ritardo.

Secondi il criterio 3), infine, occorre in primis stabilire la soglia entro cui considerare un treno “puntuale”: assumendo 5 minuti, il treno in questione risulta in ritardo nel 100% delle stazioni e lo stesso accade assumendo una soglia di 10 o 15 minuti; assumendo invece 20 minuti, risulta in ritardo nel 56% delle stazioni, che si riducono al 6% assumendo 40 minuti; e così via.

stazione	arrivo previsto	arrivo effettivo	partenza prevista	partenza effettiva
MILANO CENTRALE	--	--	13:20	13:54
MILANO LAMBRATE	13:26	13:59	13:27	14:01
MILANO ROGOREDO	13:31	14:05	13:32	14:07
LODI	13:46	14:21	13:47	14:22
CASALPUSTERLENGO	13:57	14:34	13:58	14:36
PIACENZA	14:12	14:45	14:14	14:48
FIORENZUOLA	14:26	14:58	14:27	14:59
FIDENZA	14:35	15:06	14:37	15:07
PARMÀ	14:58	15:18	15:00	15:19
SANT'ILARIO	15:07	15:25	15:08	15:27
REGGIO EMILIA	15:18	15:36	15:20	15:38
RUBIERA	15:27	15:43	15:28	15:45
MODENA	15:36	15:54	15:38	15:57
CASTELFRANCO EMILIA	15:45	16:02	15:46	16:04
SAMOGGIA	15:51	16:09	15:52	16:09
ANZOLA	15:56	16:13	15:57	16:14
BOLOGNA CENTRALE	16:10	16:24	--	--

Ogni file di testo contiene i dati di monitoraggio di uno specifico treno (ne sono forniti due).

TEMPO STIMATO PER SVOLGERE L'INTERO COMPITO:

2h15 – 3h

PARTE 1 – Modello dei dati: Punti 14

[TEMPO STIMATO: 60-80 minuti]

PARTE 2 – Persistenza: Punti 7

[TEMPO STIMATO: 35-45 minuti]

PARTE 3 – Grafica: Punti 9

[TEMPO STIMATO: 40-55 minuti]

JAVAFX – Parametri run configuration nei LAB

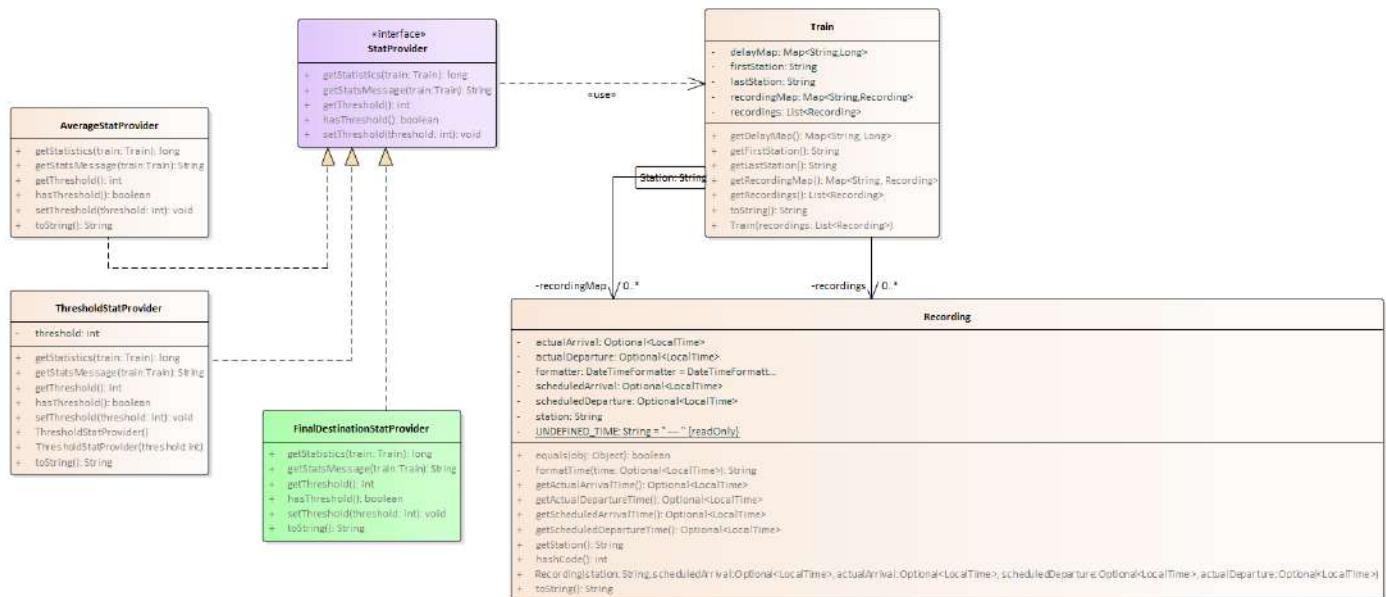
```
--module-path "C:\applicativi\moduli\javafx-sdk-19\lib"
--add-modules javafx.controls
```

Parte 1 – Modello dei dati

(punti: 13)

Package: trainstats.model

[TEMPO STIMATO: 60-70 minuti]



SEMANTICA:

- La classe **Recording** (da completare nelle verifiche di precondizioni del costruttore) rappresenta una rilevazione del treno in una data stazione: a tal fine il costruttore riceve il *nome della stazione*, l'*orario di arrivo previsto*, l'*orario di arrivo effettivo*, l'*orario di partenza previsto* e l'*orario di partenza effettivo*. Poiché gli orari, in alcune stazioni, potrebbero non essere tutti presenti, gli argomenti sono tutti **Optional<LocalTime>**. È compito del costruttore verificare che
 - nessuno di essi sia null
 - gli optional empty siano solo per le coppie di orari previste (ossia, o entrambi gli orari di arrivo, o entrambi quelli di partenza): ogni altro uso di optional empty dev'essere considerato illegale
 - l'orario di partenza previsto non sia antecedente all'orario di arrivo previsto
 - l'orario di partenza reale non sia antecedente né all'orario di arrivo previsto, né all'orario di partenza previsto.

La classe definisce i vari accessori, nonché un'opportuna **toString** e idonee definizioni di **equals** e **hashCode**.

- La classe **Train** (da completare nel costruttore) rappresenta un treno inteso come sequenza di rilevazioni: tale sequenza non può essere nulla e, poiché il percorso minimo va da una stazione iniziale a una finale, deve

sempre contenere almeno due stazioni. È compito del costruttore non solo verificare gli argomenti, ma anche predisporre tutte le strutture dati, come di seguito specificato:

- **una mappa <String, Recording>**, restituita poi dal metodo `getRecordingMap`, che associa a ogni stazione la corrispondente rilevazione;
- **una mappa <String, Long>**, restituita poi dal metodo `getDelayMap`, che associa a ogni stazione il corrispondente ritardo all'arrivo.

IMPORTANTE: se il treno è arrivato in anticipo, il ritardo va considerato zero (mai negativo!)

La classe definisce inoltre vari accessori, nonché un'opportuna `toString`; in particolare, i due metodi `getFirstStation` e `getLastStation` restituiscono il nome rispettivamente della stazione iniziale e finale.

- c) L'interfaccia **StatProvider** (fornita) dichiara i metodi per comunicare con qualunque fornitore di statistiche. Il metodo principale è `getStatistics` che, dato un **Train**, restituisce il valore della corrispondente misura sotto forma di intero long. Il metodo `getStatsMessage` è analogo ma restituisce il risultato all'intero di un'apposita stringa descrittiva. Gli altri tre metodi servono a gestire l'eventuale soglia, per quei fornitori che supportano tale concetto: in particolare, il metodo `hasThreshold` restituisce true se quel certo provider supporta il concetto di soglia, false altrimenti. Nel caso la supporti, la coppia di metodi `setThreshold / getThreshold` permette di impostare/recuperare la soglia in minuti; ove invece quel certo provider non supporti tale concetto, i due metodi dovranno lanciare **UnsupportedOperationException**.
- d) La classe **FinalDestinationStatProvider** (fornita) implementa **StatProvider** nel caso del criterio 1) del Dominio del Problema, ossia quello in cui il ritardo da considerare sia quello di arrivo alla stazione finale.
- e) Le due classi **AverageStatProvider** e **ThresholdStatProvider (da completare)** implementano **StatProvider** nei casi dei criteri 2) e 3) del Dominio del Problema, ovvero quelli in cui, rispettivamente, si debba considerare la media dei ritardi di arrivo o, al contrario, la percentuale di arrivo "puntuali" (ossia entro soglia).
Per entrambe, il metodo da implementare è solo `getStatistics`.

Parte 2 – Persistenza

Package: `trainstats.persistence`

(punti: 8)

[TEMPO STIMATO: 35-45 minuti]

Una serie di file di testo contiene in ciascun file i dati di monitoraggio di un singolo treno.

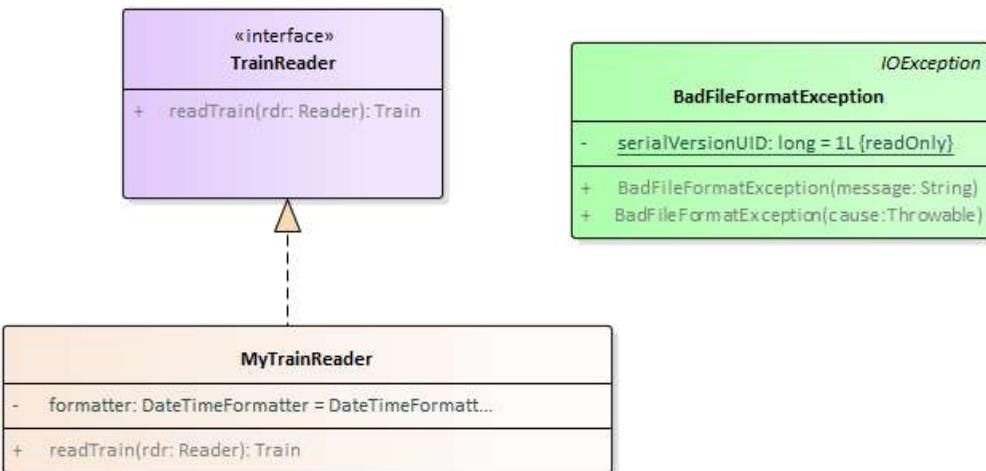
NB: il candidato non deve preoccuparsi del fatto che vi siano più file da leggere, in quanto tale aspetto è già gestito nel main (fornito): la classe da sviluppare deve leggere, come sempre, un singolo file.

Ogni riga descrive la fermata del treno in una stazione, specificando nell'ordine il *nome della stazione*, *l'orario di arrivo previsto*, *l'orario di arrivo effettivo*, *l'orario di partenza previsto* e *l'orario di partenza effettivo*. Questi valori sono separati uno dall'altro da uno o più caratteri "punto e virgola" (';'): non è dato sapere se vi siano spazi intorno, prima, davanti ai vari campi. Tutti gli orari sono in formati italiano SHORT.

Nel caso della stazione iniziale, naturalmente, gli orari di arrivo sono indefiniti; lo stesso accade nell'ultima stazione per gli orari di partenza. Al loro posto, per convenzione, è indicata una sequenza di almeno due trattini.

ESEMPIO

```
MILANO CENTRALE;-- ;-- ;13:20;13:54
MILANO LAMBRATE;13:26;13:59;13:27;14:01
MILANO ROGOREDO;13:31;14:05;13:32;14:07
...
ANZOLA ;15:56;16:13;15:57;16:14
BOLOGNA C.LE;16:10;16:24;-- ;--
```



SEMANTICA:

- a) L'interfaccia **TrainReader** (fornita) dichiara il metodo `readTrain` che carica da un apposito Reader (già aperto) i dati necessari, restituendo un **Train** perfettamente configurato. Il metodo lancia:
- **IllegalArgumentException** con opportuno messaggio d'errore in caso di argomento (reader) nullo;
 - **BadFileNotFoundException** con messaggio d'errore appropriato in caso di problemi nel formato del file (mancanza/eccesso di elementi, errori nel formato degli orari, etc.).
NB: a questo riguardo, si tenga conto che il costruttore di **Recording** lancia **IllegalArgumentException** in una serie di situazioni relative ad argomenti "illogici" (v. dettagli alle pagine precedenti): tale eccezione dovrà pertanto essere sostituita da un'opportuna **BadFileNotFoundException**.
 - una **IOException** in caso di altri problemi di I/O.
- b) La classe **MyTrainReader** (**da realizzare**) implementa **TrainReader** secondo le specifiche sopra descritte.

Parte 3

(punti: 9)

Package: trainstats.controller

(punti 0)

Il Controller è organizzato secondo il diagramma UML seguente:



SEMANTICA:

La classe **Controller** (fornita) riceve in fase di costruzione la mappa dei treni disponibili (la cui chiave è il codice univoco del treno, ottenuto a sua volta dal nome del file) e la mappa dei fornitori di statistica disponibili (la chiave è il loro nome univoco). Al suo interno mantiene l'informazione su quale **StatProvider** sia correntemente selezionato.

È ovviamente fornita un'ampia serie di metodi per recuperare tutte le informazioni utili, nonché per impostare lo **StatProvider** corrente e recuperarlo.

La terna di metodi `hasActiveThreshold / setThreshold / getThreshold` consente, rispettivamente, di sapere se lo **StatProvider** corrente abbia il concetto di soglia e, nel caso, di impostarla/recuperarla.

StatProvider corrente abbia il concetto di soglia e, nel caso, di impostarla/recuperarla.

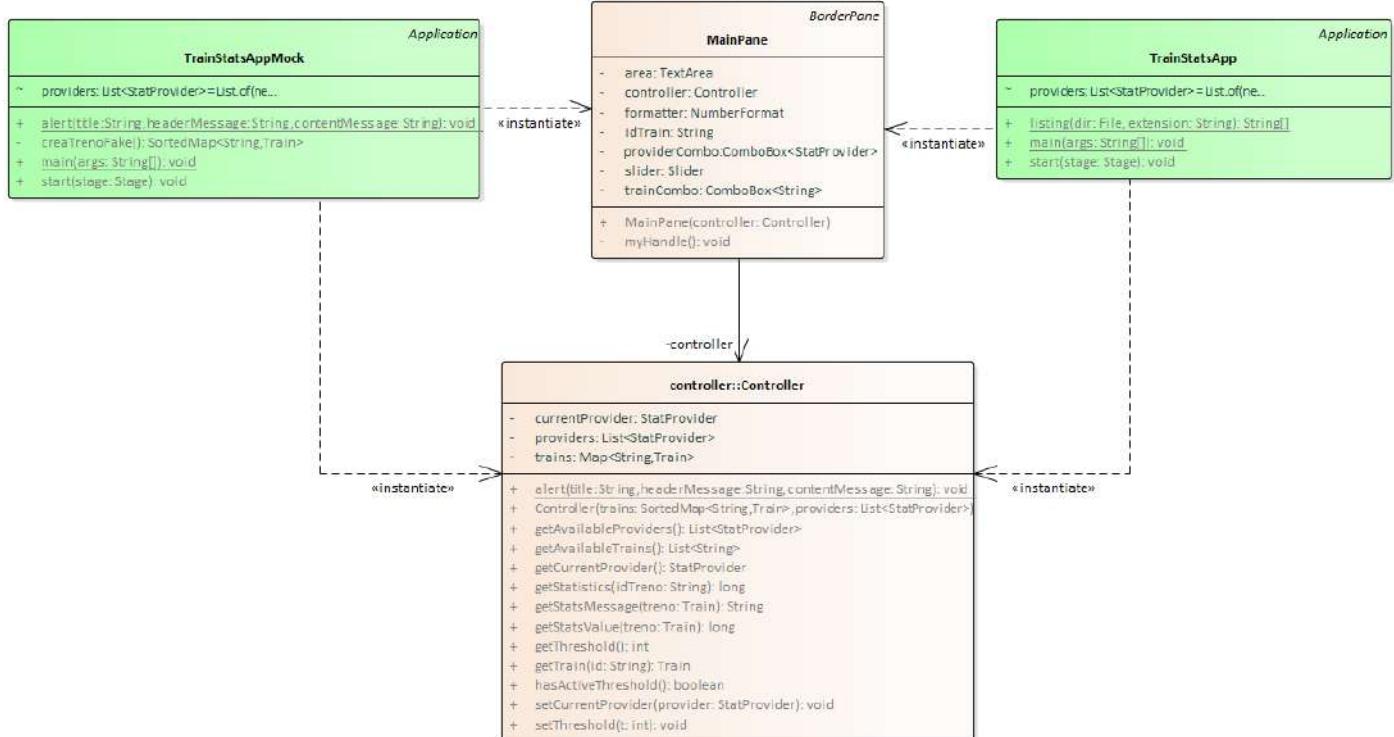
Infine, il metodo statico `alert`, utilizzabile anche dal MainPane, consente di far comparire all'utente, ove occorra, una finestra di dialogo con opportuno messaggio d'errore.

Package: trainstats.ui

[TEMPO STIMATO: 40-55 minuti] (punti 9)

La classe ***TrainStatsApp*** (fornita) costituisce l'applicazione JavaFX che si occupa di aprire i file, creare il controller e incorporare il ***MainPane***. Per consentire di collaudare la GUI anche in assenza / in caso di malfunzionamento della parte di persistenza, è possibile avviare l'applicazione mediante la classe ***TrainStatsAppMock***.

L'interfaccia utente è illustrata nelle figure seguenti e segue il modello sotto illustrato:



L'interfaccia grafica si presenta come segue:

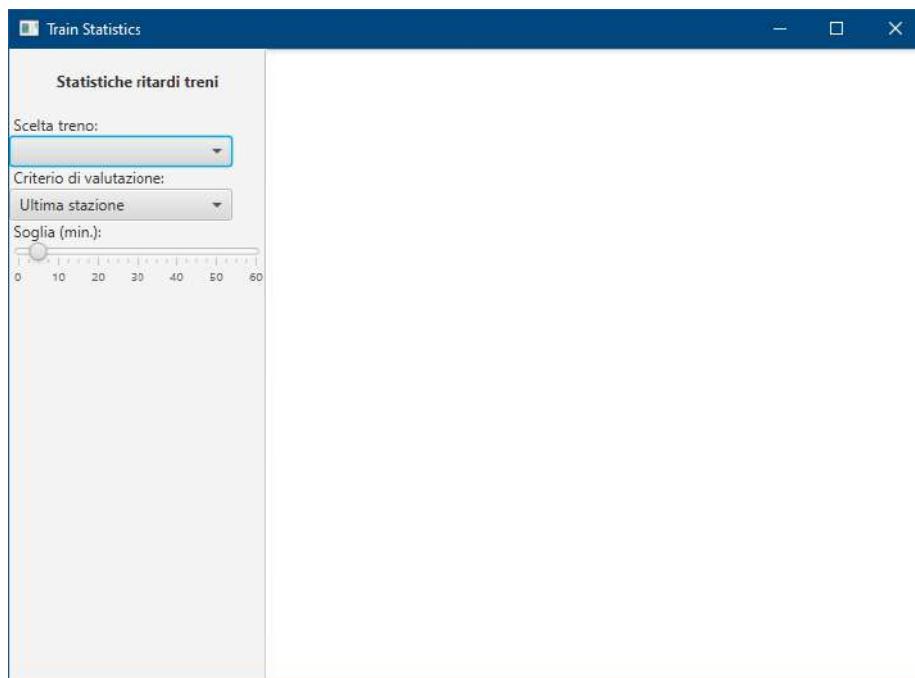
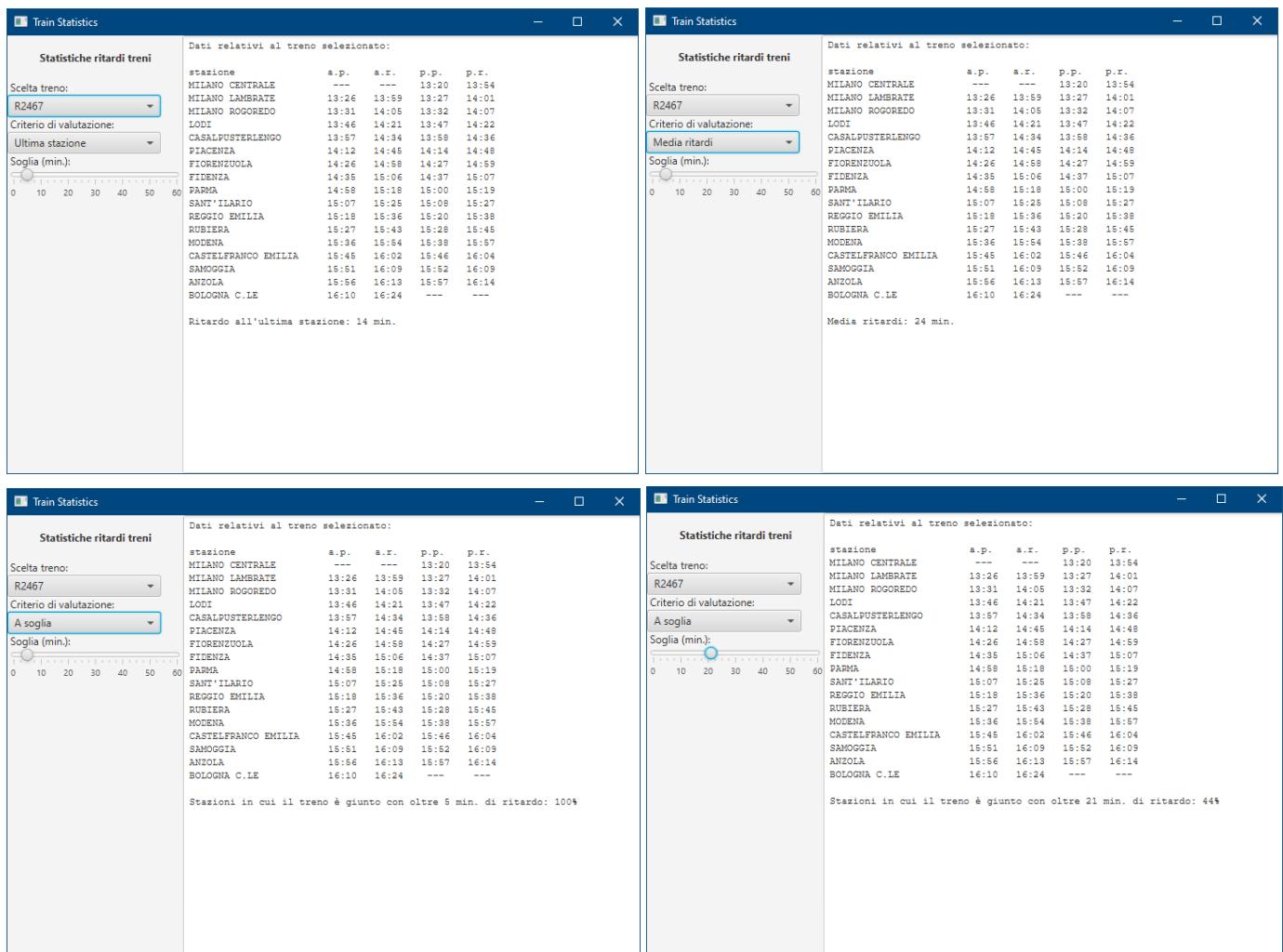


Fig. 1: la situazione iniziale della GUI, con combo treno vuota, criterio di valutazione di default e soglia 5 min.

- a sinistra c'è il pannello comandi, costituito da due combo (una con i codici dei treni, l'altra con i vari criteri di valutazione, ossia i vari **StatProvider** disponibili) e uno **Slider** (per impostare, se applicabile, la soglia, compresa fra 0 e 60 minuti);
- a destra, un'area di testo (inizialmente vuota e non scrivibile dall'utente) mostra i dettagli del treno scelto e i risultati della statistica calcolata dal provider prescelto.

Comportamento:

- la selezione di un treno dalla combo deve causare immediatamente il calcolo del ritardo col provider selezionato (inizialmente quello di default, ossia il criterio del ritardo valutato sull'ultima stazione) (Fig. 2)
- la selezione di un diverso provider dalla rispettiva combo deve causare immediatamente il ri-calcolo del ritardo secondo il nuovo criterio scelto (Fig. 3)
- la modifica della soglia dallo slider deve causare anch'essa il ri-calcolo del ritardo secondo il criterio attualmente selezionato: ovviamente, se il provider corrente non supporta il concetto di soglia, il risultato non cambierà, mentre se lo supporta si vedranno valori percentuali via via decrescenti all'aumentare della soglia (Figg. 4,5).



Figg. 2 / 3 / 4 / 5.

Il MainPane è fornito parzialmente realizzato: è presente quasi tutta l'impostazione strutturale, mentre sono da completare la configurazione di alcuni componenti e la gestione degli eventi.

In particolare, **MainPane** estende **BorderPane** e prevede:

- 1) a sinistra, una **VBox** per le varie combo, slider ed etichette ausiliarie
- 2) a destra, una **VBox** con la sola area di output.

La **parte da completare** riguarda:

- 1) la configurazione iniziale delle due combo e dello slider
- 2) l'aggancio dell'ascoltatore degli eventi (unico per i tre componenti), rappresentato dal metodo `myHandle`
- 3) la logica di gestione dell'evento, encapsulata nel metodo privato `myHandle`

In particolare, la gestione dell'evento principale, tramite `myHandle`, deve:

- recuperare dalla combo treni il nome del treno scelto e, sulla base di quello, recuperare l'oggetto **Train** corrispondente per le successive operazioni
- impostare come provider corrente nel **Controller** lo **StatProvider** selezionato nell'apposita combo
- recuperare dallo **Slider** il valore della soglia desiderata e impostarla nel **Controller**
- emettere nell'area di testo i dati del treno e, subito sotto, l'esito del calcolo della corrispondente statistica (metodo `getStatsMessage` del **Controller**).

Cose da ricordare

- salva costantemente il tuo lavoro: l'informatica a volte può essere "subdolamente ostile"...
- in particolare: se ora compila e stai per fare modifiche, salva la versione attuale (non si sa mai)

Checklist di consegna

- Hai fatto un JAR eseguibile, che contenga cioè l'indicazione del main?
- Hai controllato che **si compili e ci sia tutto?** [NB: non includere il PDF del testo]
- Hai **rinominato IL PROGETTO**, lo ZIP e il JAR esattamente come richiesto?
- Hai **chiamato** la cartella del progetto esattamente come richiesto?
- **Hai fatto un unico file ZIP (NON .7z, rar o altri formati) contenente l'intero progetto?**
In particolare, ti sei assicurato di aver incluso tutti i file .java (e non solo i .class)?
- **Hai consegnato DUE file distinti, ossia lo ZIP col progetto e il JAR eseguibile?**
- Su EOL, hai **premuto** il tasto "CONFERMA" per inviare il tuo elaborato?

ESAME DI FONDAMENTI DI INFORMATICA T-2 dell'8/2/2023

Proff. E. Denti – R. Calegari – A. Molesini

Tempo a disposizione: 3h30

NOME PROGETTO ECLIPSE: CognomeNome-matricola (es. RossiMario-0000123456)

NOME CARTELLA PROGETTO: CognomeNome-matricola (es. RossiMario-0000123456)

NOME ZIP DA CONSEGNARE: CognomeNome-matricola.zip (es. RossiMario-0000123456.zip)

NOME JAR DA CONSEGNARE: CognomeNome-matricola.jar (es. RossiMario-0000123456.jar)

Si devono consegnare DUE FILE: *l'intero progetto Eclipse e il JAR eseguibile*

Si ricorda che compiti non compilabili o palesemente lontani da 18/30 NON SARANNO CORRETTI e causeranno la verbalizzazione del giudizio “RESPINTO”

La valuta dei maghi, coniata dalla Gringott Bank, è composta da tre tipi di monete: **Galeoni** d'oro (*galleons* nell'originale inglese), **Falci** d'argento (*sickles* nell'originale inglese) e **Zellini** in bronzo (*knuts* nell'originale inglese). Un **Galeone** è suddiviso in **17 Falci**, ognuno dei quali è a sua volta suddiviso in **29 Zellini**.

Obiettivo dell'applicazione è progettare uno sportello automatico che eroghi solo monete Gringott, rispettando per ogni prelievo un massimale prestabilito per ogni utente. Per agevolare l'interazione con gli umani non-maghi, lo sportello deve altresì offrire la possibilità di esprimere gli importi nelle principali valute umane (Dollari, Euro o Sterline), il cui tasso di cambio con i Galeoni è fisso e prestabilito (1 galeone = 5 sterline = 6 dollari = 5.66 euro).

DESCRIZIONE DEL DOMINIO DEL PROBLEMA

Per effettuare un prelievo si può specificare l'ammontare desiderato o nella valuta dei maghi o in una delle valute umane supportate (Dollari, Euro o Sterline). Lo sportello rispetta le seguenti regole:

- 1) Solo gli utenti preventivamente autorizzati, elencati nell'apposito file, possono compiere prelievi
- 2) L'importo è erogabile solo se non supera il massimale per ogni singola operazione specificato nell'apposito file
- 3) L'importo in monete Gringott deve dare il massimo numero di galeoni possibile, garantendo tuttavia che siano erogati sempre almeno 5 falci d'argento.
- 4) Nel caso di importi specificati in Dollari, Euro o Sterline, deve essere usato il tasso di cambio fisso prestabilito.

ESEMPI DI APPLICAZIONE DELLE REGOLE 3 & 4:

- Richiesti 20 galeoni → Erogati 19 galeoni + 17 falci
- Richiesti 19 galeoni + 20 zellini → Erogati 18 galeoni + 17 falci + 20 zellini
- Richieste 500 sterline → (1 galeone = £ 5.00) [=100 galeoni teorici] → Erogati 99 galeoni +17 falci
- Richiesti 500 euro → (1 galeone = € 5.66) Erogati 88 galeoni + 5 falci + 22 zellini (importo effettivo: 499,67 €)
- Richiesti 500 dollari → (1 galeone = \$ 6.00) Erogati 83 galeoni + 5 falci + 19 zellini (importo effettivo: 499,67 €)

Trattandosi di una valuta di maghi, si presuppone che la disponibilità di monete sia illimitata.

Il file di testo `ceilings.txt` contiene i massimali permessi per ogni utente per ogni operazione di prelievo.

TEMPO STIMATO PER SVOLGERE L'INTERO COMPITO: 2h15 – 3h

PARTE 1 – Modello dei dati: Punti 13 [TEMPO STIMATO: 45-60 minuti]

PARTE 2 – Persistenza: Punti 11 [TEMPO STIMATO: 60-80 minuti]

PARTE 3 – Grafica: Punti 6 [TEMPO STIMATO: 30-40 minuti]

JAVAFX – Parametri run configuration nei LAB

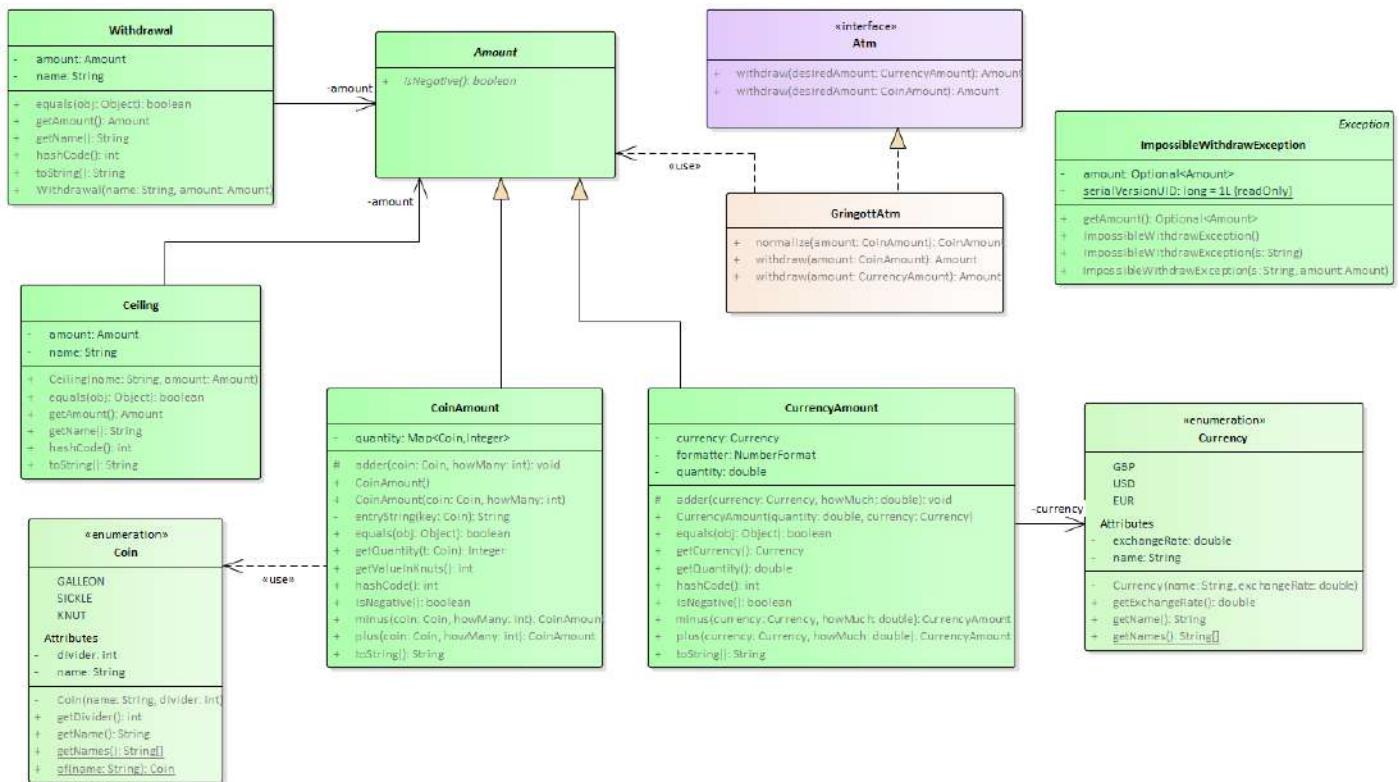
```
--module-path "C:\applicativi\moduli\javafx-sdk-17.0.2\lib"  
--add-modules javafx.controls
```

Parte 1 – Modello dei dati

(punti: 13)

Package: gringott.model

[TEMPO STIMATO: 45-60 minuti]



SEMANTICA:

- L'enumerativo **Coin** (fornito) elenca le monete Gringott: ognuna incorpora sia il proprio nome, sia il divisore (ossia di quanti sottomultipli è costituita). Entrambi queste proprietà sono recuperabili con gli appositi accessori. In aggiunta, per comodità, il metodo statico *getNames* restituisce l'elenco dei nomi delle monete sotto forma di array di stringhe, mentre il metodo-factory statico *of* restituisce l'opportuna costante enumerativa a partire dal suo nome (case-insensitive).
- L'enumerativo **Currency** (fornito) elenca le valute umane riconosciute: ognuna incorpora sia il proprio nome, sia il tasso di cambio contro 1 galeone (quindi la sterlina, GBP, incorpora il fattore 5.0 perché 1 galeone vale 5 sterline, mentre l'euro, EUR, incorpora 5.66 perché 1 galeone vale appunto 5.66 euro; e così via). Entrambe queste proprietà sono recuperabili con gli appositi accessori. In aggiunta, per comodità, il metodo statico *getNames* restituisce l'elenco dei nomi delle valute sotto forma di array di stringhe.
- La classe astratta **Amount** (fornita) rappresenta il concetto generale di un certo “ammontare” di denaro, indipendentemente dal modo con cui viene espresso. Dichiara solo il metodo astratto *isNegative*, vero se l'importo rappresentato è negativo.
- Le due classi concrete **CoinAmount** e **CurrencyAmount** (fornite) specializzano **Amount** nei due casi in cui l'importo sia espresso rispettivamente in monete Gringott o in valute umane.
 - Nel caso di **CoinAmount**, il costruttore crea un ammontare zero, a cui possono poi essere aggiunte o tolte unità di singole monete tramite i due metodi *plus* e *minus*: essi rispettano il pattern cascading e possono quindi essere invocati in sequenza per costruire importi complessi (v. esempi nel codice). Il metodo *getQuantity* restituisce la quantità di una certa moneta (galeoni, falci o zellini) presente nell'**Amount**, mentre il metodo *getValueInKnuts* restituisce l'equivalente in zellini (il sottomultiplo più piccolo, dunque un valore sempre intero) dell'ammontare stesso. Un'apposita *toString* restituisce una stringa opportuna che descrive compiutamente l'importo.

- Nel caso di **CurrencyAmount**, invece, il costruttore crea direttamente l'ammontare richiesto nella valuta (**Currency**) desiderata: i due metodi *plus* e *minus*, che operano come sopra, in questo caso *non* sono indispensabili e sono inclusi a soli fini di test ed espansioni future.
Il metodo *getQuantity* restituisce la quantità di denaro presente nell'**Amount**, mentre il metodo *getCurrency* restituisce la valuta in cui tale ammontare è espresso. Anche qui un'apposita *toString* restituisce una stringa opportuna che descrive compiutamente l'importo.
- La classe **Ceiling** (fornita) rappresenta il massimale per singola operazione concesso a un dato utente: come tale, incorpora il nome utente (una stringa) e il corrispondente **Amount**.
 - L'interfaccia **Atm** (fornita) dichiara i due metodi *withdraw* che effettuano un prelievo espresso rispettivamente in **CoinAmount** e **CurrencyAmount**.
 - La classe **GringottAtm (da completare)** implementa **Atm** applicando le regole 3 e 4 del Dominio del Problema, erogando per quanto possibile il massimo numero di galeoni d'oro ma garantendo altresì che siano sempre presenti almeno 5 falci d'argento e applicando, nel caso di valute umane, gli opportuni tassi di cambio.
Se l'ammontare richiesto è negativo dev'essere lanciata apposita **ImpossibleWithdrawException** (fornita).
 - La classe **Withdrawal** (fornita) è un puro contenitore di dati che esprime il prelievo di un certo **Amount** da parte di un dato utente. Entrambe le proprietà sono recuperabili tramite appositi accessori.

Parte 2 – Persistenza

Package: gringott.persistence

(punti: 11)

[TEMPO STIMATO: 60-80 minuti]

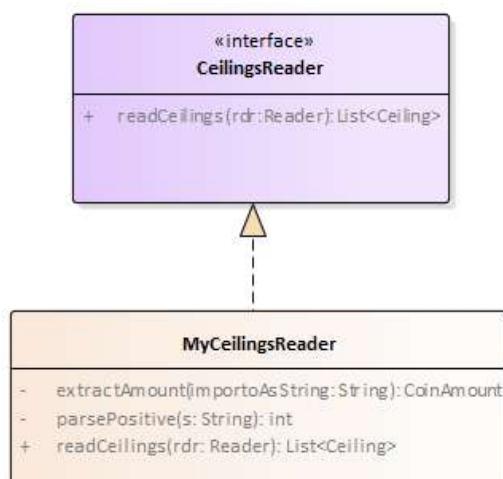
Il file di testo **Ceilings.txt** contiene i massimali permessi a ogni utente per ogni operazione. Ogni riga descrive un singolo utente: non sono previste omonimie, quindi si può assumere senz'altro che il nome dell'utente sia univoco.

Ogni riga contiene, separati da tabulazioni, il nome dell'utente e il massimale ad esso associato, espresso sempre e comunque in monete Gringott.

A sua volta, tale massimale è composto da un elenco di 1-3 parti (galeoni, falci, zellini), separate da virgolette, che possono anche non essere tutte presenti: ogni parte a sua volta è composta da un numero intero seguito, dopo uno o più spazi, dal nome della moneta.

ESEMPIO

Harry	20 galeoni
Hermione	19 galeoni, 20 zellini
Enrico	200 galeoni
Ambra	100 galeoni, 20 zellini
Roberta	100 galeoni, 28 zellini
Ron	12 galeoni, 10 falci, 6 zellini



SEMANTICA:

- a) L'interfaccia **CeilingsReader** (fornita) dichiara il metodo `readCeilings` che carica da un apposito Reader (già aperto) i dati necessari, restituendo una lista di **Ceiling**. Il metodo lancia:
- l'opportuna **BadFormatException** con messaggio d'errore appropriato in caso di problemi nel formato del file (mancanza di elementi, mancanza di parti numeriche quando previste, numeri negativi, etc.)
 - una **IOException** in caso di altri problemi di I/O.
- b) La classe **MyCeilingsReader** (da realizzare) implementa **CeilingsReader** secondo le specifiche sopra descritte.

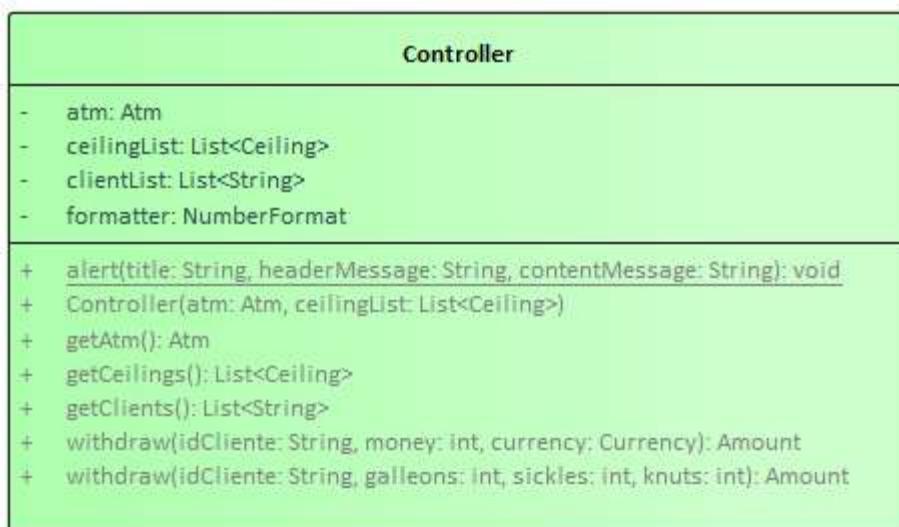
Parte 3

(punti: 6)

Package: gringott.controller

(punti 0)

Il Controller è organizzato secondo il diagramma UML seguente:



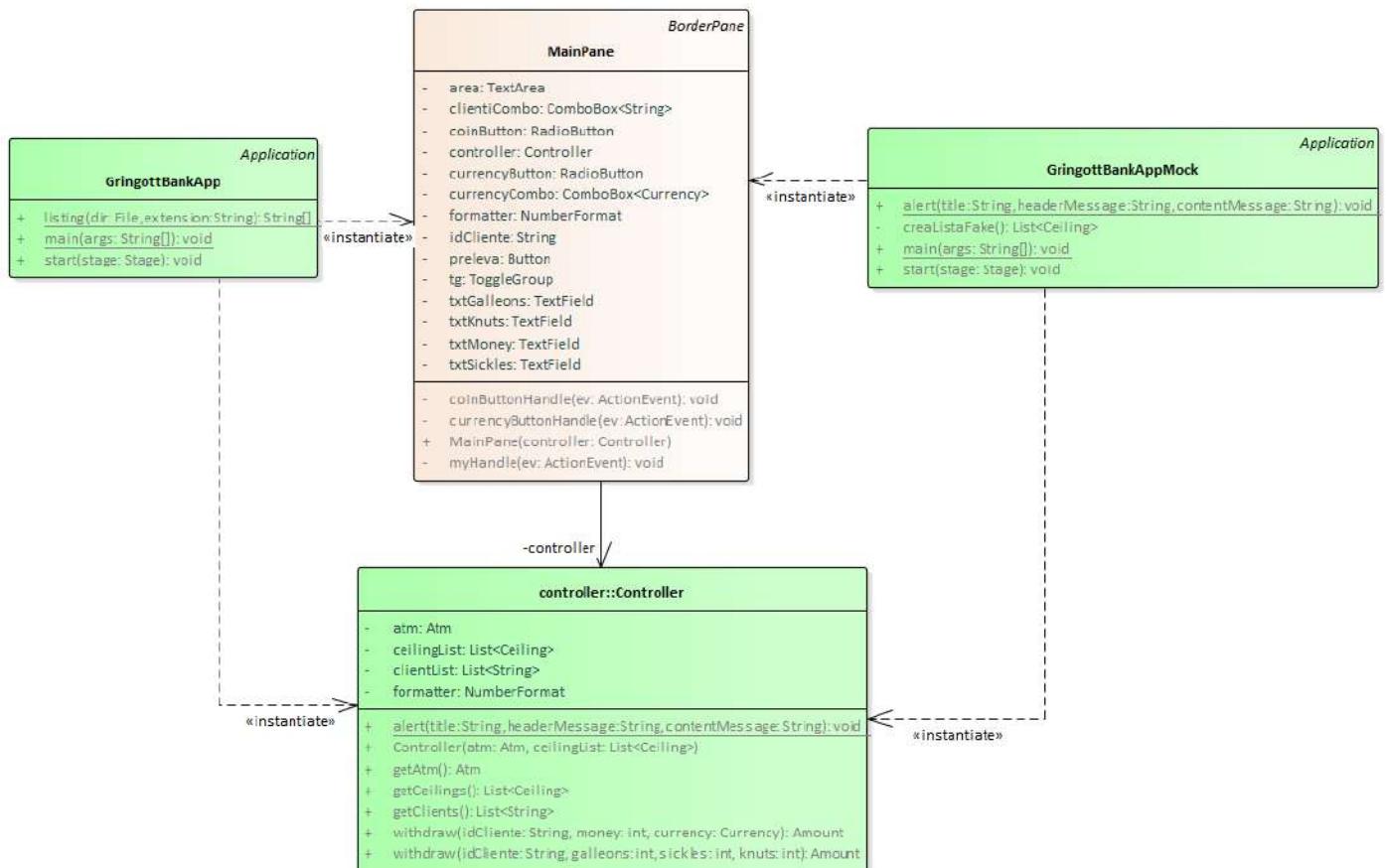
SEMANTICA:

La classe **Controller** (fornita) incorpora in fase di costruzione di un **Atm** e una lista di massimali (**Ceiling**), effettuando quindi le verifiche previste dalle regole 1 & 2 del Dominio del Problema. Oltre ai due ovvi accessori che recuperano gli argomenti passati al costruttore, rende disponibili i seguenti metodi:

- `getClients`, che restituisce la lista dei clienti (stringhe) costruita a partire da quella dei massimali;
- `withdraw` (in due varianti, con diverse liste di argomenti) che consentono alla user interface di richiedere un prelievo per un dato utente e un certo ammontare, espresso o come monete Gringott (quindi tramite terna di interi per galeoni, falci e zellini) o come valuta umana (quindi tramite importo e **Currency**). In entrambi i casi il risultato è un **Optional<Amount>**, vuoto nel caso il prelievo risulti impossibile, o istanziato all'opportuno **CoinAmount** o **CurrencyAmount** se il prelievo è possibile. Lancia **ImpossibleWithdrawException** in caso di prelievo non autorizzato: nel caso il motivo sia il superamento del massimale, l'eccezione incorpora anche un campo dati di tipo **Amount** che esprime l'importo che sarebbe stato da erogare.
- il metodo statico `alert`, utilizzabile anche dal MainPane, consente di far comparire all'utente una finestra di dialogo con opportuno messaggio d'errore.

La classe **GringottBankApp** (fornita) costituisce l'applicazione JavaFX che si occupa di aprire i file, creare il controller e incorporare il **MainPane**. Per consentire di collaudare la GUI anche in assenza / in caso di malfunzionamento della parte di persistenza, è possibile avviare l'applicazione mediante la classe **GringottBankAppMock**.

L'interfaccia utente è illustrata nelle figure seguenti e segue il modello sotto illustrato:



L'interfaccia grafica si presenta come segue:

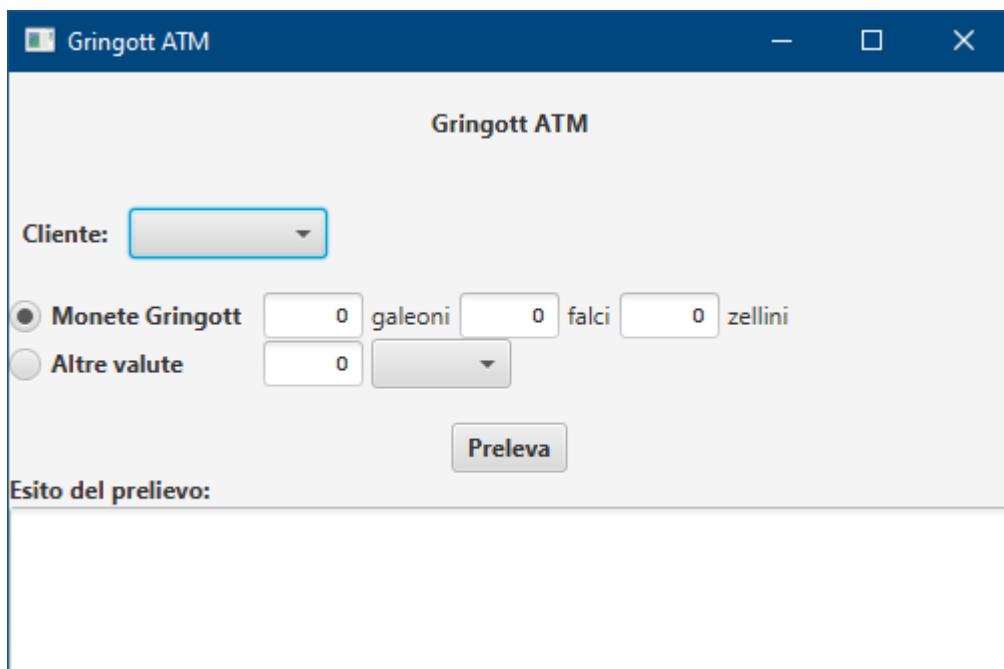


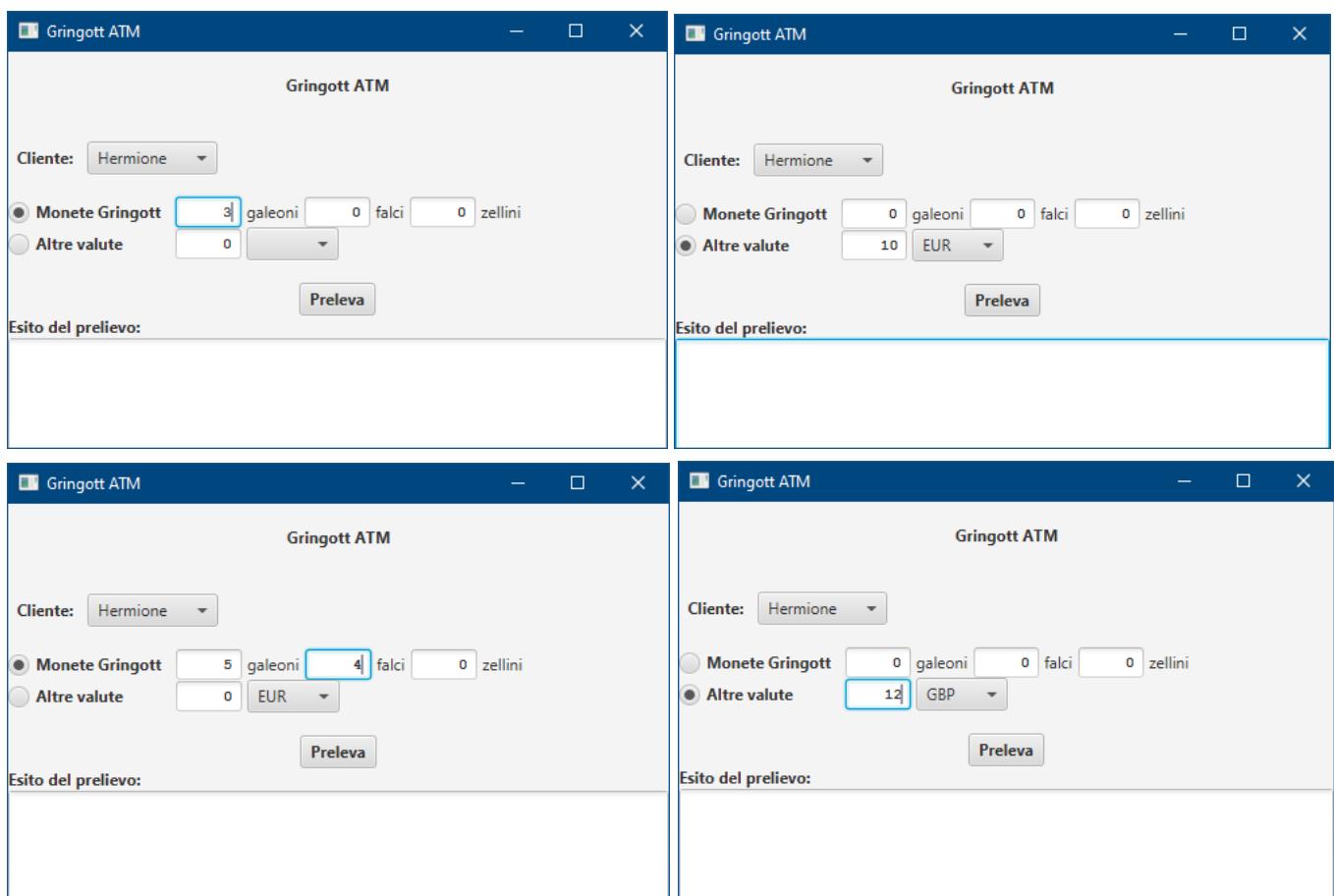
Fig. 1: la situazione iniziale della GUI, con combo vuota e nessuna selezione ancora effettuata

- in alto, una combo elenca i possibili clienti; subito sotto, due bottoni radio consentono di scegliere se specificare l'importo in monete Gringott o valute umane. A tal fine, di fianco a ciascuno di essi sono presenti alcuni campi di testo (con, occorrendo, apposite etichette descrittive) o, nel caso delle valute umane, la combo per scegliere quale valuta usare fra quelle supportate. Sotto a tutto ciò, il pulsante PRELEVA attiva la richiesta di prelievo.

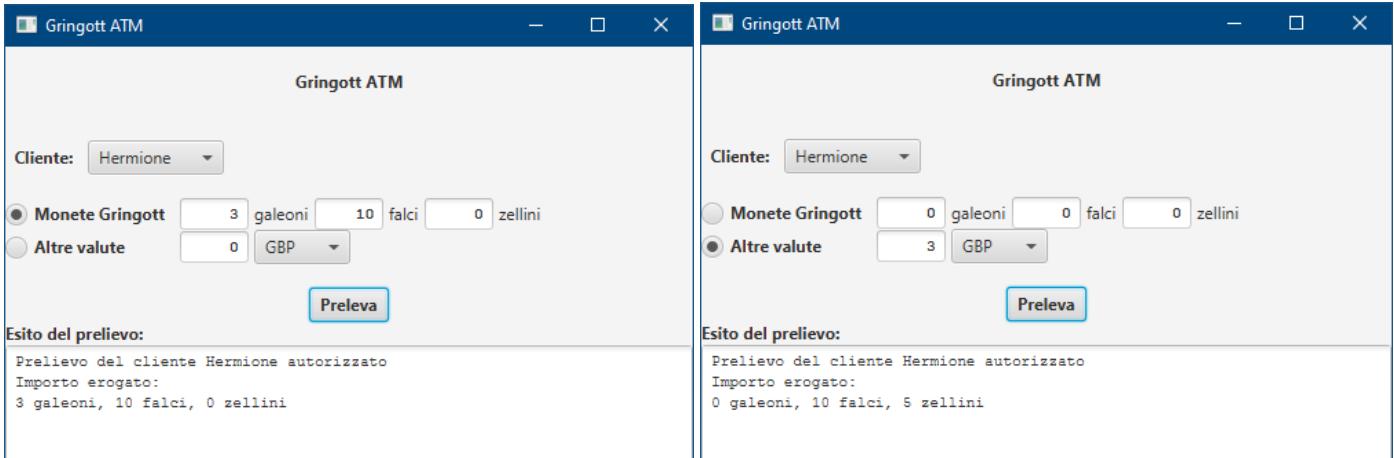
- In basso, un'area di testo (inizialmente vuota e non scrivibile dall'utente) mostra i dettagli del prelievo.

Comportamento:

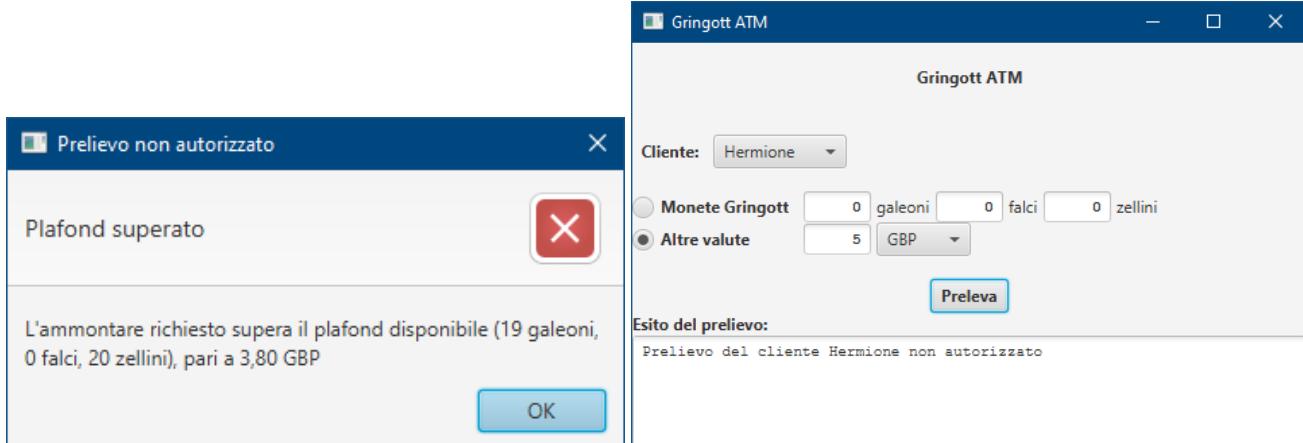
- la selezione di un utente dalla combo non produce effetti: tale valore è infatti consultato solo alla pressione del pulsante PRELEVA.
- La selezione di un bottone radio causa, invece, l'abilitazione in scrittura dei campi di testo a lato e la corrispondente disabilitazione e azzeramento di quelli non utilizzati (Figg. 2,3,4,5). Inizialmente dev'essere selezionato il caso delle monete Gringott.
- Premendo il pulsante PRELEVA, viene recuperato il nome del cliente (se non selezionato, dev'essere emesso apposito messaggio di errore tramite finestra di dialogo) e attivata, tramite Controller, la richiesta di prelievo: l'esito viene mostrato nell'area di testo sottostante (Figg. 6,7)



Figg. 2 / 3 / 4 / 5: la GUI selezionando monete Gringott (prima) o altre valute (dopo): si noti come in questo caso vengano azzerati i campi di testo delle monete Gringott. Dualmente, se dalle valute umane si tornano a selezionare monete Gringott, si azzerano a campi di testo delle prime; e così via.



Figg. 6 / 7 : prelievi autorizzati, sia in monete Gringott (prima) sia in valute umane (dopo).



Figg. 8 / 9: se il prelievo non è autorizzato, viene emesso su finestra di dialogo un apposito messaggio d'errore che ne dettaglia la causa (a sinistra), mentre l'area di testo riporta semplicemente l'esito negativo dell'operazione.

Il MainPane è fornito parzialmente realizzato: è presente quasi tutta l'impostazione strutturale, mentre sono da completare la configurazione di alcuni componenti e la gestione degli eventi.

In particolare, **MainPane** estende **BorderPane** e prevede:

- 1) sopra, una **VBox** altre box interne per combo, campi di testo, bottoni vari ed etichette ausiliarie
- 2) sotto, una **VBox** con la sola area di output.

La **parte da completare** riguarda:

- 1) la configurazione del gruppo di bottoni radio e l'aggancio dei relativi ascoltatori, già implementati nei due metodi ausiliari `coinButtonHandle` e `currencyButtonHandle`
- 2) la logica di gestione dell'evento, incapsulata nel metodo privato `myHandle`

In particolare, la gestione dell'evento principale, tramite `myHandle`, deve:

- verificare che sia selezionato un cliente nell'apposita combo e, se sì, recuperarlo, o in alternativa emettere apposito messaggio d'errore
- estrarre dai campi di testo i dati per il prelievo e, dopo averne verificato la correttezza, attivare il prelievo tramite i metodi `withdraw` del **Controller** intercettando, se necessario, `ImpossibleWithdrawException` per emettere (via `Controller.alert`) il corrispondente messaggio d'errore
- emettere nell'area di testo l'esito del prelievo, specificando innanzitutto se esso sia stato autorizzato e, in questo caso, l'importo effettivamente emesso.

Cose da ricordare

- salva costantemente il tuo lavoro: l'informatica a volte può essere "subdolamente ostile"..\
- in particolare: se ora compila e stai per fare modifiche, salva la versione attuale (non si sa mai)

Checklist di consegna

- Hai fatto un **JAR eseguibile**, che contenga cioè l'indicazione del main?
- Hai controllato che **si compili e ci sia tutto?** [NB: non includere il PDF del testo]
- Hai **rinominato IL PROGETTO**, lo ZIP e il JAR esattamente come richiesto?
- Hai **chiamato** la cartella del progetto esattamente come richiesto?
- **Hai fatto un unico file ZIP (NON .7z, rar o altri formati) contenente l'intero progetto?**
In particolare, ti sei assicurato di aver incluso tutti i file .java (e non solo i .class)?
- **Hai consegnato DUE file distinti, ossia lo ZIP col progetto e il JAR eseguibile?**
- Su EOL, hai premuto il tasto "CONFERMA" per inviare il tuo elaborato?

ESAME DI FONDAMENTI DI INFORMATICA T-2 del 13/1/2023

Proff. E. Denti – R. Calegari – A. Molesini

Tempo a disposizione: 3h30

NOME PROGETTO ECLIPSE: CognomeNome-matricola (es. RossiMario-0000123456)

NOME CARTELLA PROGETTO: CognomeNome-matricola (es. RossiMario-0000123456)

NOME ZIP DA CONSEGNARE: CognomeNome-matricola.zip (es. RossiMario-0000123456.zip)

NOME JAR DA CONSEGNARE: CognomeNome-matricola.jar (es. RossiMario-0000123456.jar)

Si devono consegnare DUE FILE: *l'intero progetto Eclipse e il JAR esegibile*

Si ricorda che compiti non compilabili o palesemente lontani da 18/30 NON SARANNO CORRETTI e causeranno la verbalizzazione del giudizio “RESPINTO”

Per incentivare gli studenti a procedere con celerità, nell'università SpeedCollege i voti ottenuti nei vari esami subiscono un *processo di decadimento* al passare del tempo: la legge con cui tale decadimento avviene è personalizzabile dal singolo corso di studio. A tutela dello studente è tuttavia stabilito che nessun voto positivo possa scendere sotto quota 18/30, che costituisce quindi la soglia minima, raggiunta la quale il decadimento si arresta e il voto rimane costante.

Si vuole sviluppare un'applicazione che consenta di seguire la carriera degli studenti al passare del tempo, *calcolando e mostrando i voti e la media pesata validi in un dato giorno*, oltre ai crediti globalmente acquisiti.

DESCRIZIONE DEL DOMINIO DEL PROBLEMA

Un'**attività formativa** rappresenta un insegnamento universitario caratterizzato da *numero di codice* (univoco), *denominazione* (che può contenere spazi) e *numero di crediti* (non necessariamente intero).

Quando, **in una certa data**, lo studente sostiene un **esame**, all'attività formativa viene associato un **voto iniziale**, che può essere o un *numero fra 18 e 30* (quest'ultimo eventualmente con lode) o un giudizio di *idoneità*; in caso di esito negativo, viene attribuito uno dei giudizi *ritirato* o *respinto*. Ogni esame superato comporta l'acquisizione dei relativi *crediti*. È possibile ri-sostenere un esame solo se in precedenza l'esito è stato negativo, mentre non è consentito ripetere un esame già sostenuto con esito positivo. Al termine della carriera, lo studente deve sostenere la **prova finale**: essa può quindi essere sostenuta solo *in data successiva* ad ogni altro esame.

In qualunque momento la carriera dello studente è quindi caratterizzata da:

- **Lista degli esami sostenuti** (sia con esito positivo che con esito negativo)
- **Crediti acquisiti** (concorrono all'acquisizione dei crediti solo gli esami superati con *esito positivo*)
- **Media pesata** (concorrono all'acquisizione dei crediti solo gli esami con voto, superati con *esito positivo*)

La media pesata al giorno d si calcola secondo la formula $MP(d) = \frac{\sum v_i(d)*p_i}{\sum p_i}$, essendo $v_i(d)$ i voti ricalcolati al giorno d, e p_i i pesi (crediti) degli esami.

La legge con cui i voti vengono ricalcolati (diminuendoli) al passare del tempo è personalizzabile: il voto così calcolato può anche essere non intero (es. 25.5/30), per evitare eccessive distorsioni.

Una serie di file di testo (i cui nomi non sono specificati né rilevanti) descrive possibili carriere di studenti, nel formato descritto più oltre.

TEMPO STIMATO PER SVOLGERE L'INTERO COMPITO:

2h15 – 3h

PARTE 1 – Modello dei dati: Punti 13

[TEMPO STIMATO: 60-70 minuti]

PARTE 2 – Persistenza: Punti 9

[TEMPO STIMATO: 40-60 minuti]

PARTE 3 – Grafica: Punti 8

[TEMPO STIMATO: 35-50 minuti]

JAVAFX – Parametri run configuration nei LAB

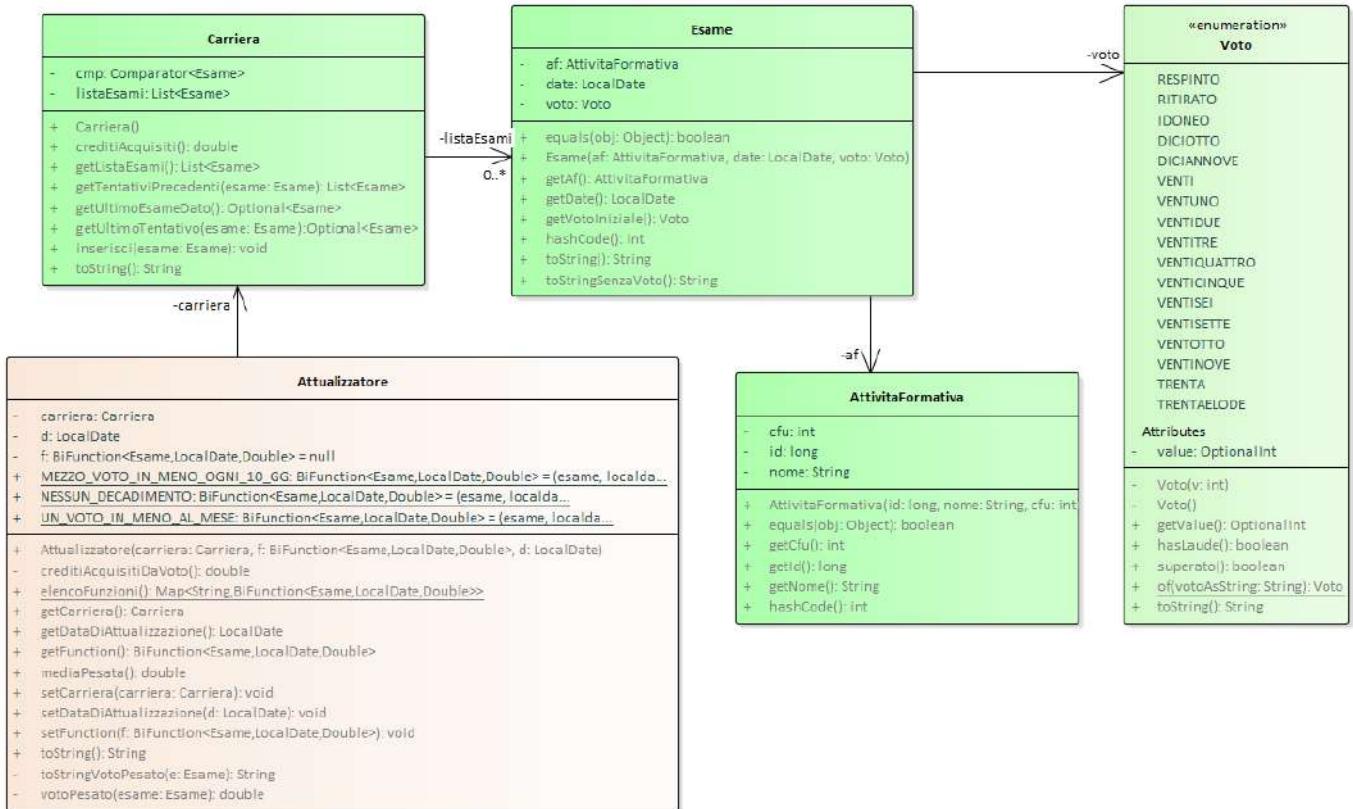
```
--module-path "C:\applicativi\moduli\javafx-sdk-17.0.2\lib"  
--add-modules javafx.controls
```

Parte 1 – Modello dei dati

(punti: 13)

Package: *speedycollege.model*

[TEMPO STIMATO: 60-70 minuti]



SEMANTICA:

- la classe **AttivitaFormativa** (fornito) descrive un'attività formativa come da dominio del problema
- l'enumerativo **Voto** (fornito) rappresenta i possibili valori di voto o giudizio, come da dominio del problema; in particolare, sono definite sia costanti enumerative per ogni voto fra 18 e 30L, associate al corrispondente valore numerico (per 30L il valore è 30), nonché costanti enumerative per i tre giudizi respinto, ritirato e idoneo, non associate ad alcun valore numerico. Il metodo `getValue` estrae il valore intero associato, se esistente, mentre i due metodi `hasLaude` e `superato` sono veri rispettivamente nel solo caso del 30L, e per i voti positivi, ossia diversi da *ritirato* o *respinto*. Il factory method statico `of` restituisce la giusta istanza dell'enumerativo in base alla stringa ricevuta, che può essere o il valore numerico fra "18" e "30", oppure una delle stringhe "30L", "ID", "RT", "RE". Dualmente, un'apposita `toString` emette la stringa appropriata per ogni valore dell'enumerativo.
- La classe **Esame** (fornito) rappresenta un esame, che associa a un'attività formativa un voto ottenuto in una ben precisa data. Il costruttore riceve tali dati, che sono poi recuperabili tramite gli appositi *accessori*; idonee `toString`, `equals`, `hashCode` completano l'implementazione. Da segnalare il metodo di utilità `toStringSenzaVoto`, che emette una stringa analoga a quella di `toString` ma priva del voto finale (utile per generare facilmente, in altra situazione, la carriera attualizzata, in cui il voto dovrà essere diverso)
- la classe **Carriera** (fornita) mantiene al proprio interno la lista degli esami sostenuti, ordinata per data crescente e in subordine per codice univoco dell'attività formativa. Il costruttore crea una carriera inizialmente vuota, che verrà poi riempita via via tramite il metodo `inserisci`. Più precisamente:
 - il metodo `inserisci` inserisce un esame in carriera, a condizione che un esame per la stessa attività formativa non sia già stato sostenuto in precedenza con esito positivo (altrimenti, deve lanciare **IllegalArgumentException** con opportuno messaggio d'errore); deve inoltre, ovviamente, verificare che l'argomento ricevuto non sia nullo, lanciando in tal caso **IllegalArgumentException** con

apposito messaggio d'errore. Nel solo caso in cui l'esame da inserire sia la prova finale (aspetto verificabile unicamente dalla descrizione, che in tal caso conterrà la dizione “PROVA FINALE”), deve altresì *verificare che la data della stessa sia posteriore a quella di ogni altro esame in carriera* – anche in questo caso, lanciando **IllegalArgumentException** con adeguato messaggio

- il metodo `getListaEsami` restituisce semplicemente la lista ordinata degli esami attualmente presenti in carriera
- il metodo `getTentativiPrecedenti` restituisce la lista ordinata dei soli esami (con esito ovviamente negativo) sostenuti in precedenza per la stessa attività formativa dell'esame ricevuto come argomento (se non ce ne sono, restituisce una lista vuota)
- il metodo `getUltimoEsameData` restituisce, se esiste (per questo il tipo di ritorno è un **Optional**), il più recente degli esami sostenuti
- il metodo `getUltimoTentativo` restituisce, se esiste (per questo il tipo di ritorno è un **Optional**), il più recente degli esami (con esito negativo) sostenuti in precedenza per la stessa attività formativa dell'esame ricevuto come argomento
- il metodo `creditiAcquisiti` restituisce la somma dei crediti acquisiti da esami superati con esito positivo (sia con voto numerico, sia con giudizio di idoneità)
- un'apposita `toString` emette una stringa corrispondente alla lista di tutti gli esami (superati o meno), separati da “a capo”, rispettando l'ordine di lista.

- e) la classe **Attualizzatore (da completare)** attualizza una **Carriera**: il costruttore riceve tre argomenti, che costituiscono lo stato iniziale dell'attualizzatore (la carriera da attualizzare, la legge di decadimento desiderata e la data a cui effettuare l'attualizzazione) e le memorizza nello stato interno. Tale stato è però accessibile e modificabile in qualsiasi momento tramite le coppie di accessori `setCarriera / getCarriera`, `setFunction / getFunction`, `setDataDiAttualizzazione / getDataDiAttualizzazione`. Inoltre:

- il metodo **mediaPesata (da fare)** restituisce la media pesata dei voti (numerici) fin qui ottenuti, attualizzata alla data specificata per mezzo della funzione di decadimento specificata (NB: possono essere utili alcuni metodi privati forniti...)
- un'apposita `toString` emette una stringa corrispondente alla lista di tutti gli esami (superati o meno), separati da “a capo”, rispettando l'ordine di lista

La classe definisce altresì, sotto forma di metodi statici, alcune funzioni di decadimento prestabilite, recuperabili tramite l'apposito metodo `elencoFunzioni`, che restituisce una mappa che associa un nome convenzionale (stringa) a ogni funzione (una `BiFunction<Esame, LocalDate, Double>`). **Al momento ne sono previste tre, di cui due già fornite e una da realizzare.**

- la funzione statica **UN_VOTO_IN_MENO_AL_MESE (da fare)** attualizza l'esame dato alla data fornita come argomento, restituendo il risultato sotto forma di double, secondo la logica di sottrarre un voto per ogni mese di distanza dalla data dell'esame (ad esempio, un esame sostenuto il 22/3/2020 con voto 23 vedrà il voto calare a 22 dal 22/4/2020; e così via), garantendo comunque che nessun voto possa mai scendere sotto quota 18 e che i voti non numerici restino inalterati (NB: può essere utile ispirarsi alle altre due fornite... 😊)

Parte 2 – Persistenza

Package: `speedycollege.persistence`

(punti: 9)

[TEMPO STIMATO: 40-60 minuti]

Sono forniti un certo numero di file di testo, tutti strutturati secondo il medesimo formato (possono contenere righe vuote). L'applicazione principale provvede da sé a recuperarne l'elenco dalla directory corrente e richiamarne ciclicamente la lettura: pertanto il candidato deve occuparsi unicamente di implementare il classico reader per un singolo file.

Ogni riga è organizzata in una serie di elementi separati da una o più tabulazioni. Due elementi ci sono sempre:

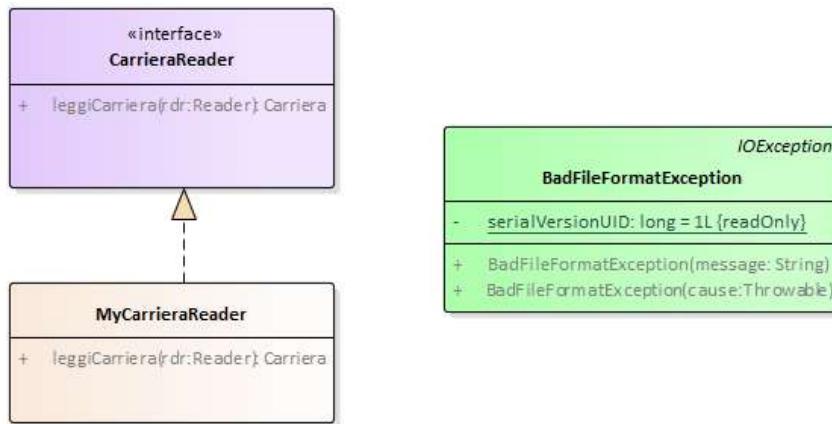
- codice identificativo dell'attività formativa (un intero long)
- denominazione dell'attività formativa (che può contenere spazi e ogni altro carattere diverso da tabulazione), seguita – dopo almeno uno spazio – dal numero di crediti nella forma “(cfu:N)”, essendo N un numero intero positivo (NB: la sigla “cfu:” è sempre minuscola)

Solo se l'esame è stato sostenuto, sono presenti due ulteriori elementi:

- data in cui l'esame è stato sostenuto, nel formato italiano short GG/MM/AA
- voto (un numero fra 18 e 30, oppure una delle sigle 30L, ID, RT, RE rispettivamente per trenta e lode, idoneo, ritirato, respinto)

ESEMPIO:

27991	ANALISI MATEMATICA T-1 (cfu:9)	12/01/20	RT
27991	ANALISI MATEMATICA T-1 (cfu:9)	10/02/20	22
28004	FONDAMENTI DI INFORMATICA T-1 (cfu:12)	13/02/20	24
29228	GEOMETRIA E ALGEBRA T (cfu:6)	18/01/20	26
26337	LINGUA INGLESE B-2 (cfu:6)	18/06/20	ID
27993	ANALISI MATEMATICA T-2 (cfu:6)	10/06/20	RE
27993	ANALISI MATEMATICA T-2 (cfu:6)	02/07/20	RT
27993	ANALISI MATEMATICA T-2 (cfu:6)	22/07/20	23
28006	FONDAMENTI DI INFORMATICA T-2 (cfu:12)	21/07/20	27
28011	RETI LOGICHE T (cfu:6)		
28012	CALCOLATORI ELETTRONICI T (cfu:6)	22/01/21	RT
28012	CALCOLATORI ELETTRONICI T (cfu:6)	22/02/21	20
30780	FISICA GENERALE T (cfu:9)	12/02/21	25
28032	MATEMATICA APPLICATA T (cfu:6)	02/02/21	24
28027	SISTEMI INFORMATIVI T (cfu:9)	03/06/21	28
28030	ECONOMIA E ORG. AZIENDALE T (cfu:6)	02/07/21	RE
28030	ECONOMIA E ORG. AZIENDALE T (cfu:6)	22/07/21	24
28029	ELETROTECNICA T (cfu:6)	02/09/21	26
28014	FOND. DI TELECOMUNICAZIONI T (cfu:9)	15/09/21	30
28020	SISTEMI OPERATIVI T (cfu:9)	12/01/22	24
28015	CONTROLLI AUTOMATICI T (cfu:9)	13/01/21	25
28016	ELETTRONICA T (cfu:6)	10/02/21	22
28024	RETI DI CALCOLATORI T (cfu:9)	05/02/21	23
28659	TECNOLOGIE WEB T (cfu:9)	12/06/21	25
28021	INGEGNERIA DEL SOFTWARE T (cfu:9)	24/06/21	27
17268	PROVA FINALE (cfu:3)		
28074	TIROCINIO T (cfu:6)	27/09/21	ID
88324	AMMINISTRAZIONE DI SISTEMI T (cfu:6)	13/07/21	29
88325	LAB. SICUREZZA INFORMATICA T (cfu:6)	07/06/21	30L



SEMANTICA:

- a) L'eccezione **BadFileFormatException** (fornita) esprime l'idea di file formattato in modo scorretto
- b) L'interfaccia **CarrieraReader** (fornita) dichiara il metodo **leggiCarriera**, che legge da un **Reader** (ricevuto come argomento) i dati di una carriera, configurando e restituendo l'opportuno oggetto **Carriera**.

IMPORTANTE: poiché la **Carriera** è composta di **Esami**, le righe contenenti la sola descrizione di attività formative, ossia quelle senza data e voto, devono essere comunque verificate a livello di formato ma **ignorate per quanto riguarda l'inserimento in carriera**, in quanto non descrivono un esame sostenuto.

- c) La classe **MyCarrieraReader** (**da realizzare**) implementa **CarrieraReader**: non prevede costruttori, si limita a implementare il metodo **leggiCarriera** come sopra specificato. In caso di problemi di I/O deve essere propagata l'opportuna **IOException**, mentre in caso di **Reader** nullo o altri problemi di formato dei file deve essere lanciata una opportuna **BadFormatException**, il cui messaggio dettagli l'accaduto.
In particolare, il reader deve verificare, lanciando **BadFormatException** in caso contrario, che:
- i. la riga contenga due o quattro elementi
 - ii. il primo elemento sia un intero long
 - iii. i crediti siano presenti nel formato sopra specificato, ovvero "(cfu:N)" nonché, dove siano presenti anche gli altri due elementi:
 - iv. la data sia correttamente formattata secondo la struttura GG/MM/AA
 - v. il voto sia un numero intero compreso fra 18 e 30, oppure una delle sigle sopra specificate

Parte 3

(punti: 7)

Package: *speedycollege.controller*

(punti 0)

Il **Controller** (fornito) è organizzato secondo il diagramma UML nella figura seguente: esso mantiene internamente una mappa <stringa, carriera> che associa a ogni **Carriera** una opportuna stringa identificativa univoca.



SEMANTICA:

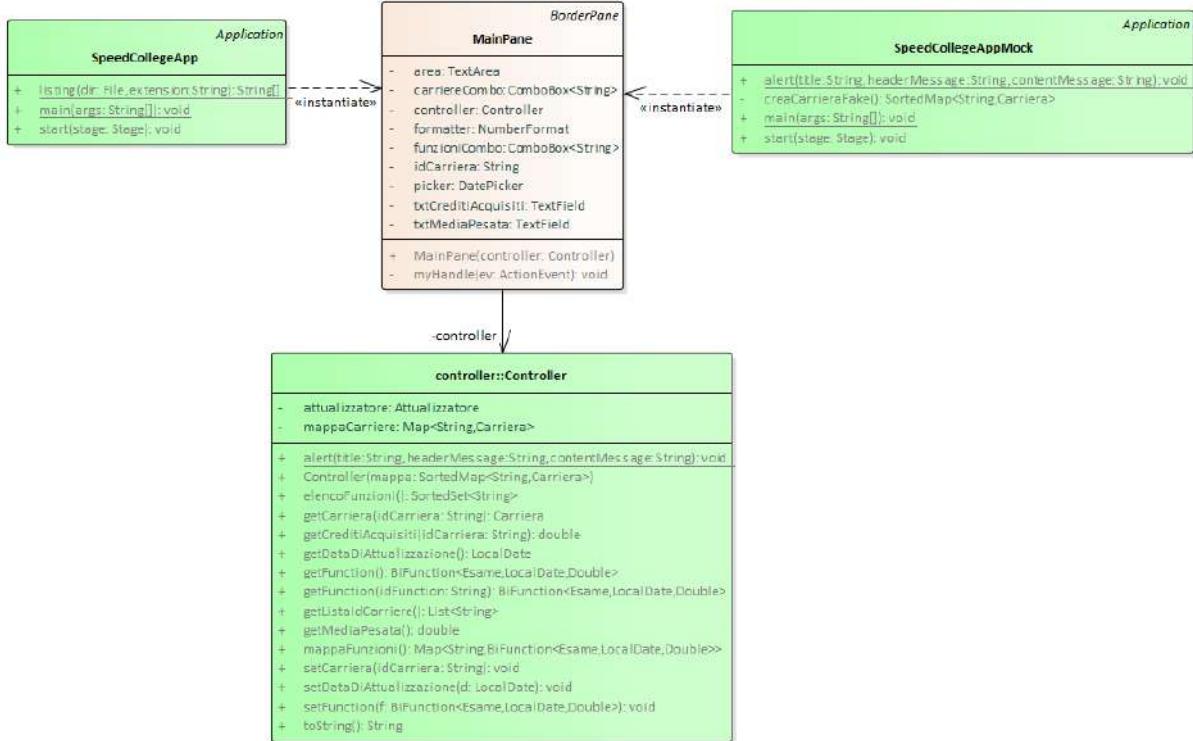
- il costruttore riceve la mappa <stringa,carriera> su cui opererà
- un'ampia serie di metodi fa da ponte con i quasi omonimi metodi dell'**Attualizzatore**
- la classe contiene anche il **metodo statico ausiliario alert**, utile per mostrare avvisi all'utente.

Package: *speedycollege.ui*

[TEMPO STIMATO: 35-50 minuti] (punti 8)

La classe **SpeedyCollegeApp** (fornita) costituisce l'applicazione JavaFX che si occupa di aprire i file, creare il controller e incorporare il **MainPane**. Per consentire di collaudare la GUI anche in assenza / in caso di malfunzionamento della parte di persistenza, è possibile avviare l'applicazione mediante la classe **SpeedyCollegeAppMock**.

L'interfaccia utente è illustrata nelle figure seguenti e segue il modello sotto illustrato:



L'interfaccia grafica si presenta come segue:

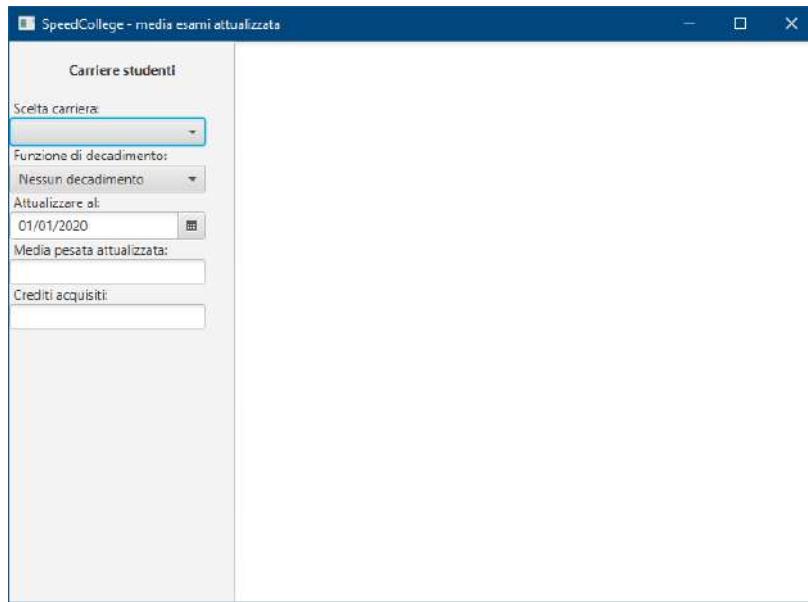


Fig. 1: la situazione iniziale della GUI, con combo vuota e nessuna selezione ancora effettuata

- a sinistra, due combo elencano una le carriere disponibili, l'altra le funzioni di decadimento predefinite; sotto, un caledarietto (**DatePicker**) consente di stabilire la data a cui attualizzare i voti (default: 1° gennaio 2020). Subito a seguire, due campi di testo (inizialmente vuoti e non scrivibili dall'utente) mostrano rispettivamente le media pesata attualizzata (che quindi può cambiare al variare della data scelta) e i crediti acquisiti relativi alla carriera selezionata (che invece restano, ovviamente, fissi al variare della data).
- a destra, un'area di testo (inizialmente vuota e non scrivibile dall'utente) mostra i dettagli della carriera selezionata, ossia la lista degli esami in essa contenuti.

L'utente può interagire selezionando o la carriera nella combo, o la funzione di decadimento, o la data di attualizzazione: in tutti i casi viene scatenato lo stesso gestore di eventi, che provvede a popolare i campi di testo sottostanti e l'area sulla destra con i dati corrispondenti.

Figg. 2 / 3: la GUI dopo la selezione di una carriera, prima senza decadimento (a sinistra), poi con una legge di decadimento (a destra), in quel caso riferia alla data del 19/2/2020.

Il MainPane è fornito *parzialmente realizzato*: è presente buona parte dell'impostazione strutturale, mentre sono da completare la configurazione di alcuni componenti e la gestione degli eventi.

La classe **MainPane (da completare)** estende **BorderPane** e prevede:

- 1) a sinistra, una **VBox** con le due combo, il picker e i due campi di testo, oltre alle opportune etichette
- 2) a destra, una **VBox** con la sola area di output.

La **parte da completare** riguarda:

- 1) la configurazione iniziale dei formattatori numerici
- 2) il popolamento delle due **combo**
- 3) la predisposizione, con opportuna data di default al 1° gennaio 2020, del **DatePicker**
- 4) l'aggancio dell'opportuno listener encapsulato nel metodo ausiliario *myHandler*
- 5) la logica di gestione dell'evento, encapsulata nel metodo privato *myHandler*

In particolare, la gestione dell'evento deve:

- recuperare la carriera, la funzione di decadimento e la data di attualizzazione selezionate
- utilizzare tali dati per popolare i vari campi di uscita, tramite gli appositi metodi del controller

Cose da ricordare

- salva costantemente il tuo lavoro: l'informatica a volte può essere "subdolamente ostile".
- in particolare: se ora compila e stai per fare modifiche, salva la versione attuale (non si sa mai)

Checklist di consegna

- Hai fatto un **JAR eseguibile**, che contenga cioè l'indicazione del main?
- Hai controllato che **si compili e ci sia tutto**? [NB: non includere il PDF del testo]
- Hai **rinominato IL PROGETTO**, lo ZIP e il JAR esattamente come richiesto?
- Hai **chiamato** la cartella del progetto esattamente come richiesto?
- **Hai fatto un unico file ZIP (NON .7z, rar o altri formati) contenente l'intero progetto?**
In particolare, ti sei assicurato di aver incluso tutti i file .java (e non solo i .class)?
- **Hai consegnato DUE file distinti, ossia lo ZIP col progetto e il JAR eseguibile?**
- Su EOL, hai premuto il tasto "CONFERMA" per inviare il tuo elaborato?

ESAME DI FONDAMENTI DI INFORMATICA T-2 del 14/9/2022

Proff. E. Denti – R. Calegari – A. Molesini

Tempo a disposizione: 3h30

NOME PROGETTO ECLIPSE: CognomeNome-matricola (es. RossiMario-0000123456)

NOME CARTELLA PROGETTO: CognomeNome-matricola (es. RossiMario-0000123456)

NOME ZIP DA CONSEGNARE: CognomeNome-matricola.zip (es. RossiMario-0000123456.zip)

NOME JAR DA CONSEGNARE: CognomeNome-matricola.jar (es. RossiMario-0000123456.jar)

Si devono consegnare DUE FILE: *l'intero progetto Eclipse e il JAR eseguibile*

Si ricorda che compiti non compilabili o palesemente lontani da 18/30 NON SARANNO CORRETTI e causeranno la verbalizzazione del giudizio “RESPINTO”

La società di gestione di un parcheggio sotterraneo di una grande città ha richiesto un'applicazione per consentire agli utenti di pre-calcolare il costo della sosta, secondo lo schema tariffario descritto di seguito.

DESCRIZIONE DEL DOMINIO DEL PROBLEMA

Il sistema tariffario adottato dalla società deve coniugare diversi aspetti:

- consentire anche soste molto brevi (anche di soli 15 minuti), senza tuttavia che esse risultino in importi troppo bassi, che porterebbero a un utilizzo improprio (“mordi e fuggi”) dello spazio di sosta
- disincentivare soste di troppe ore consecutive, evitando però di allontanare clienti con importi troppo elevati
- incentivare comunque anche le soste medio-lunghe, di 12 o più ore, con una tariffazione ad hoc.

A questo fine, è stato elaborato il seguente schema:

- nelle prime 12 ore, la tariffazione procede a intervalli di 15 minuti, del costo di 0,80 € ciascuno (=3,20€/ora); il primo intervallo di 15 minuti, tuttavia, viene eccezionalmente tariffato 2,00 € (anziché 0,80 €)
- è previsto un tetto giornaliero (price cap) di 35,00 €, che si raggiunge entro le prime 12 ore: raggiunta tale cifra, il costo resta costante fino a 24 ore di sosta
- se la sosta dura più di 24 ore, i periodi successivi alle prime 24 ore sono tariffati a blocchi indivisibili di 12 ore, del costo di 16,00 € ciascuno.

ESEMPI

- sosta entro i 15': 2,00 €
- sosta da 16' e fino a 30': 2,80 € (slot iniziale di 15' da 2,00 € + slot di 15' a 0,80 €)
- sosta di 1 ora: 4,40 € (slot iniziale di 15' da 2,00 € + 3 slot di 15' a 0,80 € ciascuno)
- sosta di 1 ora e 1 minuto: 5,20 € (slot iniziale di 15' da 2,00 € + 4 slot di 15' a 0,80 € ciascuno)
- ...
- sosta di 12 ore: 35,00 € (price cap raggiunto)
- sosta di 13,14,...24 ore: sempre 35,00 € (price cap raggiunto)
- sosta di 24 ore e 1 minuto: 51,00 € (35,00 € per le prime 24 ore + slot indivisibile di 12 ore da 16,00€)
- sosta di 25,26,... 36 ore: sempre 51,00 € (35,00 € per le prime 24 ore + slot indivisibile di 12 ore da 16,00€)
- sosta di 36 ore e 1 minuto: 67,00 € (35,00 € per le prime 24 ore + 2 slot indivisibili di 12 ore da 16,00€ ciascuno)

Il file **Tariffa.txt** contiene il sistema tariffario sopra descritto, opportunamente generalizzato in modo da parametrizzare durata degli slot e costi, nel formato descritto più oltre.

L'applicazione dovrà mostrare a video il costo previsto della sosta, nel formato illustrato nelle figure seguenti.

PARTE 1 – Modello dei dati: Punti 10

[TEMPO STIMATO: 40-50 minuti]

PARTE 2 – Persistenza: Punti 8

[TEMPO STIMATO: 30-40 minuti]

PARTE 3 – Grafica: Punti 12

[TEMPO STIMATO: 50-60 minuti]

TEMPO STIMATO PER SVOLGERE L'INTERO COMPITO:

2h – 2h30

ESEMPI DI CALCOLO DEL COSTO DELLA SOSTA MOSTRATO A VIDEO

The image contains two side-by-side screenshots of a Java application window titled "City Parking".

Screenshot 1 (Left): Shows the input fields for a parking session from "Inizio sosta:" (24/08/2022, 18:56) to "Fine sosta:" (24/08/2022, 18:56). Below the fields is a "Calcola costo" button, and the result area shows "Costo sosta:" (empty).

Screenshot 2 (Right): Shows the input fields for a parking session from "Inizio sosta:" (01/09/2022, 08:30) to "Fine sosta:" (01/09/2022, 08:56). Below the fields is a "Calcola costo" button, and the result area displays the calculation details:

```
City Parking
dalle 08:30 di giovedì 1 settembre 2022
alle 08:56 di giovedì 1 settembre 2022
Durata totale: 0:26
Costo totale: 2,80 €
```

Cose da ricordare

- salva costantemente il tuo lavoro: l'informatica a volte può essere "subdolamente ostile" ..
- in particolare: se ora compila e stai per fare modifiche, salva la versione attuale (non si sa mai)

Checklist di consegna

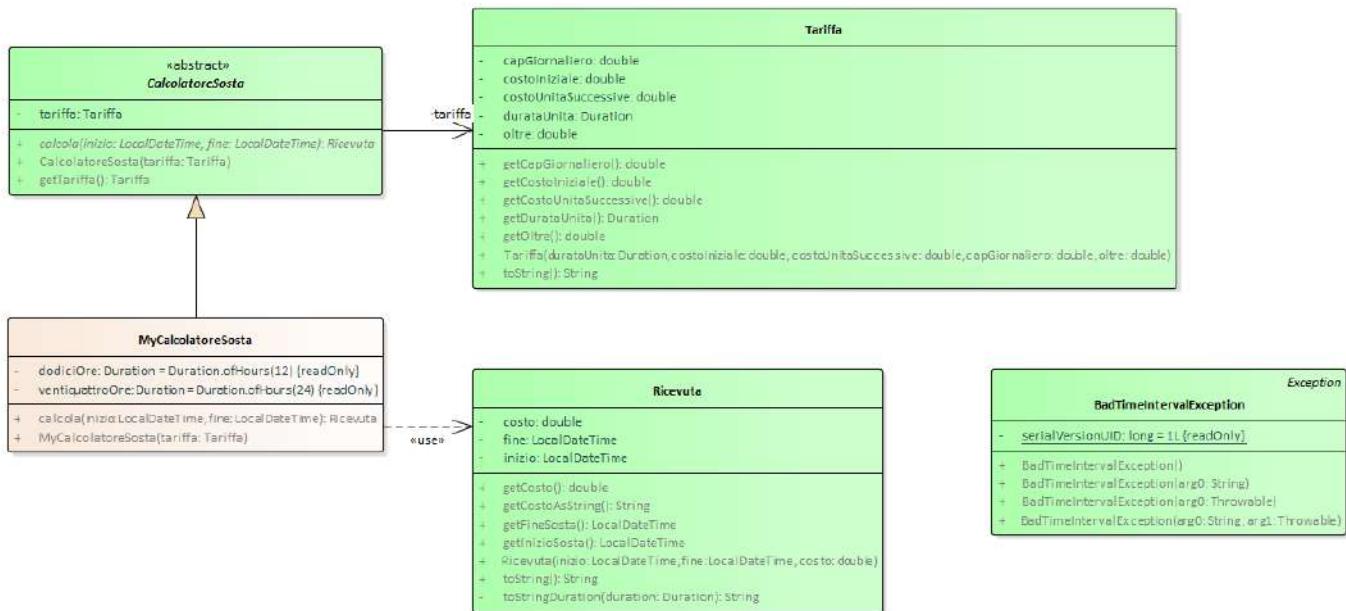
- Hai fatto un **JAR eseguibile**, che contenga cioè l'indicazione del main?
- Hai controllato che **si compil e ci sia tutto?** [NB: non includere il PDF del testo]
- Hai **rinominato IL PROGETTO**, lo ZIP e il JAR esattamente come richiesto?
- Hai **chiamato** la cartella del progetto esattamente come richiesto?
- **Hai fatto un unico file ZIP (NON .7z, rar o altri formati)** contenente l'intero progetto?
In particolare, ti sei assicurato di aver incluso tutti i file .java (e non solo i .class)?
- **Hai consegnato DUE file distinti, ossia lo ZIP col progetto e il JAR eseguibile?**
- Su EOL, hai premuto il tasto "CONFERMA" per inviare il tuo elaborato?

Parte 1 – Modello dei dati

Punti: 10

Package: *cityparking.model*

[TEMPO STIMATO: 40-50 minuti]



SEMANTICA:

- la classe **Tariffa** (fornita) rappresenta il sistema tariffario come descritto nel dominio del problema: i vari *accessor* consentono di recuperare le proprietà
- la classe **Ricevuta** (fornita) rappresenta la ricevuta di parcheggio: contiene il costo e i vari elementi che hanno contribuito a determinarlo, recuperabili tramite appositi *accessor*
- la classe astratta **CalcolatoreSosta** (fornita) rappresenta il componente che effettua il calcolo del costo della sosta sulla base di una data tariffa. Il metodo `calcola`, astratto, è destinato a catturare lo specifico algoritmo di calcolo: data e ora di inizio/fine sosta sono forniti tramite due oggetti *LocalDateTime*.
Se l'orario di inizio sosta non precede quello di fine sosta, il metodo deve lanciare l'apposita **BadTimeIntervalException** (fornita), con adeguato messaggio d'errore.
- la classe **MyCalcolatoreSosta (da completare)** estende la precedente implementando `calcola` secondo l'algoritmo descritto nel dominio del problema.

Parte 2 – Persistenza

Punti: 8

Package: *cityparking.persistence*

[TEMPO STIMATO: 30-40 minuti]

Il file di testo **Tariffa.txt** contiene la descrizione del sistema tariffario, opportunamente parametrizzata. Il file contiene sempre e solo 5 righe, che descrivono, esattamente in quest'ordine:

- la durata dell'unità di tariffazione (tipicamente, 15 minuti)
- il costo della prima unità
- il costo delle successive unità
- il price cap giornaliero, valido entro le prime 24 ore
- il costo dei periodi indivisibili da 12h successivi

Il formato è illustrato nell'esempio seguente:

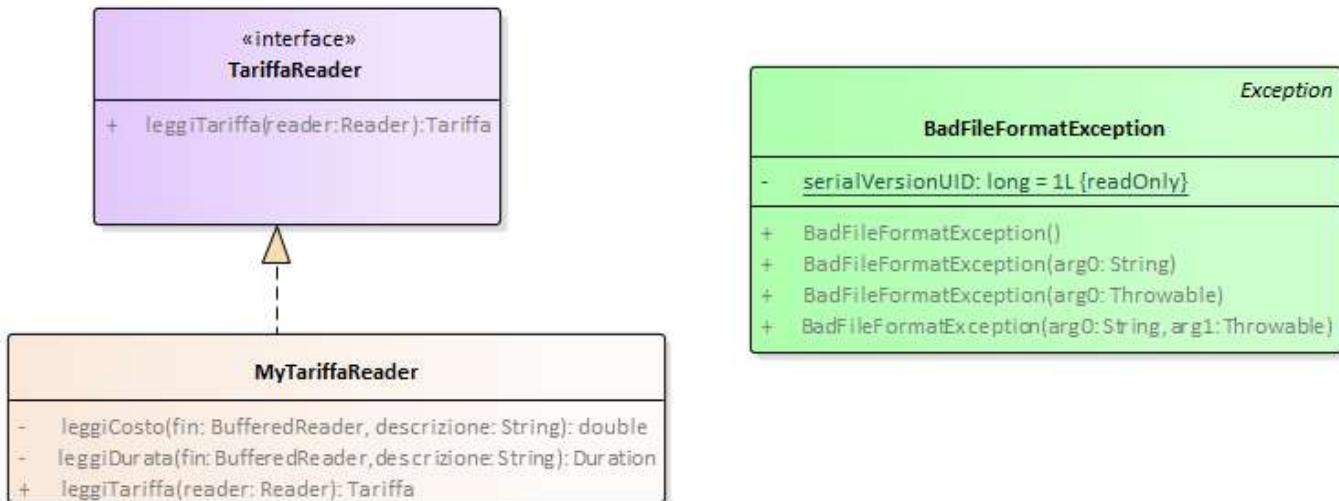
```

Durata unità = 15m
Costo iniziale = 2,00 €
Unità successive = 0,80 €
Cap giornaliero = 35,00 €
12h successive = 16,00 €

```

Si noti che:

- tutte le righe hanno la forma **descrizione = valore**
- la descrizione può essere qualsiasi e può contenere spazi e ogni altro carattere diverso da '='
- per le righe che specificano costi, il valore è un prezzo in euro, formattato secondo lo standard italiano
- per la prima riga, che specifica una durata in minuti, il valore ha la forma di un numero intero, seguito senza spazi intermedi dal carattere 'm'



SEMANTICA:

- a) L'eccezione **BadFormatException** (fornita) esprime l'idea di file formattato in modo scorretto
- b) L'interfaccia **TariffaReader** (fornita) dichiara il metodo **leggiTariffa**, che legge da un **Reader** (ricevuto come argomento) i dati del sistema tariffario, configurando e restituendo l'opportuno oggetto **Tariffa**
- c) La classe **MyTariffaReader** (**da realizzare**) implementa **TariffaReader**: non prevede costruttori e **implementa il metodo leggiTariffa** in accordo alle specifiche sopra definite. In caso di problemi di I/O deve propagare l'opportuna **IOException**, in caso di **Reader** nullo **IllegalArgumentException** e in caso di altri problemi di formato del file **BadFormatException**, il cui messaggio dettagli l'accaduto.

Al fine di modularizzare opportunamente la lettura, **leggiTariffa** si appoggia a due metodi privati:

- **leggiDurata (da implementare)**, che gestisce la prima riga
- **leggiCosto (da implementare)** che gestisce le altre righe

essi restituendo l'opportuno oggetto o lanciano **BadFormatException** se necessario.

Il metodo **leggiTariffa** deve effettuare specifici controlli sul formato delle varie righe, in particolare:

- i. verificando che la descrizione (che precede il simbolo '=') sia esattamente quella richiesta
- ii. che il valore fornito sia di formato e valore coerenti con quanto richiesto, e specificatamente che la durata dell'unità non sia negativa o nulla, e che i vari costi non siano negativi, infiniti o NaN.

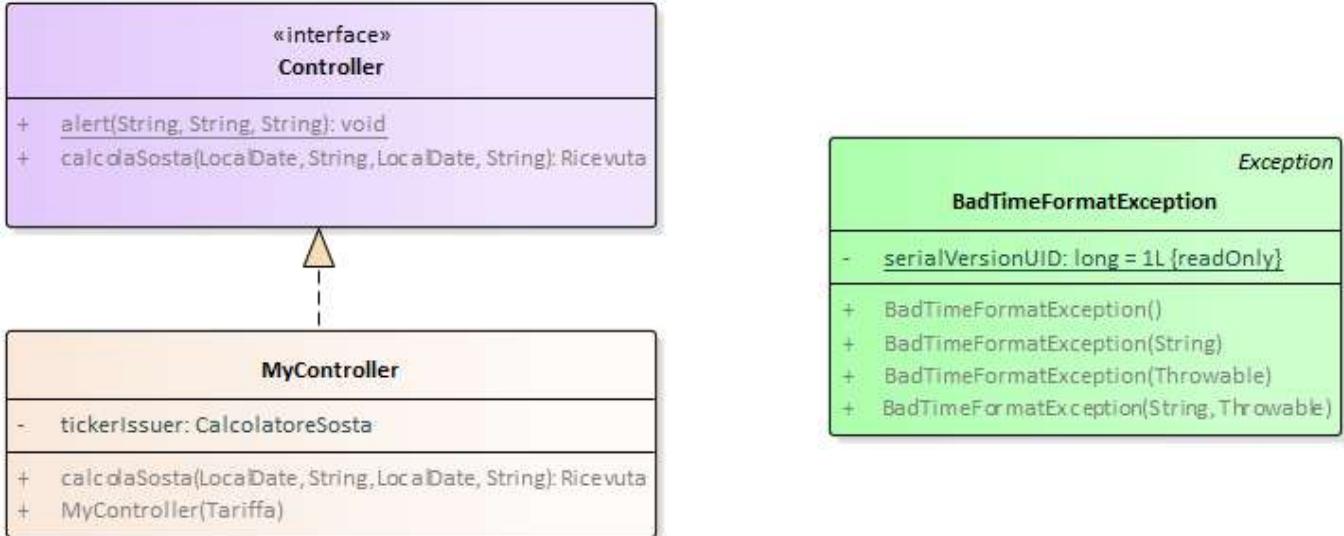
Parte 3 - Grafica

Punti: 12

Package: **cityparking.controller**

[TEMPO STIMATO: 15-20 minuti] (punti 6)

L'interfaccia **Controller** (fornita) dichiara il metodo **calcolaSosta**, che riceve data e ora di inizio/fine sosta: suo compito è, tramite il **CalcolatoreSosta**, ottenere e restituire l'opportuna **Ricevuta**.



A differenza del **CalcolatoreSosta**, il cui metodo `calcola` riceve data e ora di inizio/fine sosta come **LocalDateTime**, il metodo `calcolaSosta` del **Controller** riceve quattro argomenti: due **LocalDate**, che forniscono il giorno di inizio/fine sosta, e due stringhe della forma **HH:MM**, che forniscono la corrispondente ora di inizio/fine sosta.

È compito del metodo `calcolaSosta` verificare che le due stringhe abbiano il corretto formato, lanciando in caso contrario l'apposita **BadTimeFormatException** (fornita).

NB: si ricorda che l'analogia verifica relativa al fatto che l'istante di inizio sosta preceda quello di fine sosta è già svolta da **CalcolatoreSosta**, che lancia in tal caso l'apposita **BadTimeIntervalException** (fornita).

La classe **MyController (da realizzare)** concretizza **Controller** implementando il metodo `calcolaSosta` come sopra specificato: la **Tariffa** su cui lavorare dev'essere passata all'atto della costruzione.

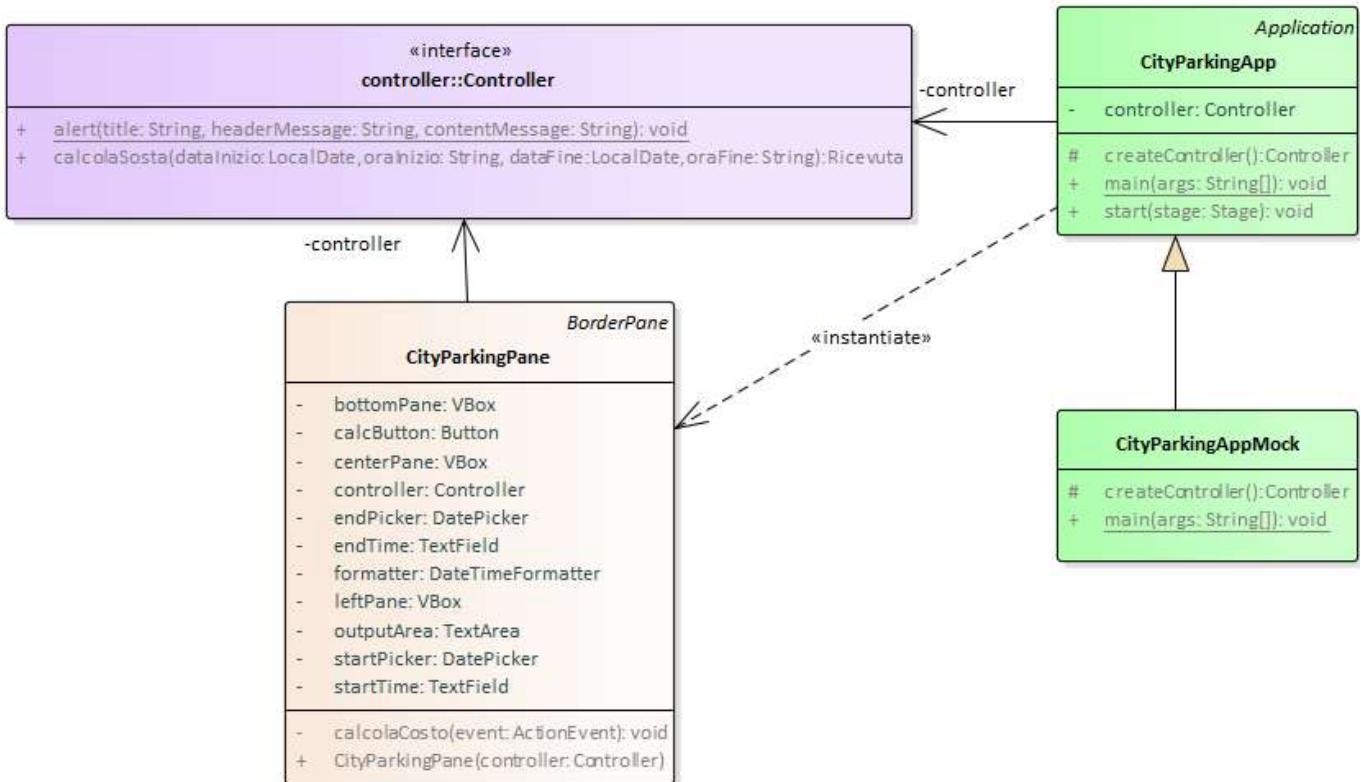
NB: l'interfaccia **Controller** contiene anche il **metodo statico ausiliario alert**, utile per mostrare avvisi all'utente.

Package: **cityparking.ui**

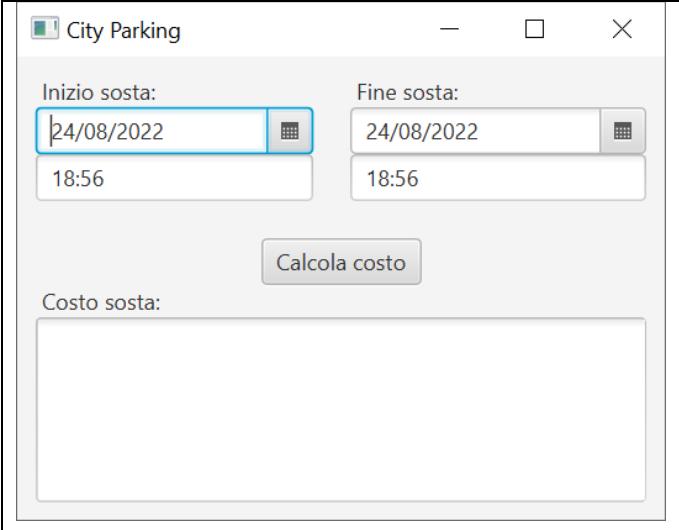
[TEMPO STIMATO: 15-20 minuti] (punti 6)

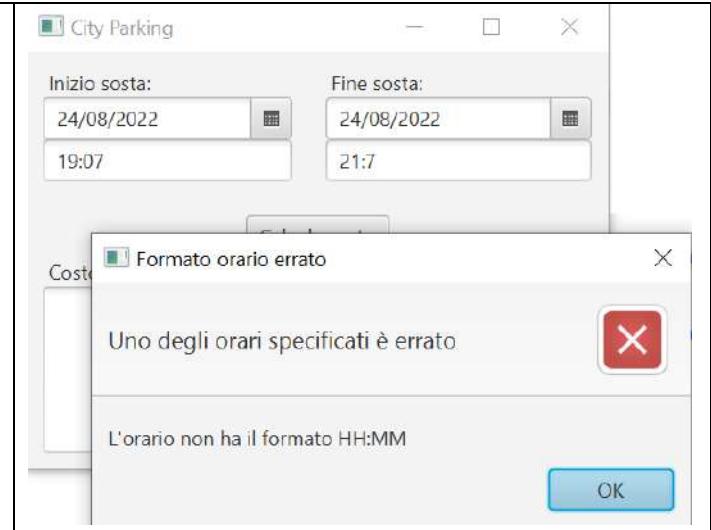
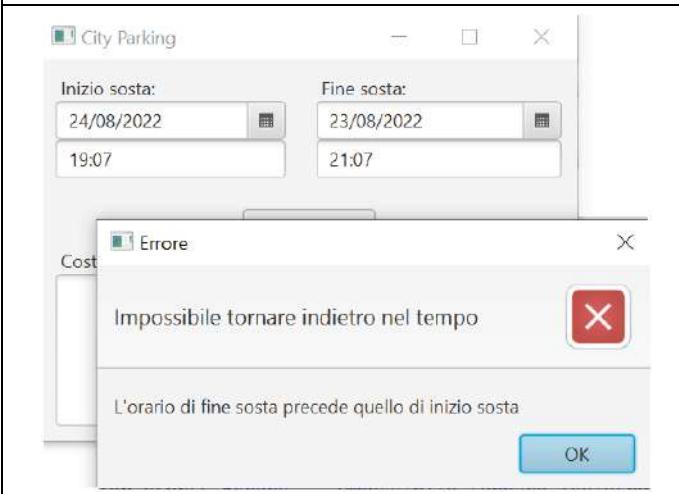
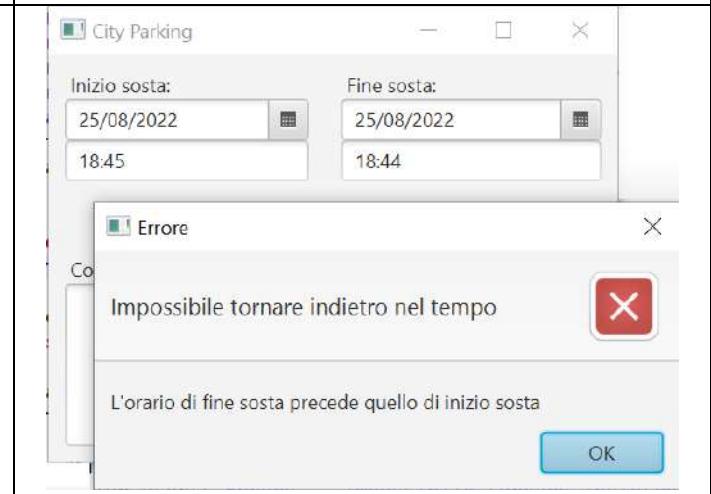
La classe **CityParkingApp** (fornita) costituisce l'applicazione JavaFX che si occupa di aprire i file, creare il controller e incorporare il pannello principale, **CityParkingPane**. Per consentire di collaudare la GUI anche in assenza / in caso di malfunzionamento della parte di persistenza, è possibile avviare l'applicazione mediante la classe **CityParkingAppMock**.

L'interfaccia utente è illustrata nelle figure seguenti e segue il modello sotto illustrato:



L'interfaccia grafica si presenta come segue:

 <p>Fig. 1: la situazione iniziale della GUI</p>	 <p>Fig. 2: un primo caso di funzionamento normale</p>
--	--

	
Fig. 3: un altro caso di funzionamento normale	Fig. 4: errore: orario con formato errato
	
Fig. 5: errore: data/ora finale precedente quella iniziale	Fig. 6:

- in alto a sinistra, un **DatePicker** e un campo di testo consentono di specificare data e ora di inizio sosta
- in alto a destra, un **DatePicker** e un campo di testo consentono di specificare data e ora di fine sosta
- al centro, il pulsante *Calcola costo* effettua il calcolo del costo della sosta per il periodo sopra specificato: se l'istante di fine sosta precede quello di inizio sosta, o se gli orari non seguono il prescritto formato HH:MM secondo l'*uso italiano*, devono essere mostrati opportuni avvisi all'utente (Figg. 4-5).
- in basso, un'area di testo mostra la ricevuta di sosta calcolata, *usando come font il Courier New, 12 punti, grassetto*.

È richiesto che all'avvio dell'applicazione date e orari siano quelli relativi e data e ora correnti.

CityParkingPane è fornito parzialmente realizzato: è presente quasi tutta la parte strutturale, mentre sono da completare la configurazione dei campi di testo e la gestione dell'evento.

In particolare, la parte da completare riguarda:

- 1) il popolamento dei valori di default di date e ore iniziali, nel corretto formato
- 2) l'aggancio dell'opportuno *listener*
- 3) la logica di gestione dell'evento, *che deve catturare le eccezioni eventualmente lanciate dal controller, mostrando all'utente, in tali casi, i messaggi di errore sopra illustrati*.

ESAME DI FONDAMENTI DI INFORMATICA T-2 del 22/7/2022

Proff. E. Denti – R. Calegari – A. Molesini

Tempo a disposizione: 3h30

NOME PROGETTO ECLIPSE: CognomeNome-matricola (es. RossiMario-0000123456)

NOME CARTELLA PROGETTO: CognomeNome-matricola (es. RossiMario-0000123456)

NOME ZIP DA CONSEGNARE: CognomeNome-matricola.zip (es. RossiMario-0000123456.zip)

NOME JAR DA CONSEGNARE: CognomeNome-matricola.jar (es. RossiMario-0000123456.jar)

Si devono consegnare DUE FILE: *l'intero progetto Eclipse e il JAR eseguibile*

Si ricorda che compiti *non compilabili o palesemente lontani da 18/30* NON SARANNO CORRETTI e causeranno la verbalizzazione del giudizio “RESPINTO”

È stato richiesto di sviluppare un'applicazione che consenta all'ufficio personale della ditta *HappyWork* di tenere traccia delle ore svolte/da svolgere da ogni dipendente, come meglio descritto di seguito.

DESCRIZIONE DEL DOMINIO DEL PROBLEMA

Il contratto di lavoro fissa, per ogni dipendente, le **ore di lavoro** da svolgere giorno per giorno in una settimana lavorativa. Esse possono essere anche diverse da un giorno all'altro (ad esempio, una settimana lavorativa di 36h potrebbe essere articolata sia su sei giorni da 6h ciascuno, sia su cinque giorni, di cui due da 9h e tre da 6h, etc.).

Un sistema *marcatempo* registra le ore lavorate: i dipendenti che entrano/escono dal lavoro devono timbrare le entrate, le uscite e le pause pranzo. Più precisamente il marcatempo registra:

- **l'ora di entrata e l'ora di uscita** dal lavoro (anche più volte al giorno, es. mattina, pomeriggio, etc.)
- **l'ora di inizio e l'ora di fine della pausa pranzo.**

Tutti gli orari si intendono nella stessa giornata: non esistono turni a cavallo della mezzanotte (se esistessero sarebbero comunque spezzati come se si entrasse/uscisse in due giornate diverse).

Le **ore lavorate** si calcolano banalmente facendo la differenza fra l'ora di uscita e l'ora di entrata, *detratta la pausa pranzo*. Nei giorni non lavorativi (di norma sabato, domenica e festivi) chiaramente non risultano timbrature nel sistema marcatempo, dato che il dipendente non è stato presente sul posto di lavoro.

Il dipendente può anche prendere **permessi**, che possono essere solo per alcune ore (“riposo compensativo a ore”) o per l’intera giornata (“riposo compensativo”): anche queste voci vengono registrate dal sistema marcatempo.

Al termine di ogni mese, tutte le voci registrate dal marcatempo sono assemblate nel **cedolino**, che riassume i dati del dipendente (nome e cognome, ore da svolgere nei vari giorni della settimana) e le varie timbrature effettuate, nonché i permessi che il dipendente ha chiesto.

Per garantire adeguata flessibilità, il dipendente può svolgere *più o meno ore* di quelle previste in una data giornata, in modo molto libero, grazie all’istituzione della **banca ore**. Ogni giorno si calcola la differenza fra le ore previste e quelle effettivamente svolte: se il dipendente ha svolto meno ore di quelle previste, quelle mancanti saranno detratte dal suo conto in banca ore; viceversa, se ne svolge di più, le ore extra verranno accumulate sul suo conto in banca ore. E’ anche possibile andare “a debito”, ossia avere sulla banca ore saldi negativi: le ore mancanti dovranno poi essere recuperate in periodi successivi. I permessi si considerano come ore non svolte: le corrispondenti ore vengono quindi detratte dal conto in banca ore.

Per ogni dipendente, la banca ogni giorno registra:

- le **ore di lavoro previste** per quella giornata (possono essere diverse da un giorno della settimana all’altro)
- le **ore lavorate** (esclusa, come detto, la pausa pranzo)
- il **saldo ore disponibili sul conto** del lavoratore (positivo o negativo).

Al termine di ogni mese occorre quindi produrre l'**estratto conto** della banca ore, con tutti i dettagli delle ore lavorate e dei permessi fruiti, nonché il **saldo finale** (positivo o negativo) delle ore presenti sul conto.

A differenza del cedolino, l’estratto conto deve elencare tutti i giorni del mese, anche quelli in cui non si è lavorato e non risultano quindi timbrature.

Il file **Cedolino.txt** contiene le timbrature mensili di un dipendente, nel formato descritto più oltre.
L'applicazione dovrà generare a video l'estratto conto della banca ore, nel formato illustrato nelle figure seguenti.

ESEMPIO DI CEDOLINO (da leggere): il formato è descritto in dettaglio più oltre

Dipendente:	Mario Rossi		
Mese di:	GENNAIO 2022		
Ore previste:	6H/9H/6H/9H/6H/0H/0H		
Saldo precedente:	12H07M		
03 Lunedì	08:30	14:30	
04 Martedì	08:30	17:30	
05 Mercoledì	08:30	14:30	Riposo Compensativo
07 Venerdì	08:30	14:30	Riposo Compensativo
10 Lunedì	07:30	13:42	
10 Lunedì	13:42	13:53	Pausa Pranzo
10 Lunedì	13:53	15:45	
11 Martedì	07:30	13:28	
11 Martedì	13:28	13:38	Pausa Pranzo
11 Martedì	13:38	14:46	
11 Martedì	14:46	16:40	Riposo Compensativo a Ore
12 Mercoledì	07:30	13:49	
12 Mercoledì	13:49	13:59	Pausa Pranzo
12 Mercoledì	13:59	14:49	
13 Giovedì	07:30	13:39	
13 Giovedì	13:39	13:49	Pausa Pranzo
13 Giovedì	13:49	16:44	
...			

ESEMPIO DI ESTRATTO CONTO A VIDEO (il formato è descritto in dettaglio più oltre)

The screenshot shows a software interface for managing a monthly time log. On the left, a list of daily entries is displayed, each with start and end times, and calculated values like 'prev', 'eff', 'diff', and 'saldo'. On the right, various summary statistics are shown in boxes:

- Ore previste: 150:00
- Ore caricate: 147:13
- di cui:
 - Ore lavorate: 129:24
 - Riposi a ore: 05:49
 - Riposi: 12:00
- Banca ore:
 - Saldo iniziale: 12:07
 - Saldo finale: -08:29
- Dettagli: (empty box)

PARTE 1 – Modello dei dati: Punti 10

[TEMPO STIMATO: 50-65 minuti]

PARTE 2 – Persistenza: Punti 15

[TEMPO STIMATO: 75-95 minuti]

PARTE 3 – Grafica: Punti 5

[TEMPO STIMATO: 10-20 minuti]

TEMPO STIMATO PER SVOLGERE L'INTERO COMPIUTO:

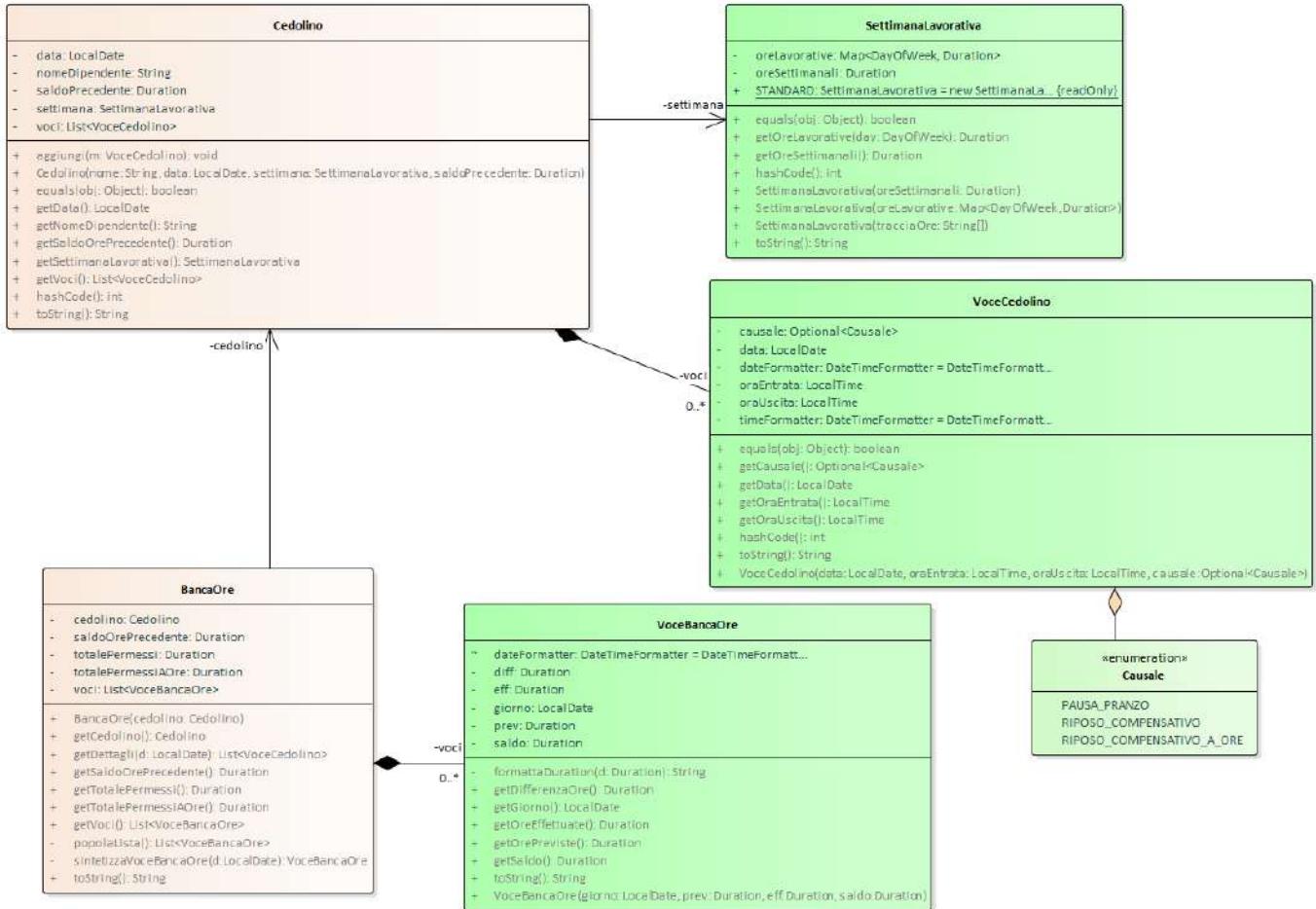
2h15 – 3h

Parte 1 – Modello dei dati

Punti: 10

Package: bancaore.model

[TEMPO STIMATO: 50-65 minuti]



SEMANTICA:

- l'enumerativo **Causale** (fornito) definisce le tre possibili causali relative alle pause e ai permessi: PAUSA PRANZO, RIPOSO COMPENSATIVO (giornaliero), RIPOSO COMPENSATIVO A ORE.
- La classe **SettimanaLavorativa** (fornita) rappresenta le ore lavorative previste per un lavoratore nei sette giorni della settimana, sotto forma di altrettante **Duration**: i tre costruttori accettano o una durata globale (**Duration**) da ripartire convenzionalmente in parti uguali sui cinque giorni lunedì-venerdì, o una mappa che associa a ogni giorno della settimana (**DayOfWeek**) le rispettive durate, o una stringa di sette durate separate da barre nel formato “*nHmM/nHmM.../nHmM*”, in cui ogni “*nHmM*” rappresenta una durata di *n* ore e *m* minuti, da intendersi da lunedì a domenica. **ESEMPIO**, “*6H/9H/6H/9H/6H/0H/0H*” rappresenta una settimana lavorativa di 36 ore, di cui 6 da svolgere il lunedì, mercoledì e venerdì, 9 martedì e giovedì, mentre sabato e domenica sono liberi.
I due metodi `getOreSettimanali` e `getOreLavorative` restituiscono una **Duration** che rappresenta, rispettivamente, il totale delle ore lavorative settimanali previste e le ore previste per il giorno della settimana (**DayOfWeek**) specificato.
- la classe **VoceCedolino** (fornita) rappresenta una voce del cedolino, che contiene data, orario di entrata e uscita, ed eventuale **Causale** (optional): in particolare, le voci con causale istanziata rappresentano pause o permessi, mentre quelle con causale vuota rappresentano ore lavorate. Appositi accessori consentono di estrarre le proprietà rilevanti: opportune `equals`, `hashCode` e `toString` completano l'implementazione.

d) la classe **Cedolino (da completare nel metodo pubblico aggiungi)** rappresenta il cedolino, composto dai dati di intestazione (nome lavoratore, mese e anno a cui il cedolino si riferisce, settimana lavorativa, saldo banca ore al mese precedente) e una lista di **VoceCedolino**. Il costruttore crea un cedolino con lista voci inizialmente vuota, che dovrà poi essere via via popolata tramite il **metodo aggiungi (da implementare)**. Poiché possono essere presenti più voci per la stessa data, la lista delle varie voci deve essere mantenuta ordinata per data crescente e in subordine per ora di entrata crescente: pertanto, *il metodo aggiungi deve garantire che l'inserimento di una nuova voce avvenga in modo da mantenere la lista costantemente ordinata secondo tale criterio*. (NB: non è richiesto verificare che le varie voci siano coerenti, ossia che non si determinino "buchi" o ci siano sovrapposizioni: ci si affida per questo all'hardware del sistema marcatempo).

Il cedolino mantiene al suo interno anche il totale delle ore di permesso (sia giornalieri che a ore) richiesti nel mese. Appositi metodi accessor consentono di estrarre le proprietà rilevanti, mentre idonee *equals*, *hashCode* e *toString* completano l'implementazione.

- e) la classe **VoceBancaOre** (fornita) rappresenta una voce della banca ore, che contiene data, ore previste, ore effettivamente lavorate e saldo attuale della banca alla data specificata. Anche in questo caso, appositi accessor consentono di estrarre le proprietà rilevanti e un'apposita *toString* emette una stringa ben formattata che riassume la voce stessa.
- f) la classe **BancaOre (da completare nel metodo privato sintetizzaVoceBancaOre)** costituisce la banca ore: a tal fine incapsula un **Cedolino**, ricevuto come argomento dal costruttore, e configura la banca a partire dai dati in esso contenuti. Appositi metodi accessor consentono, al solito, di estrarre le proprietà rilevanti. La banca mantiene al proprio interno una lista di **VoceBancaOre**, popolata dal metodo ausiliario *popolaLista (fornito)*, che garantisce la generazione di una e una sola VoceBancaOre per ogni giorno del mese. A sua volta questi si appoggia, per produrre l'oggetto **VoceBancaOre** relativo a una specifica data, al metodo privato ausiliario **sintetizzaVoceBancaOre (da implementare)**, che cura la generazione della VoceBancaOre per il giorno richiesto. A tal fine, esso deve:

- recuperare tutte le voci presenti nel cedolino per la data richiesta (NB: *potrebbero non essercene, dato che non tutte le date del mese sono in generale presenti nel cedolino*)
- calcolare le ore di lavoro globalmente effettuate nella giornata richiesta, escludendo pause pranzo e riposi di qualsiasi tipo
- recuperare le ore previste per quel giorno della settimana
- calcolare la differenza (positiva o negativa) fra le ore previste e quelle effettivamente svolte, aggiornando poi, di conseguenza, il saldo della banca ore
- nel caso di riposi compensativi (giornalieri o ad ore), limitarsi ad aggiornare i relativi totali.

ESEMPIO: per la giornata di lunedì 10 gennaio, in cui il cedolino riporta:

10 Lunedì 07:30	13:42	
10 Lunedì 13:42	13:53	Pausa Pranzo
10 Lunedì 13:53	15:45	

la voce banca ore da sintetizzare deve riportare: ore previste 6 (perché il lunedì la settimana lavorativa del dipendente prevede 6 ore), ore effettivamente svolte 8:04 (6:12 al mattino + 1:52 al pomeriggio), differenza +2:04 e quindi, poiché il saldo precedente era -8:53, nuovo saldo -6:49.

Analogamente, il giorno successivo, martedì 11 gennaio, in cui il cedolino riporta:

11 Martedì	07:30	13:28	
11 Martedì	13:28	13:38	Pausa Pranzo
11 Martedì	13:38	14:46	
11 Martedì	14:46	16:40	Riposo Compensativo a Ore

la voce sintetizzata deve riportare: ore previste 9 (perché è martedì), ore effettivamente svolte 7:06 (5:58 al mattino + 1:08 al pomeriggio), differenza -1:54 e quindi nuovo saldo -8:43.

Parte 2 – Persistenza

Punti: 15

Package: bancaore.persistence

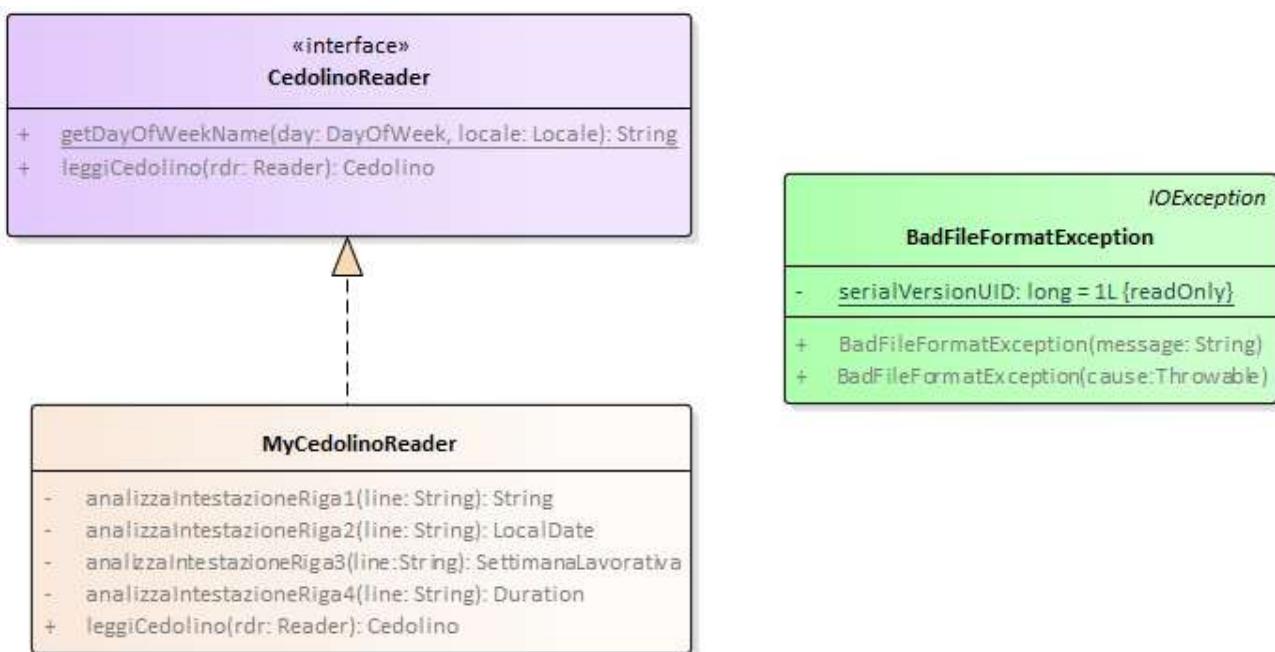
[TEMPO STIMATO: 75-95 minuti]

Come anticipato sopra, il file di testo **Cedolino.txt** contiene le timbrature mensili di un dipendente. Le prime quattro righe costituiscono un'intestazione, che specifica:

- il nome del dipendente
- il mese e anno a cui il cedolino si riferisce
NB: il nome del mese può essere scritto con qualunque mix di maiuscole/minuscole
- le ore lavorative previste nella settimana
- il saldo precedente della banca ore di questo lavoratore, nella forma nHmM (n=ore, m=minuti)

Le righe successive (fra cui possono esservi anche righe vuote, utili soltanto a livello estetico) descrivono ciascuna una voce del cedolino, tramite tre o quattro elementi separati da tabulazioni:

- il giorno del mese (01-31) e della settimana (lunedì, martedì, ...), separati fra loro da uno o più spazi
NB: il giorno della settimana può essere scritto con qualunque mix di maiuscole/minuscole
- gli orari di entrata e di uscita, nel formato italiano SHORT
- eventualmente, per le sole timbrature relative a pause pranzo o riposi, la causale, intesa come stringa che può valere “Riposo compensativo”, “Riposo compensativo a ore” o “Pausa pranzo”, con un qualunque mix di maiuscole/minuscole; le timbrature relative a ore effettivamente lavorate sono prive di causale.



SEMANTICA:

- L'eccezione **BadFormatException** (fornita) esprime l'idea di file formattato in modo scorretto
- L'interfaccia **CedolinoReader** (fornita) dichiara il metodo `leggiCedolino`, che legge da un `Reader` (ricevuto come argomento) i dati di un intero cedolino, configurando e restituendo l'opportuno oggetto **Cedolino**. Tale interfaccia ospita anche il metodo statico di utilità `getDayOfWeekName`, che consente di ottenere la stringa (minuscola) corrispondente al giorno della settimana fornito come argomento, espresso nella cultura locale passata come secondo argomento (**ESEMPIO: passando DayOfWeek.THURSDAY e Locale.ITALY si otterrà in uscita la stringa “giovedì”**)

- c) La classe **MyCedolinoReader** (**da completare**) implementa **CedolinoReader**: non prevede costruttori e implementa il metodo **leggiCedolino** in accordo alle specifiche sopra definite. In caso di problemi di I/O deve propagare l'opportuna **IOException**, in caso di **Reader** nullo **IllegalArgumentException** e in caso di altri problemi di formato dei file **BadFormatException**, il cui messaggio dettagli l'accaduto.

Al fine di modularizzare opportunamente la lettura, **leggiCedolino** si appoggia a quattro metodi privati **analizzaIntestazioneRiga1**, ..., **analizzaIntestazioneRiga4** che elaborano ciascuno una delle quattro righe di intestazione, restituendo l'opportuno oggetto – o lanciando **BadFormatException** se necessario.

Di questi quattro, l'unico da realizzare è analizzaIntestazioneRiga2: gli altri tre sono già implementati.

Successivamente, il ciclo principale di lettura di **leggiCedolino** (**da implementare**) deve leggere ed elaborare le righe corrispondenti alle singole voci del cedolino, costruendo altrettante **VoceCedolino** da aggiungere al **Cedolino** stesso. Nel farlo deve verificare che:

- i. la riga contenga o tre o quattro elementi
- ii. il primo elemento sia costituito da un numero intero fra 1 e 31, seguito da uno o più spazi e dal nome (case-insensitive) del giorno della settimana corrispondente
SUGGERIMENTO: a partire dalla data del cedolino, che contiene già mese e anno, modificare il giorno della settimana con quello dato, verificando poi, a parte, che il nome del giorno della nuova data così ottenuta sia quello atteso – sfruttando il metodo **CedolinoReader.getDayOfWeekName**
OPPURE: combinare gli elementi relativi al giorno con quelli relativi a mese e anno (forniti nell'intestazione del cedolino) per ottenere una data in formato full, di cui sia facile fare il parse
- iii. il secondo e il terzo elemento siano orari correttamente formattati secondo la cultura italiana (NB: provvede **VoceCedolino** a verificare che l'orario di entrata sia antecedente all'orario di uscita)
- iv. l'eventuale quarto elemento, se presente, contenga una delle stringhe (case-insensitive) “PAUSA PRANZO”, “RIPOSO COMPENSATIVO” o “RIPOSO COMPENSATIVO A ORE”, da trasformare nella corrispondente **Causale**.

Parte 3 - Grafica

Punti: 5

Package: **bancaore.controller**

(punti 0)

Il **Controller** (fornito) è organizzato secondo il diagramma UML nella figura seguente: esso lavora su una **BancaOre** ricevuta all'atto della costruzione.

SEMANTICA:

- il costruttore riceve la **BancaOre** su cui opererà e ne ricava, memorizzandolo internamente per comodità, il corrispondente **Cedolino**; calcola poi, tramite il metodo ausiliario **calcolaOreLavorate**, il totale delle ore lavorate (ossia, non di riposo o pause pranzo) risultanti dalle voci della banca per tutti i giorni del mese.
- una nutrita serie di metodi accessori, dall'ovvia denominazione, permette di ottenere sotto forma di stringa già opportunamente formattata nelma forma hh:mm i totali delle ore previste nel mese, di quelle lavorate, di quelle relative ai diversi tipi di riposi utilizzati, nonché la loro somma (ore caricate totali); altri metodi consentono altresì di ottenere i saldi iniziale e

Controller	
-	banca: BancaOre
-	cedolino: Cedolino
-	dateFormatter: DateTimeFormatter = DateTimeFormat...
-	oreLavorate: Duration
+	alert(title:String, headerMessage:String, contentMessage:String): void
+	calcolaOreLavorate(): void
+	Controller(banca: BancaOre)
+	formattaDuration(d: Duration): String
+	getDettagli(dataCorrente: LocalDate): String
+	getMeseAnno(): String
+	getNomeDipendente(): String
+	getSaldoOreFinale(): String
+	getSaldoOreIniziale(): String
+	getTotaleOreCaricate(): String
+	getTotaleOreLavorate(): String
+	getTotaleOrePreviste(): String
+	getTotalePermessiAOre(): String
+	getTotalePermessiGiornaliari(): String
+	getVociBancaOre(): List<VoceBancaOre>
+	getVociCedolino(): List<VoceCedolino>

finale delle ore disponibili sul conto. Il metodo `getDettagli` restituisce una stringa, opportunamente formattata, contenente i dettagli delle voci del cedolino corrispondenti a una specifica data

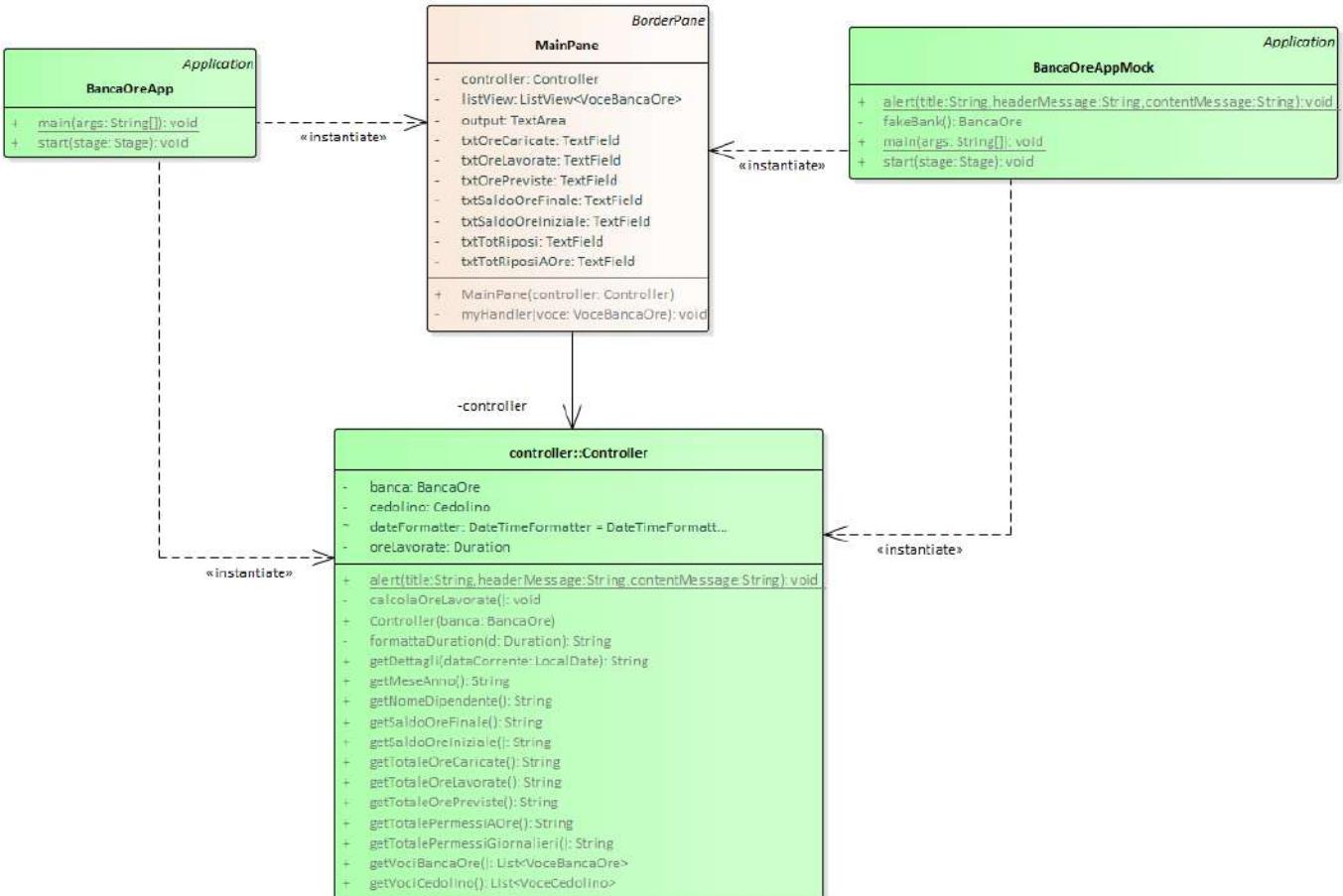
- la classe contiene anche il **metodo statico ausiliario** `alert`, utile per mostrare avvisi all'utente.

Package: `bancaore.ui`

[TEMPO STIMATO: 10-20 minuti] (punti 5)

La classe ***BancaOreApp*** (fornita) costituisce l'applicazione JavaFX che si occupa di aprire i file, creare il controller e incorporare il ***MainPane***. Per consentire di collaudare la GUI anche in assenza / in caso di malfunzionamento della parte di persistenza, è possibile avviare l'applicazione mediante la classe ***BancaOreAppMock***.

L'interfaccia utente è illustrata nelle figure seguenti e segue il modello sotto illustrato:



L'interfaccia grafica si presenta come segue:

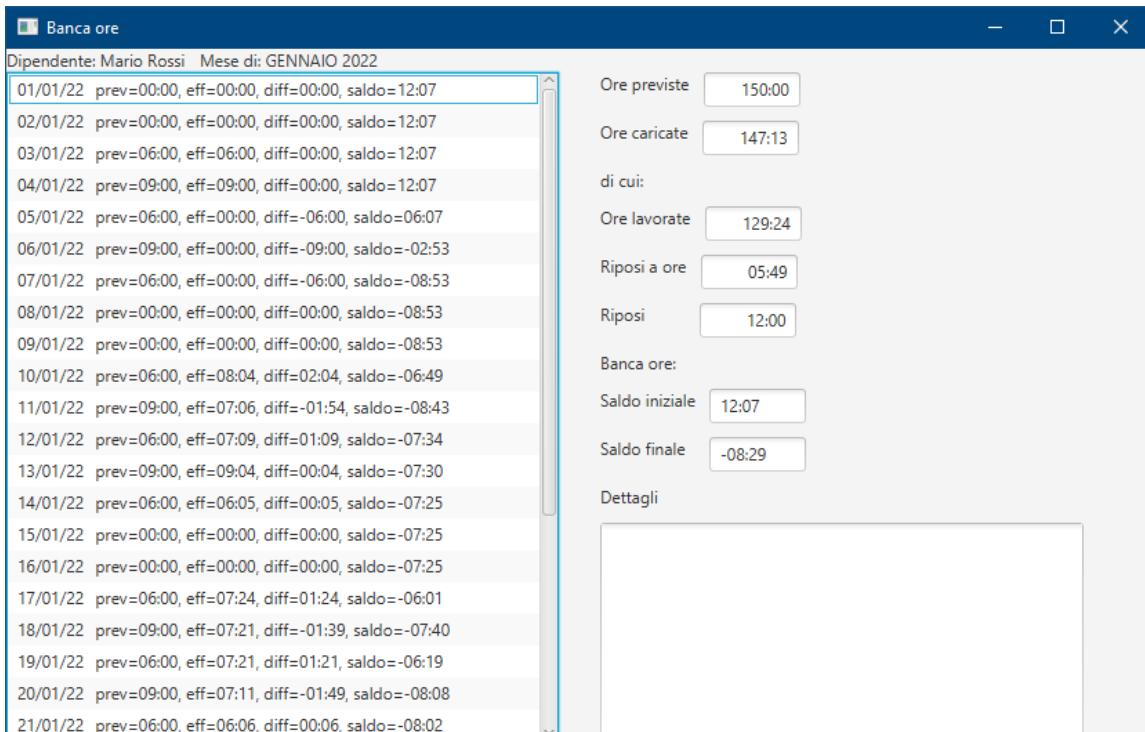


Fig. 1: la situazione iniziale della GUI

- in alto, due etichette riportano rispettivamente il nome del dipendente e, in maiuscolo, mese e anno a cui l'estratto conto si riferisce
- a sinistra, una **ListView** mostra le voci della banca ore, ordinate per data crescente

- a destra, una serie di etichette e campi di testo (non editabili) mostrano i dati rilevanti (ore previste nel mese, ore caricate con relativo dettaglio, saldi iniziale e finale delle ore disponibili sul conto. Subito sotto, un'area di testo funge da dispositivo di uscita per mostrare, a richiesta dell'utente, i dettagli della voce selezionata.

L'utente può interagire unicamente selezionando le varie righe sulla sinistra: in risposta, nell'area di testo vengono mostrate le singole voci del cedolino che "scorporano" la voce della banca ore selezionata.

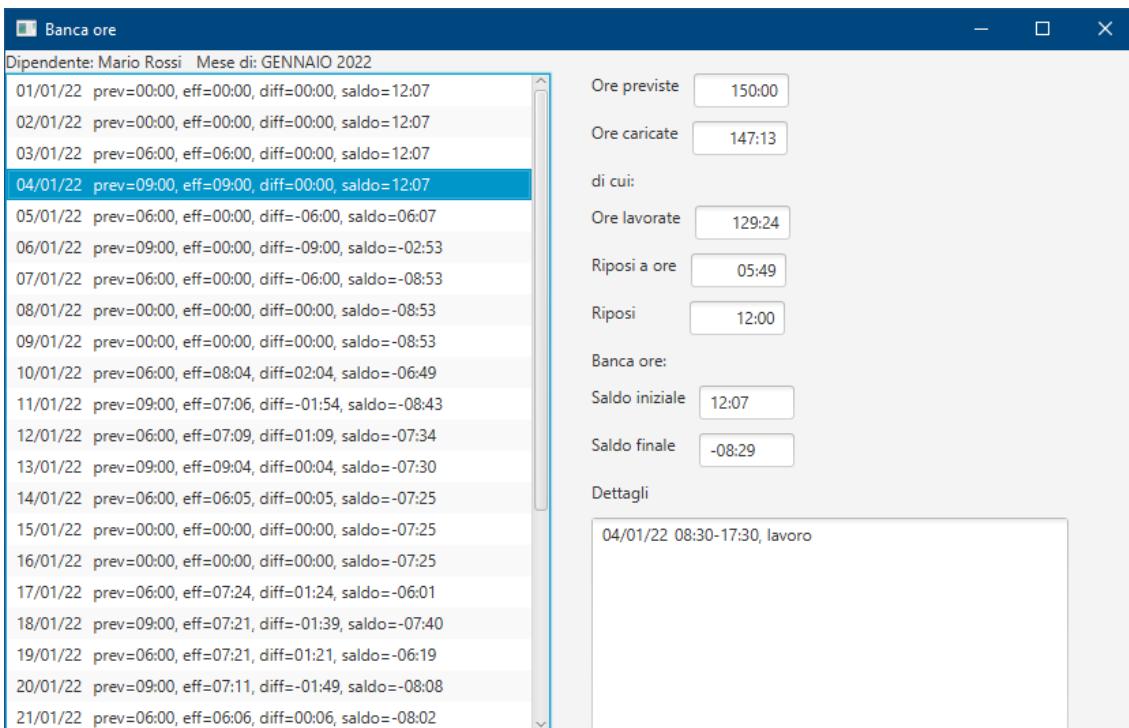


Fig. 2: i dettagli relativi a una giornata con una sola voce di cedolino

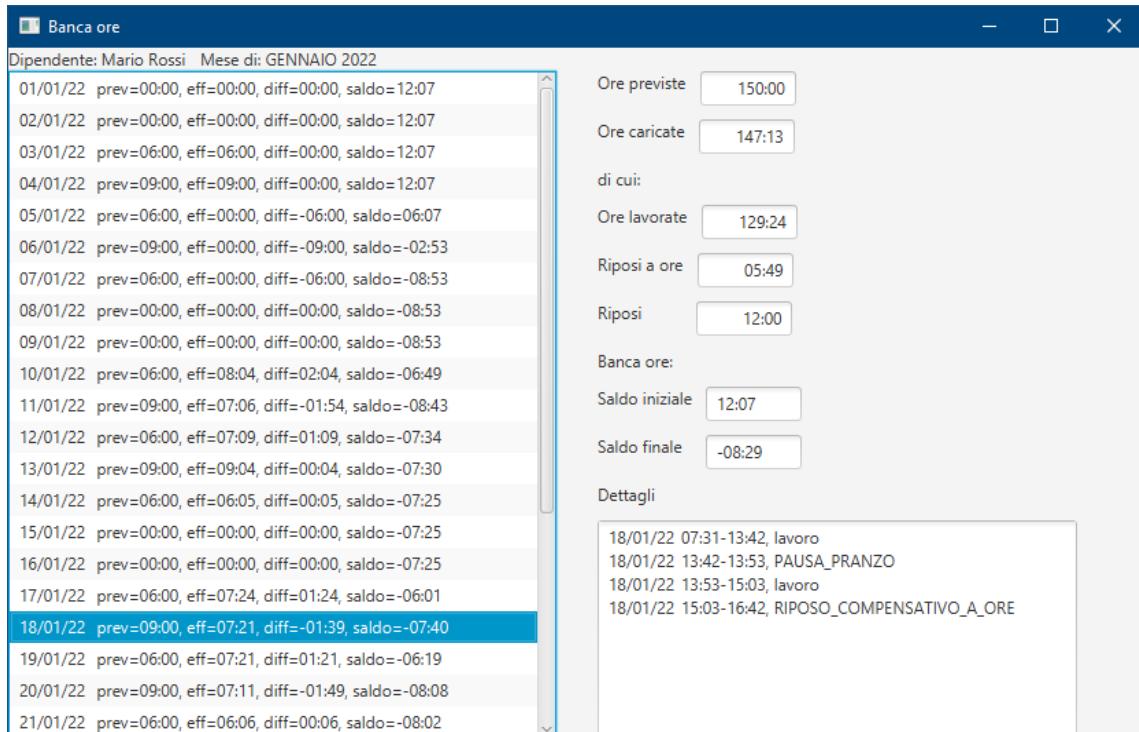


Fig. 3: i dettagli relativi a una giornata con più voci di cedolino

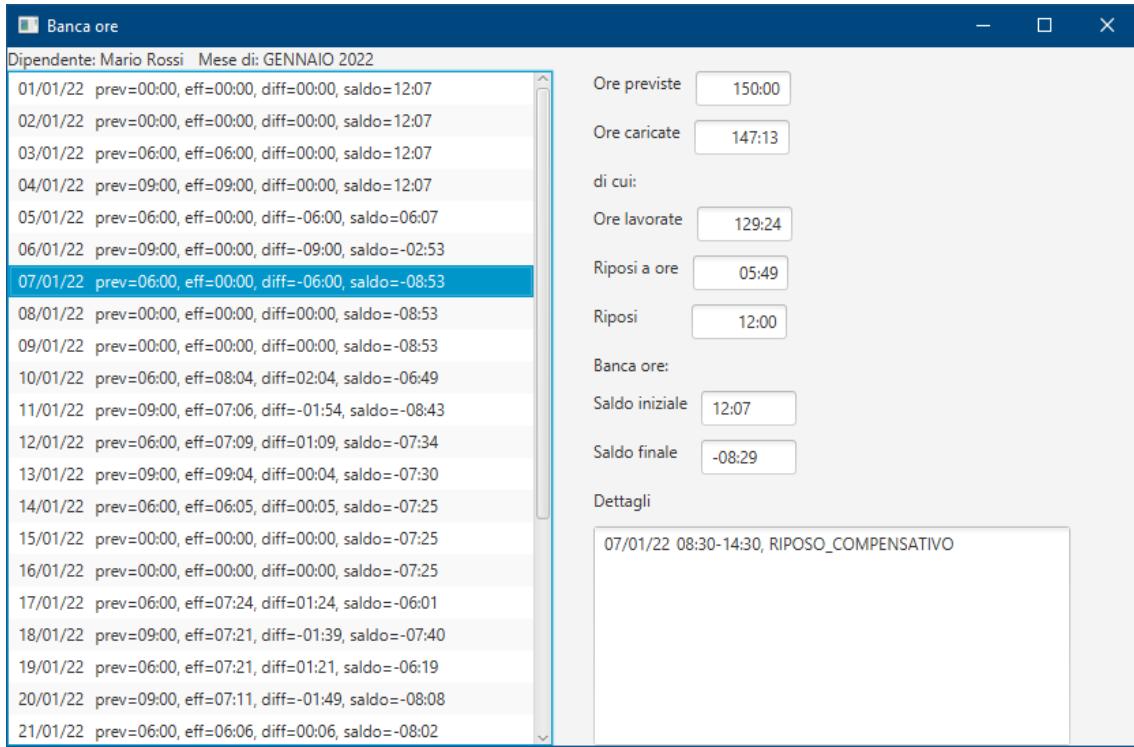


Fig. 4: i dettagli relativi a una giornata con una sola voce di riposo compensativo

Se il cedolino non contiene voci per la data selezionata, viene mostrata opportuna indicazione:

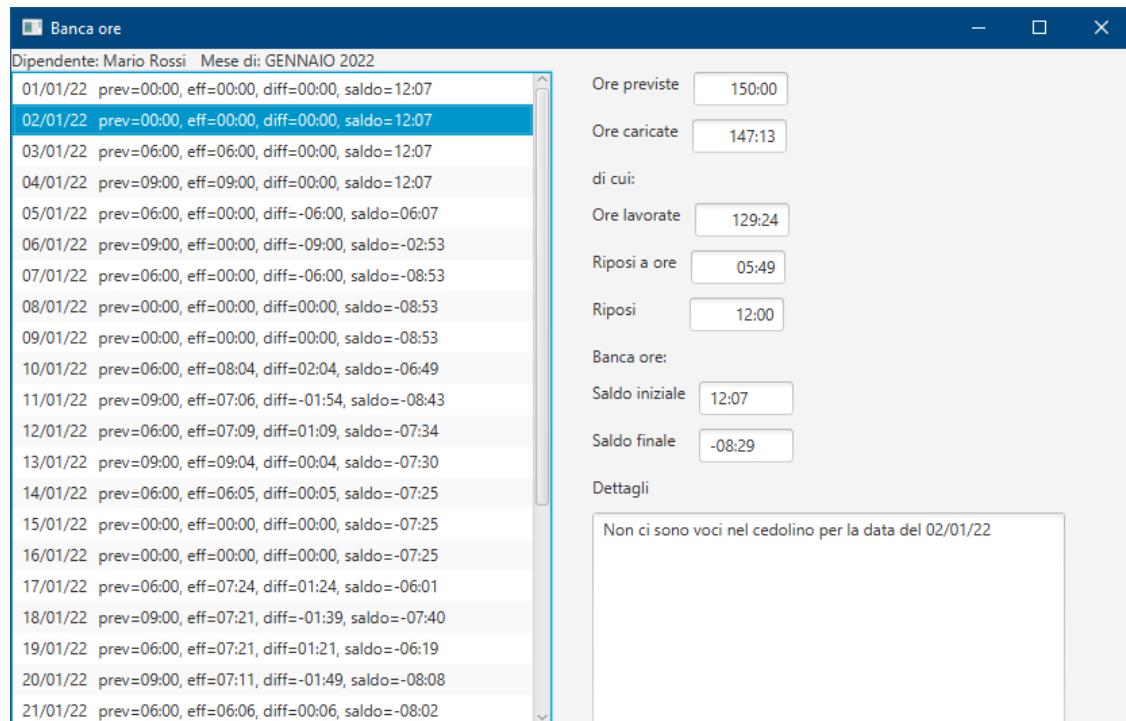


Fig. 5: i dettagli relativi a una giornata per la quale il cedolino non conteneva voci

Il MainPane è fornito quasi totalmente realizzato: è presente tutta la parte strutturale, mentre sono da completare la configurazione della ListView e la gestione dell'evento.

In particolare, la parte da completare riguarda:

- 1) il popolamento della **ListView**
- 2) l'aggancio alla **ListView** dell'opportuno listener
- 3) la logica di gestione dell'evento

Cose da ricordare

- salva costantemente il tuo lavoro: l'informatica a volte può essere “subdolamente ostile”..
- in particolare: se ora compila e stai per fare modifiche, salva la versione attuale (non si sa mai)

Checklist di consegna

- Hai fatto un **JAR eseguibile**, che contenga cioè l'indicazione del main?
- Hai controllato che **si compilì e ci sia tutto?** [NB: non includere il PDF del testo]
- Hai **rinominato IL PROGETTO**, lo ZIP e il JAR esattamente come richiesto?
- Hai **chiamato** la cartella del progetto esattamente come richiesto?
- **Hai fatto un unico file ZIP (NON .7z, rar o altri formati) contenente l'intero progetto?**
In particolare, ti sei assicurato di aver incluso tutti i file .java (e non solo i .class)?
- **Hai consegnato DUE file distinti, ossia lo ZIP col progetto e il JAR eseguibile?**
- Su EOL, hai **premuto** il tasto “CONFERMA” per inviare il tuo elaborato?

ESAME DI FONDAMENTI DI INFORMATICA T-2 del 5/7/2022

Proff. E. Denti – R. Calegari – A. Molesini

Tempo a disposizione: 3h30

NOME PROGETTO ECLIPSE: CognomeNome-matricola (es. RossiMario-0000123456)

NOME CARTELLA PROGETTO: CognomeNome-matricola (es. RossiMario-0000123456)

NOME ZIP DA CONSEGNARE: CognomeNome-matricola.zip (es. RossiMario-0000123456.zip)

NOME JAR DA CONSEGNARE: CognomeNome-matricola.jar (es. RossiMario-0000123456.jar)

Si devono consegnare DUE FILE: *l'intero progetto Eclipse e il JAR eseguibile*

Si ricorda che compiti non compilabili o palesemente lontani da 18/30 NON SARANNO CORRETTI e causeranno la verbalizzazione del giudizio “RESPINTO”

Si vuole sviluppare un'applicazione che consenta di monitorare la carriera universitaria di studenti, mostrando media pesata e crediti acquisiti, oltre a effettuare controlli generali sulla correttezza della carriera stessa.

DESCRIZIONE DEL DOMINIO DEL PROBLEMA

Un'**attività formativa** rappresenta un insegnamento universitario caratterizzato da *numero di codice* (univoco), *denominazione* (che può contenere spazi) e *numero di crediti* (non necessariamente intero).

Quando, in una certa *data*, lo studente sostiene un **esame**, all'attività formativa viene associato un **voto**, che può essere o un *numero fra 18 e 30* (quest'ultimo eventualmente con lode) o un giudizio di *idoneità*; in caso di esito negativo, viene attribuito uno dei giudizi *ritirato* o *respinto*. Ogni esame superato comporta l'acquisizione dei relativi *crediti*. È possibile ri-sostenere un esame solo se in precedenza l'esito è stato negativo, mentre non è consentito ripetere un esame già sostenuto con esito positivo.

Al termine della carriera, lo studente deve sostenere la **prova finale**: essa può quindi essere sostenuta solo *in data successiva* ad ogni altro esame.

In qualunque momento la carriera dello studente è quindi caratterizzata da:

- **Lista degli esami sostenuti** (sia con esito positivo che con esito negativo)
- **Crediti acquisiti** (concorrono all'acquisizione dei crediti solo gli esami superati con *esito positivo*)
- **Media pesata** (concorrono all'acquisizione dei crediti solo gli esami con voto, superati con *esito positivo*)

La media pesata si calcola secondo la nota formula $MP = \frac{\sum v_i * p_i}{\sum p_i}$, essendo v_i i voti e p_i i pesi (crediti) degli esami.

Una serie di file di testo (i cui nomi non sono specificati né rilevanti) descrive possibili carriere di studenti, nel formato descritto più oltre.

TEMPO STIMATO PER SVOLGERE L'INTERO COMPITO: **2h15 – 3h**

PARTE 1 – Modello dei dati: Punti 12 [TEMPO STIMATO: 55-70 minuti]

PARTE 2 – Persistenza: Punti 11 [TEMPO STIMATO: 50-70 minuti]

PARTE 3 – Grafica: Punti 7 [TEMPO STIMATO: 30-40 minuti]

JAVAFX – Parametri run configuration nei LAB

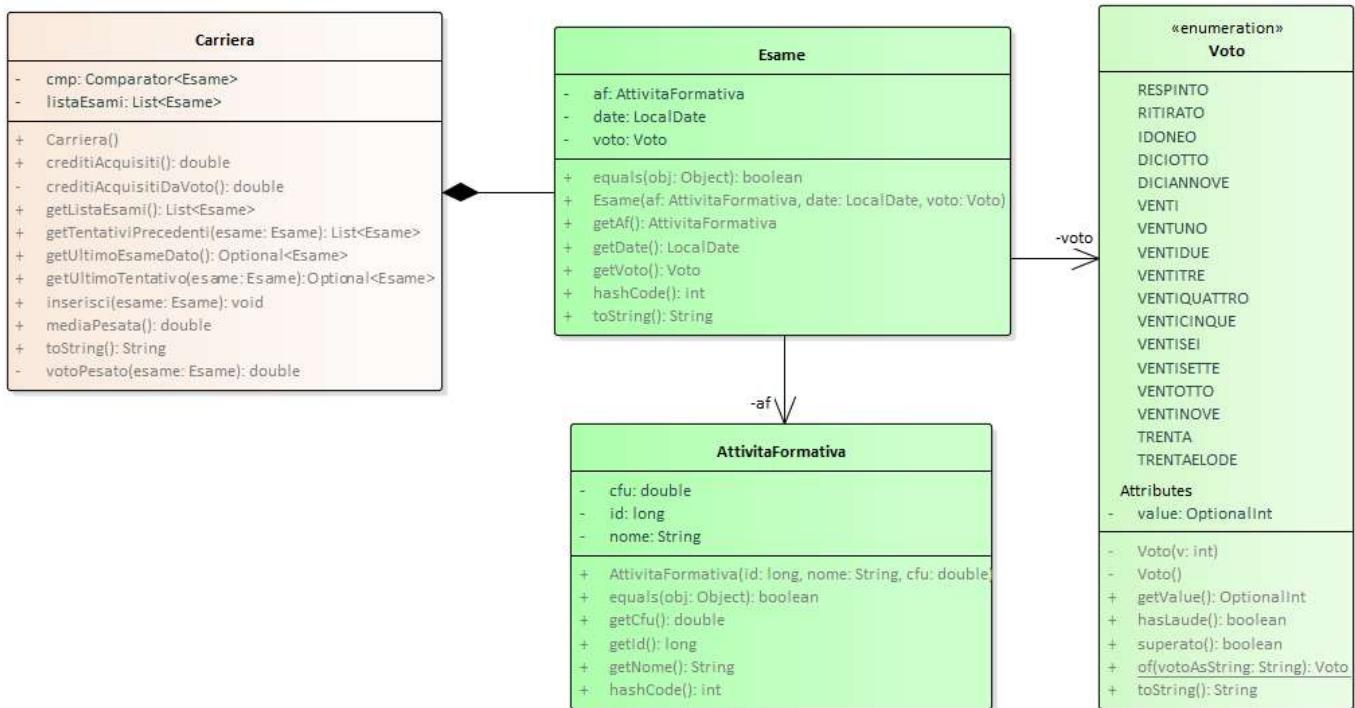
```
--module-path "C:\applicativi\moduli\javafx-sdk-17.0.2\lib"  
--add-modules javafx.controls
```

Parte 1 – Modello dei dati

(punti: 12)

Package: mediaesami.model

[TEMPO STIMATO: 55-70 minuti]



SEMANTICA:

- la classe **AttivitaFormativa** (fornito) descrive un'attività formativa come da dominio del problema
- l'enumerativo **Voto** (fornito) rappresenta i possibili valori di voto o giudizio, come da dominio del problema; in particolare, sono definite sia costanti enumerative per ogni voto fra 18 e 30L, associate al corrispondente valore numerico (per 30L il valore è 30), nonché costanti enumerative per i tre giudizi respinto, ritirato e idoneo, non associate ad alcun valore numerico. Il metodo `getValue` estrae il valore intero associato, se esistente, mentre i due metodi `hasLaude` e `superato` sono veri rispettivamente nel solo caso del 30L, e per i voti positivi, ossia diversi da *ritirato* o *respinto*. Il factory method statico `of` restituisce la giusta istanza dell'enumerativo in base alla stringa ricevuta, che può essere o il valore numerico fra "18" e "30", oppure una delle stringhe "30L", "ID", "RT", "RE". Dualmente, un'apposita `toString` emette la stringa appropriata per ogni valore dell'enumerativo.
- La classe **Esame** (fornito) rappresenta un esame, che associa a un'attività formativa un voto ottenuto in una ben precisa data. Il costruttore riceve tali dati, che sono poi recuperabili tramite gli appositi accessori; idonee `toString`, `equals`, `hashcode` completano l'implementazione.
- la classe **Carriera (da implementare)** mantiene al proprio interno la lista degli esami sostenuti, ordinata per data crescente e in subordine per codice univoco dell'attività formativa. Il costruttore crea una carriera inizialmente vuota, che verrà poi riempita via via tramite il metodo `inserisci`. Più precisamente:
 - il metodo `inserisci` inserisce un esame in carriera, a condizione che un esame per la stessa attività formativa non sia già stato sostenuto in precedenza con esito positivo (altrimenti, deve lanciare **IllegalArgumentException** con opportuno messaggio d'errore); deve inoltre, ovviamente, verificare che l'argomento ricevuto non sia nullo, lanciando in tal caso **IllegalArgumentException** con apposito messaggio d'errore. Nel solo caso in cui l'esame da inserire sia la prova finale (aspetto verificabile unicamente dalla descrizione, che in tal caso conterrà la dizione "PROVA FINALE"),

deve altresì verificare che la data della stessa sia posteriore a quella di ogni altro esame in carriera – anche in questo caso, lanciando **IllegalArgumentException** con adeguato messaggio

- il metodo `getListaEsami` restituisce semplicemente la lista ordinata degli esami attualmente presenti in carriera
- il metodo `getTentativiPrecedenti` restituisce la lista ordinata dei soli esami (con esito ovviamente negativo) sostenuti in precedenza per la stessa attività formativa dell'esame ricevuto come argomento (se non ce ne sono, restituisce una lista vuota)
- il metodo `getUltimoTentativo` restituisce, se esiste (per questo il tipo di ritorno è un **Optional**), il più recente degli esami (con esito negativo) sostenuti in precedenza per la stessa attività formativa dell'esame ricevuto come argomento
- il metodo `creditiAcquisiti` restituisce la somma dei crediti acquisiti da esami superati con esito positivo (sia con voto numerico, sia con giudizio di idoneità)
- il metodo `creditiAcquisitiDaVoto` restituisce la somma dei crediti acquisiti da esami superati con esito positivo con solo voto numerico (utile per calcolare la media pesata)
- il metodo `mediaPesata` restituisce la media pesata dei voti (numerici) fin qui ottenuti
- un'apposita `toString` emette una stringa corrispondente alla lista di tutti gli esami (superati o meno), separati da “a capo”, rispettando l'ordine di lista.

Parte 2 – Persistenza

(punti: 11)

Package: `mediaesami.persistence`

[TEMPO STIMATO: 50-70 minuti]

Sono forniti un certo numero di file di testo, tutti strutturati secondo il medesimo formato (possono contenere righe vuote). L'applicazione principale provvede da sé a recuperarne l'elenco dalla directory corrente e richiamarne ciclicamente la lettura: pertanto il candidato deve occuparsi unicamente di implementare il classico reader per un singolo file.

Ogni riga è organizzata in una serie di elementi separati da una o più tabulazioni. Tre di essi sono sempre presenti:

- codice identificativo dell'attività formativa (un intero long)
- denominazione dell'attività formativa (può contenere spazi e ogni altro carattere diverso da tabulazione)
- numero di crediti (un valore reale, scritto in formato italiano, con la virgola come separatore decimale)

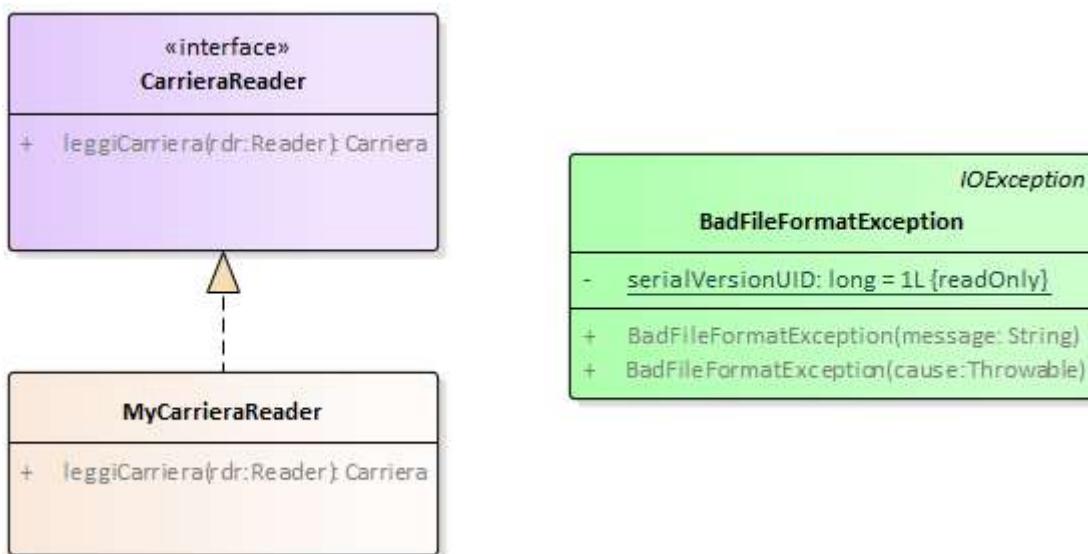
Solo se l'esame è stato sostenuto, sono presenti due ulteriori elementi:

- data in cui l'esame è stato sostenuto (nel particolare formato GG/MM/AAAA)
NB: tale formato non è fra quelli standard previsti dai formattatori, in quanto il formato SHORT prevede l'anno su due sole cifre, mentre i formati che prevedono l'anno su quattro cifre includono anche altri elementi, qui non presenti. Si suggerisce un attento studio della classe `Esame` per trarre ispirazione.. ☺
- voto (un numero fra 18 e 30, oppure una delle sigle 30L, ID, RT, RE rispettivamente per trenta e lode, idoneo, ritirato, respinto)

ESEMPIO:

27991	ANALISI MATEMATICA T-1	9,0	12/1/2020	RT
27991	ANALISI MATEMATICA T-1	9,0	10/2/2020	22
28004	FONDAMENTI DI INFORMATICA T-1	12,0	13/2/2020	24
29228	GEOMETRIA E ALGEBRA T	6,0	18/1/2020	26
26337	LINGUA INGLESE B-2	6,0	18/6/2020	ID
27993	ANALISI MATEMATICA T-2	6,0	10/6/2020	RE
27993	ANALISI MATEMATICA T-2	6,0	02/7/2020	RT
27993	ANALISI MATEMATICA T-2	6,0	22/7/2020	23
28006	FONDAMENTI DI INFORMATICA T-2	12,0	21/7/2020	27
28011	RETI LOGICHE T	6,0	22/2/2022	22
28012	CALCOLATORI ELETTRONICI T	6,0	22/1/2021	RT
28012	CALCOLATORI ELETTRONICI T	6,0	22/2/2021	20

30780	FISICA GENERALE T	9,0	12/2/2021	25
28032	MATEMATICA APPLICATA T	6,0	02/2/2021	24
28027	SISTEMI INFORMATIVI T	9,0	03/6/2021	28
28030	ECONOMIA E ORGANIZZAZIONE AZIENDALE T	6,0	02/7/2021	RE
28030	ECONOMIA E ORGANIZZAZIONE AZIENDALE T	6,0	22/7/2021	24
28029	ELETTRONICA T	6,0	02/9/2021	26
28014	FONDAMENTI DI TELECOMUNICAZIONI T	9,0	15/9/2021	30
28020	SISTEMI OPERATIVI T	9,0	12/1/2022	24
28015	CONTROLLI AUTOMATICI T	9,0	13/1/2021	25
28016	ELETTRONICA T	6,0	10/2/2021	22
28024	RETI DI CALCOLATORI T	9,0	05/2/2021	23
28659	TECNOLOGIE WEB T	9,0	12/6/2021	25
28021	INGEGNERIA DEL SOFTWARE T	9,0	24/6/2021	27
17268	PROVA FINALE	3,0		
28074	TIROCINIO T	6,0	27/9/2021	ID
88324	AMMINISTRAZIONE DI SISTEMI T	6,0	13/7/2021	29
88325	LABORATORIO DI SICUREZZA INFORMATICA T	6,0		



SEMANTICA:

- a) L'eccezione **BadFormatException** (fornita) esprime l'idea di file formattato in modo scorretto
- b) L'interfaccia **CarrieraReader** (fornita) dichiara il metodo `leggiCarriera`, che legge da un **Reader** (ricevuto come argomento) i dati di una carriera, configurando e restituendo l'opportuno oggetto **Carriera**.
IMPORTANTE: poiché la **Carriera** è composta di **Esami**, le righe contenenti la sola descrizione di attività formative, ossia quelle senza data e voto, devono essere comunque verificate a livello di formato ma **ignorate** per quanto riguarda l'inserimento in **carriera**, in quanto non descrivono un esame sostenuto.
- c) La classe **MyCarrieraReader (da realizzare)** implementa **CarrieraReader**: non prevede costruttori, si limita a implementare il metodo `leggiCarriera` come sopra specificato. In caso di problemi di I/O deve essere propagata l'opportuna **IOException**, mentre in caso di **Reader** nullo o altri problemi di formato dei file deve essere lanciata una opportuna **BadFormatException**, il cui messaggio dettagli l'accaduto.

In particolare, il reader deve verificare, lanciando **BadFormatException** in caso contrario, che:

- i. la riga contenga tre o cinque elementi
- ii. il primo elemento sia un intero long
- iii. il terzo elemento sia un valore numerico reale formattato, per quanto attiene alla parte decimale, secondo le convenzioni italiane

nonché, dove siano presenti anche gli altri due elementi:

- iv. la data sia correttamente formattata secondo la struttura GG/MM/AAAAA (vedi nota sopra)
- v. il voto sia un numero intero compreso fra 18 e 30, oppure una delle sigle sopra specificate

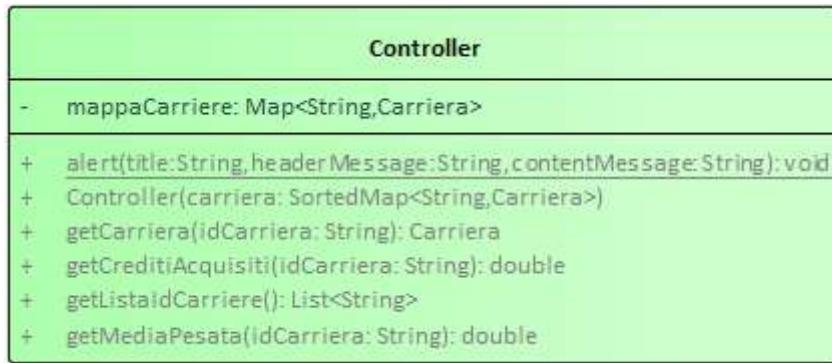
Parte 3

(punti: 7)

Package: mediaesami.controller

(punti 0)

Il **Controller** (fornito) è organizzato secondo il diagramma UML nella figura seguente: esso mantiene internamente una mappa <stringa, carriera> che associa a ogni **Carriera** una opportuna stringa identificativa univoca.



SEMANTICA:

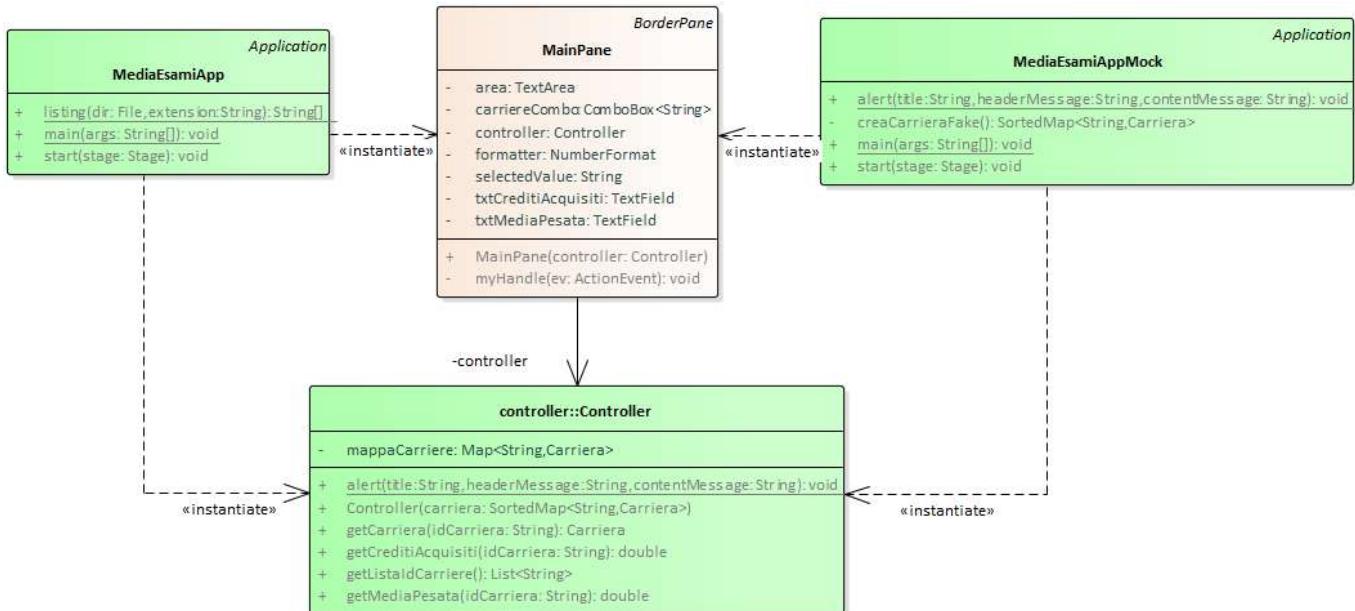
- il costruttore riceve la mappa <stringa,carriera> su cui opererà
- il metodo `getCarriera` restituisce l'oggetto **Carriera** corrispondente alla chiave stringa ricevuta
- il metodo `getListIdCarriere` restituisce una lista contenente le stringhe identificative delle varie carriere
- i due metodi `getMediaPesata` e `getCreditiAcquisiti` restituiscono rispettivamente la media pesata e i crediti acquisiti della carriera il cui identificativo (chiave stringa) è ricevuto come argomento
- la classe contiene anche il **metodo statico ausiliario alert**, utile per mostrare avvisi all'utente.

Package: mediaesami.ui

[TEMPO STIMATO: 30-40 minuti] (punti 7)

La classe **MediaEsamiApp** (fornita) costituisce l'applicazione JavaFX che si occupa di aprire i file, creare il controller e incorporare il **MainPane**. Per consentire di collaudare la GUI anche in assenza / in caso di malfunzionamento della parte di persistenza, è possibile avviare l'applicazione mediante la classe **MediaEsamiAppMock**.

L'interfaccia utente è illustrata nelle figure seguenti e segue il modello sotto illustrato:



L'interfaccia grafica si presenta come segue:

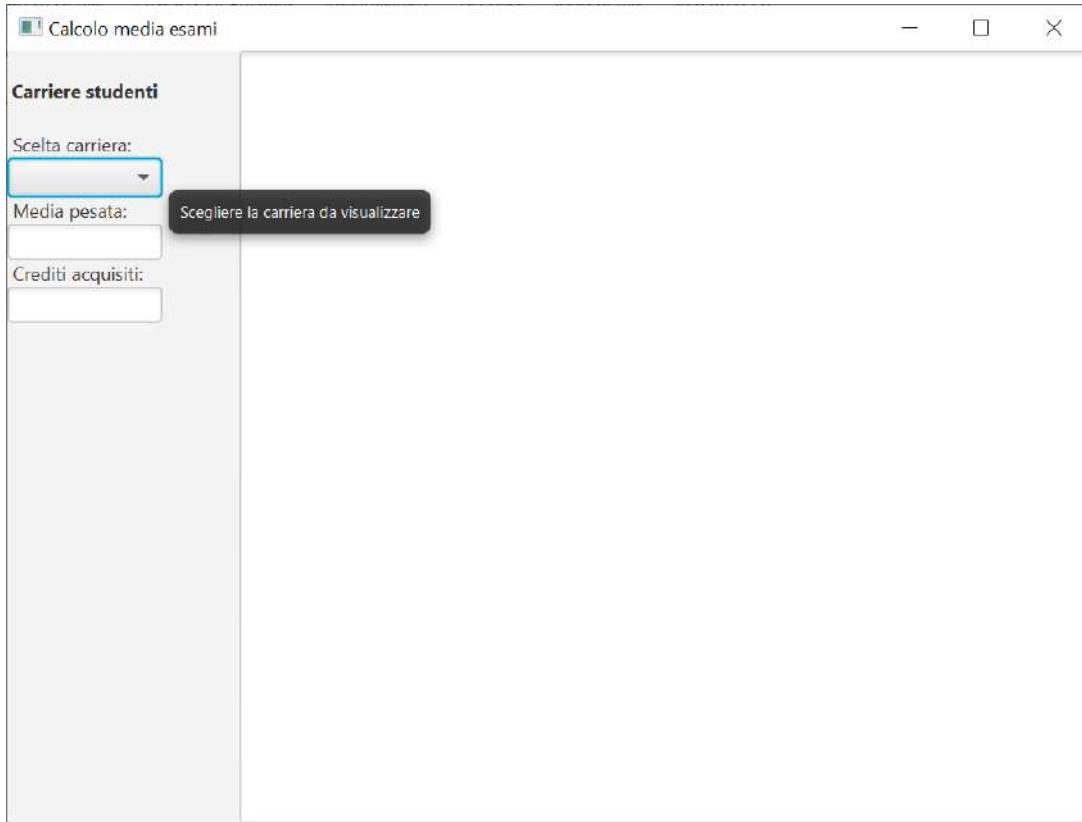
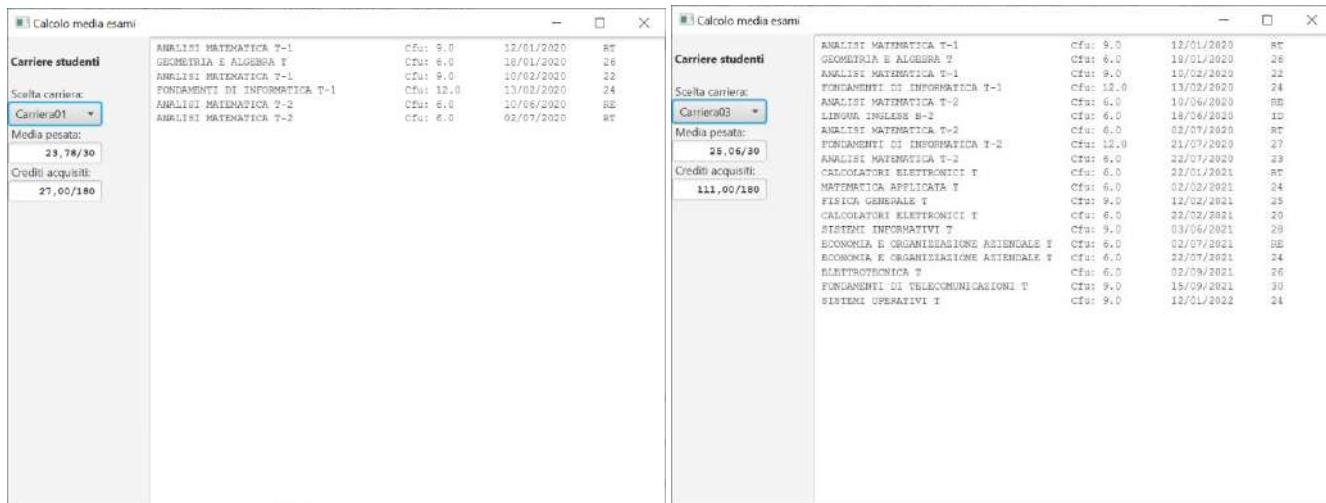


Fig. 1: la situazione iniziale della GUI, con combo vuota e nessuna selezione ancora effettuata

- a sinistra, una combo elenca le carriere disponibili; sotto, due campi di testo (inizialmente vuoti e non scrivibili dall'utente) mostrano rispettivamente le media pesata e i crediti acquisiti relativi alla carriera selezionata
- a destra, un'area di testo (inizialmente vuota e non scrivibile dall'utente) mostra i dettagli della carriera selezionata, ossia la lista degli esami in essa contenuti.

L'utente può interagire unicamente selezionando la carriera desiderata nella combo: immediatamente, i campi di testo sottostanti e l'area sulla destra vengono popolati con i dati corrispondenti



Figg. 2 / 3: la GUI dopo la selezione della prima carriera (a sinistra) e di una carriera successiva (a destra)

Il MainPane è fornito *parzialmente realizzato*: è presente buona parte dell'impostazione strutturale, mentre sono da completare la configurazione di alcuni componenti e la gestione degli eventi.

La classe **MainPane** (da completare) estende **BorderPane** e prevede:

- 1) a sinistra, una **VBox** con la combo e i due campi di testo, oltre alle opportune etichette
- 2) a destra, una **VBox** con la sola area di output.

La **parte da completare** riguarda:

- 1) la configurazione iniziale dei formattatori numerici
- 2) il popolamento della **combo** per la scelta della carriera
- 3) l'aggancio alla **combo** dell'opportuno listener encapsulato nel metodo ausiliario **myHandler**
- 4) la configurazione dei **campi di testo** non editabili che devono utilizzare il font Courier grassetto 11 pt
- 5) la configurazione dell'area di testo non editabile che deve utilizzare il font Courier (normale) 11 pt
- 6) la logica di gestione dell'evento, encapsulata nel metodo privato **myHandler**

In particolare, la gestione dell'evento deve:

- recuperare la carriera selezionata
- utilizzare tale dato per popolare i vari campi di uscita, tramite gli appositi metodi del controller

Cose da ricordare

- salva costantemente il tuo lavoro: l'informatica a volte può essere "subdolamente ostile" ..
- in particolare: se ora compila e stai per fare modifiche, salva la versione attuale (non si sa mai)

Checklist di consegna

- Hai fatto un **JAR eseguibile**, che contenga cioè l'indicazione del main?
- Hai controllato che **si compili e ci sia tutto?** [NB: non includere il PDF del testo]
- Hai **rinominato IL PROGETTO**, lo ZIP e il JAR esattamente come richiesto?
- Hai **chiamato** la cartella del progetto esattamente come richiesto?
- **Hai fatto un unico file ZIP (NON .7z, rar o altri formati)** contenente l'intero progetto?
In particolare, ti sei assicurato di aver incluso tutti i file .java (e non solo i .class)?
- **Hai consegnato DUE file distinti, ossia lo ZIP col progetto e il JAR eseguibile?**
- Su EOL, hai premuto il tasto "CONFERMA" per inviare il tuo elaborato?

ESAME DI FONDAMENTI DI INFORMATICA T-2 del 14/6/2022

Proff. E. Denti – R. Calegari – A. Molesini

Tempo a disposizione: 3h30

NOME PROGETTO ECLIPSE: CognomeNome-matricola (es. RossiMario-0000123456)

NOME CARTELLA PROGETTO: CognomeNome-matricola (es. RossiMario-0000123456)

NOME ZIP DA CONSEGNARE: CognomeNome-matricola.zip (es. RossiMario-0000123456.zip)

NOME JAR DA CONSEGNARE: CognomeNome-matricola.jar (es. RossiMario-0000123456.jar)

Si devono consegnare DUE FILE: *l'intero progetto Eclipse e il JAR eseguibile*

Si ricorda che compiti *non compilabili o palesemente lontani da 18/30* NON SARANNO CORRETTI e causeranno la verbalizzazione del giudizio “RESPINTO”

È stato richiesto di sviluppare un'applicazione che consenta ai clienti di una banca di tenere sott'occhio il saldo, giorno per giorno, di un conto corrente bancario.

DESCRIZIONE DEL DOMINIO DEL PROBLEMA

Un **conto corrente** è uno strumento bancario per effettuare e ricevere pagamenti, nonché per tenere depositato denaro presso la banca stessa. L'operare del cliente sul conto è descritto da un insieme ordinato di **movimenti**, che possono essere di diverse **tipologie**:

- **Accrediti**: rappresentano versamenti di denaro sul conto
- **Addebiti**: rappresentano prelievi di denaro sul conto o spese addebitate sul conto
- **Nulli**: rappresentano movimenti che coinvolgono (in addebito o accredito) un importo pari a zero
- **Saldo**: rappresentano l'ammontare di denaro esistente sul conto

Ogni movimento è caratterizzato da una serie di proprietà:

- **data contabile** in cui l'operazione viene registrata sul conto
- **data valuta** in cui i denari vengono effettivamente prelevati/versati sul conto
- **importo** del movimento stesso
- **causale** ossia una descrizione testuale del motivo del prelievo/versamento/operazione.

Logicamente, il saldo delle operazioni a una certa data D2 è pari al saldo in una precedente data D1 più tutti gli accrediti e meno tutti gli addebiti nel frattempo intervenuti.

Il file **Movimenti.txt** contiene l'elenco dei movimenti di un conto corrente, nel formato descritto più oltre.

TEMPO STIMATO PER SVOLGERE L'INTERO COMPITO:

2h15 – 3h

PARTE 1 – Modello dei dati: Punti 12

[TEMPO STIMATO: 60-75 minuti]

PARTE 2 – Persistenza: Punti 7

[TEMPO STIMATO: 30-45 minuti]

PARTE 3 – Grafica: Punti 11

[TEMPO STIMATO: 45-60 minuti]

JAVAFX – Parametri run configuration nei LAB

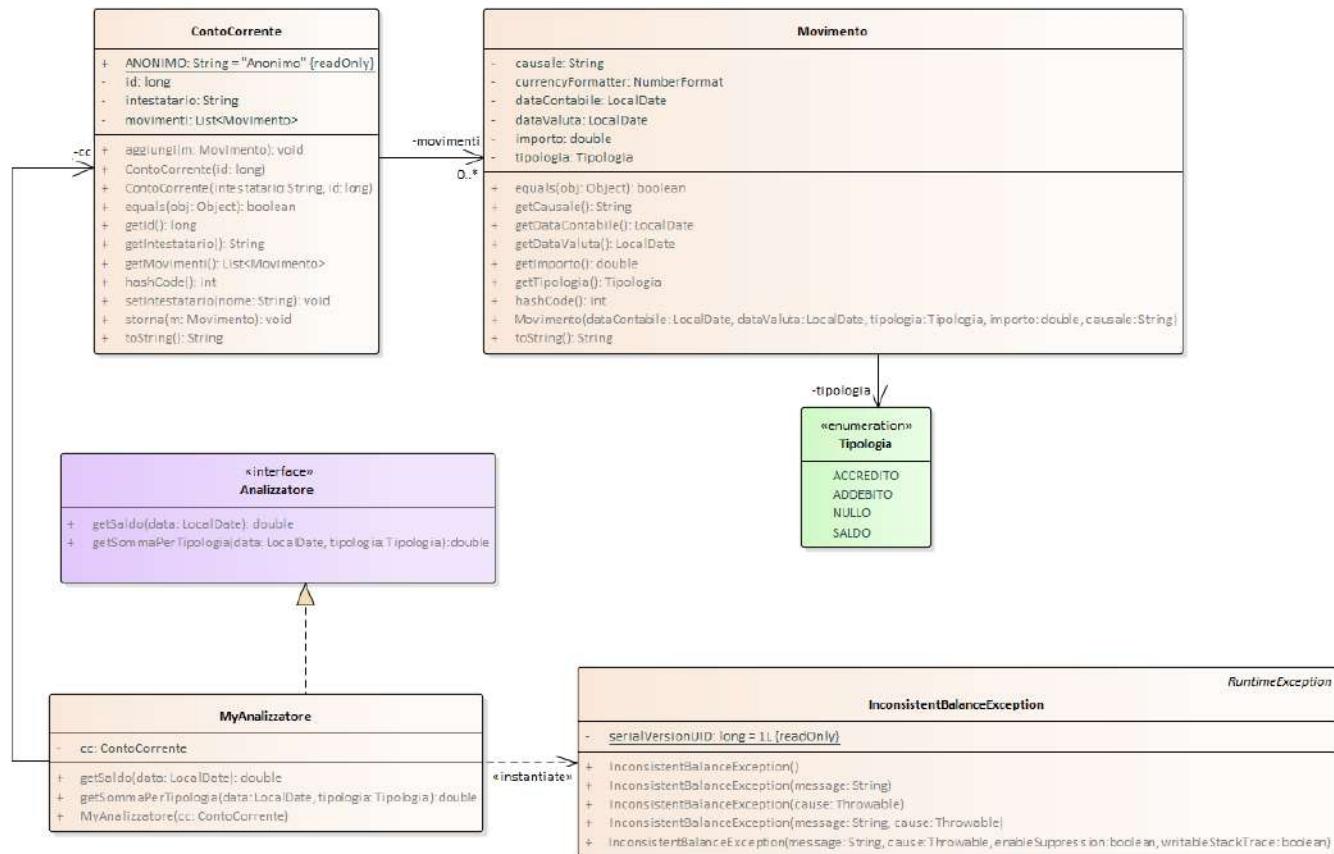
```
--module-path "C:\applicativi\moduli\javafx-sdk-17.0.2\lib"  
--add-modules javafx.controls
```

Parte 1 – Modello dei dati

(punti: 12)

Package: contocorrente.model

[TEMPO STIMATO: 60-75 minuti]



SEMANTICA:

- l'enumerativo **Tipologia** (fornito) elenca le quattro possibili tipologie di movimento
- la classe **Movimento (da completare nel costruttore)** rappresenta un movimento come descritto nel Dominio del Problema: appositi accessori consentono di estrarre le proprietà rilevanti, e idonee `equals`, `hashCode` e `toString` completano l'implementazione. *Il costruttore deve verificare con cura gli argomenti ricevuti*, verificando in particolare che non siano nulli (per il valore `double`, che non sia `NaN`) e che l'importo – positivo, negativo o nullo – sia coerente con la tipologia di movimento specificata. Le formattazioni degli importi devono utilizzare apposito formatter con convenzioni culturali italiane.
- la classe **ContoCorrente (da completare nel metodo aggiungi)** rappresenta il conto corrente, caratterizzato da *lista movimenti*, *intestatario* (una stringa) e *identificativo univoco* (un valore long). Anche in questo caso appositi accessori consentono di estrarre le proprietà rilevanti mentre idonee `equals`, `hashCode` e `toString` completano l'implementazione. Il conto corrente mantiene al suo interno la lista dei movimenti costantemente ordinata per data contabile crescente e in subordine per data valuta crescente: essa è manipolabile attraverso la coppia di metodi `aggiungi(Movimento)` e `storna(Movimento)` che rispettivamente inseriscono/rimuovono il movimento dato (se non nullo) dalla lista. In particolare, *il metodo aggiungi deve garantire che l'inserimento del movimento avvenga in modo da mantenere la lista movimenti costantemente ordinata come sopra specificato*.
- l'interfaccia **Analizzatore** (fornita) rappresenta il componente che analizza ed elabora i movimenti, esponendo a tal fine i due metodi `getSaldo(LocalDate)`, che restituisce il saldo del conto alla data contabile specificata, e `getSommaPerTipologia(LocalDate, Tipologia)` che restituisce la somma dei movimenti della tipologia data alla data contabile specificata.

- e) la classe **MyAnalizzatore** (da completare implementando i due metodi), il cui costruttore riceve il **ContoCorrente** su cui operare, implementa la precedente:
- il metodo **getSaldo** effettua la somma algebrica dei movimenti effettuati entro la data contabile specificata, *avendo cura però di escludere le righe relative ai saldi diversi dal saldo iniziale* (altrimenti, gli importi verrebbero sommati più volte e il totale sarebbe inconsistente); **in presenza di un saldo intermedio deve invece verificare che la somma dei movimenti fino a quel momento coincida con quella dichiarata nel saldo stesso (a meno di € 0,01)**: ove così non sia, deve lanciare l'apposita **InconsistentBalanceException** (fornita)
 - il metodo **getSommaPerTipologia** procede similmente, limitando tuttavia la somma ai soli movimenti della tipologia specificata, che può essere solo **ACCREDITO** o **ADDEBITO**: in caso contrario, dev'essere lanciata **IllegalArgumentException**

Parte 2 – Persistenza

(punti: 7)

Package: *contocorrente.persistence*

[TEMPO STIMATO: 30-45 minuti]

La prima riga del file di testo specifica il numero del conto corrente, secondo il seguente formato:

- la sigla “CC”, seguita da uno o più spazi
- il numero del conto nella forma “N.123455667”, dove il valore numerico è un *long* ma occorre tenere in considerazione che potrebbero essere presenti spazi dopo “N.”

Entrambe le sigle “N.” e “CC” sono case-insensitive.

Le righe successive descrivono ciascuna un movimento, tramite quattro elementi separati da tabulazioni:

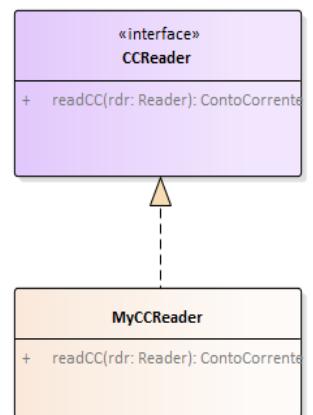
- la data contabile, nel formato italiano SHORT
- la data valuta, anch'essa nel formato italiano SHORT
- l'importo espresso come numero positivo o negativo, senza simbolo di valuta, ma con l'utilizzo dei separatori italiani per le migliaia e per la parte decimale; *tuttavia, in questa banca l'eventuale segno “-“ negli importi negativi potrebbe essere seguito da spazi prima del valore assoluto del numero*
- la causale, intesa come stringa libera che può contenere spazi

ESEMPIO:

CC N.123456789			
31/01/22	31/01/22	5.317,81	SALDO INIZIALE A VOSTRO CREDITO
02/02/22	02/02/22	- 2,90	IMP.BOLLO CC
07/02/22	07/02/22	- 61,59	SPESE GESTIONE
09/02/22	08/02/22	- 29,14	PAGAMENTO POS CARBURANTI BOLOGNA
...			
21/02/22	20/02/22	- 31,57	PAGAMENTO POS SUPERMARKET BOLOGNA
21/02/22	20/02/22	- 24,90	PAGAMENTO POS TECNOLOGIA FIRENZE
22/02/22	21/02/22	- 28,56	PAGAMENTO POS CARBURANTI BOLOGNA
25/02/22	25/02/22	2.144,58	RETRIBUZIONE
27/02/22	27/02/22	- 21,11	UTENZE
28/02/22	28/02/22	6.855,30	SALDO FINALE AL 28/02/2022
02/03/22	02/03/22	- 2,62	IMP.BOLLO CC
07/03/22	05/03/22	- 49,97	PAGAMENTO POS CARBURANTI MODENA
...			

SEMANTICA:

- L'eccezione **BadFormatException** (fornita) esprime l'idea di file formattato in modo scorretto
- L'interfaccia **CCReader** (fornita) dichiara il metodo **readCC**, che legge da un *Reader* (ricevuto come argomento) i dati di un conto corrente, configurando e restituendo l'opportuno oggetto **ContoCorrente**



- c) La classe **MyCCReader** (da realizzare) implementa **CCReader**: non prevede costruttori, si limita a implementare il metodo **readCC** come sopra specificato. In caso di problemi di I/O deve essere propagata l'opportuna **IOException**, mentre in caso di **Reader** nullo o altri problemi di formato dei file deve essere lanciata una opportuna **BadFormatException**, il cui messaggio dettagli l'accaduto.
- Per la prima riga, il reader deve verificare, lanciando **BadFormatException** in caso contrario, che:
- i. la riga contenga o due o tre elementi
 - ii. il primo elemento sia “CC” (case-insensitive)
 - iii. il secondo elemento inizi per “N.” (case-insensitive) e, nel caso non sia presente un terzo elemento, contenga il numero del conto espresso come intero long
 - iv. l'eventuale terzo elemento, se non già presente nel secondo, contenga il numero del conto come intero long
- d) Per le righe successive, il reader deve verificare, sempre lanciando **BadFormatException** in caso contrario, che:
- i. ogni riga contenga esattamente quattro elementi (tab-separated)
 - ii. le date siano espresse secondo le convenzioni italiane (formato SHORT)
 - iii. l'importo segua le regole sopra descritte

Sulla base dei dati letti deve essere costruito un nuovo oggetto **Movimento**, avendo cura di attribuirgli la corretta **Tipologia** in base all'importo (positivo, negativo, nullo) e alla causale (per distinguere il saldo iniziale, da trattare come accredito, dagli altri saldi).

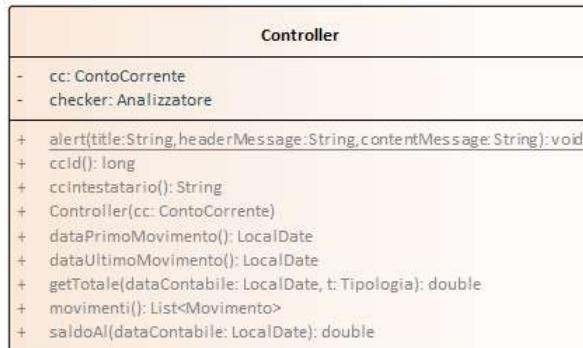
Parte 3

(punti: 11)

Package: contocorrente.controller

(punti 0)

Il **Controller** (fornito) è organizzato secondo il diagramma UML nella figura seguente: esso lavora su un **ContoCorrente** ricevuto all'atto della costruzione ed utilizza internamente un **Analizzatore** per operare su esso.



SEMANTICA:

- il costruttore riceve il **ContoCorrente** su cui opererà e istanzia l'opportuno **Analizzatore**;
- i metodi accessor, dall'ovvia denominazione, fanno da ponte verso gli omonimi metodi del model
- la classe contiene anche il **metodo statico ausiliario alert**, utile per mostrare avvisi all'utente.

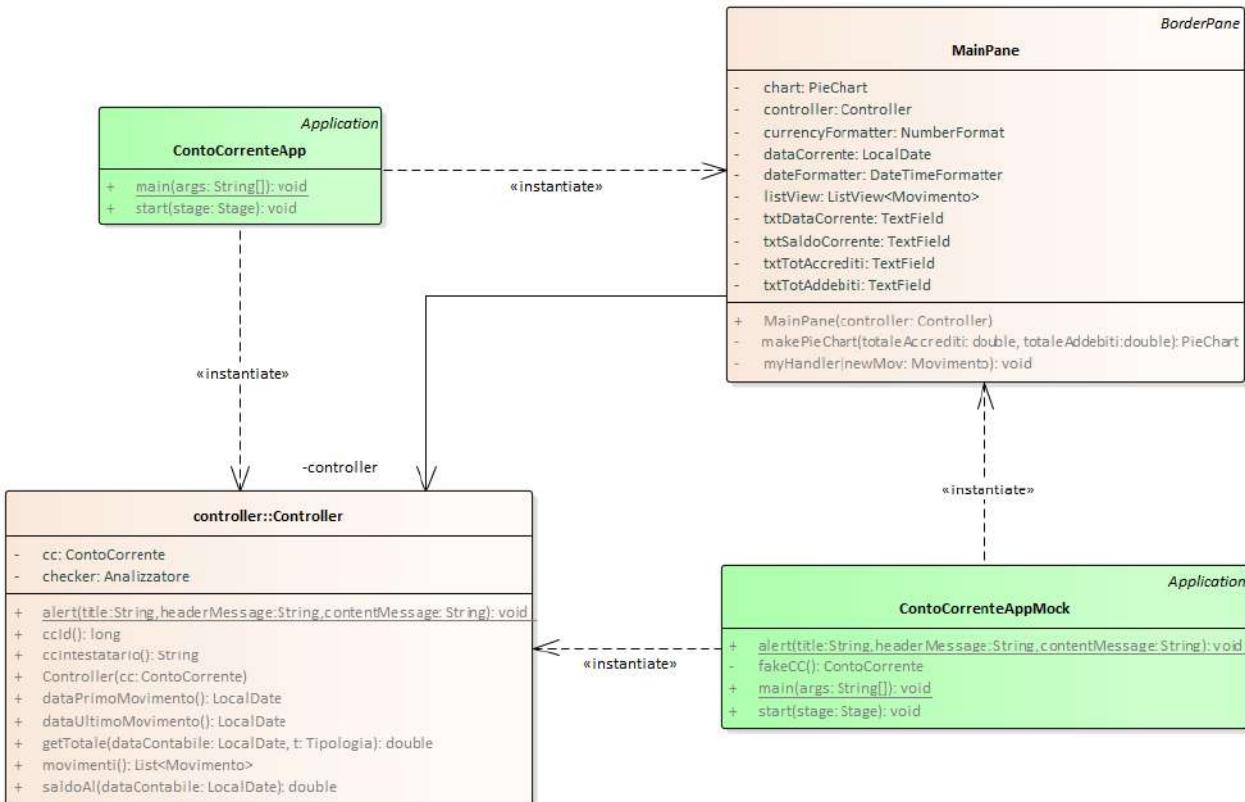
Package: contocorrente.ui

[TEMPO STIMATO: 45-60 minuti] (punti 11)

La classe **ContoCorrenteApp** (fornita) costituisce l'applicazione JavaFX che si occupa di aprire i file, creare il controller e incorporare il **MainPane**. Per consentire di collaudare la GUI anche in assenza / in caso di

malfunzionamento della parte di persistenza, è possibile avviare l'applicazione mediante la classe **ContoCorrenteAppMock**.

L'interfaccia utente è illustrata nelle figure seguenti e segue il modello sotto illustrato:



L'interfaccia grafica si presenta come segue:

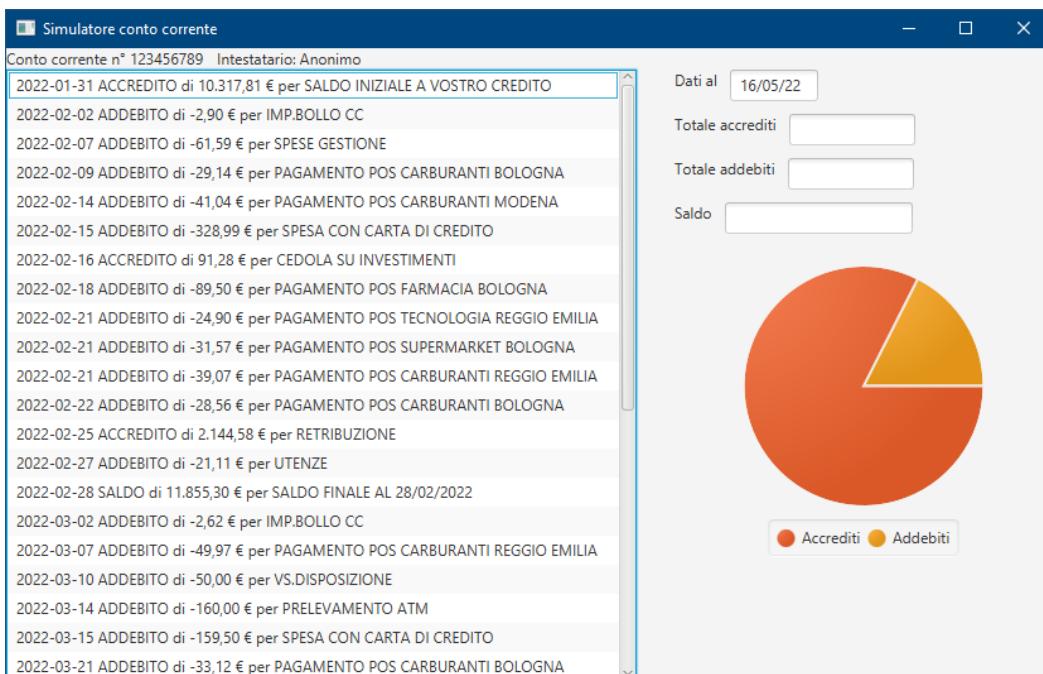
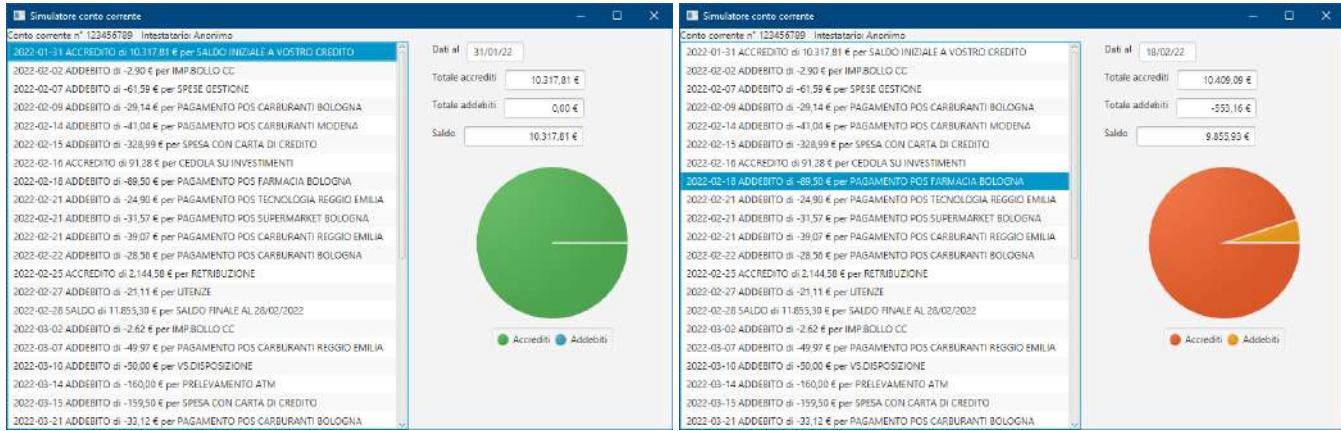


Fig. 1: la situazione iniziale della GUI, con data contabile impostata di default alla *data odierna*

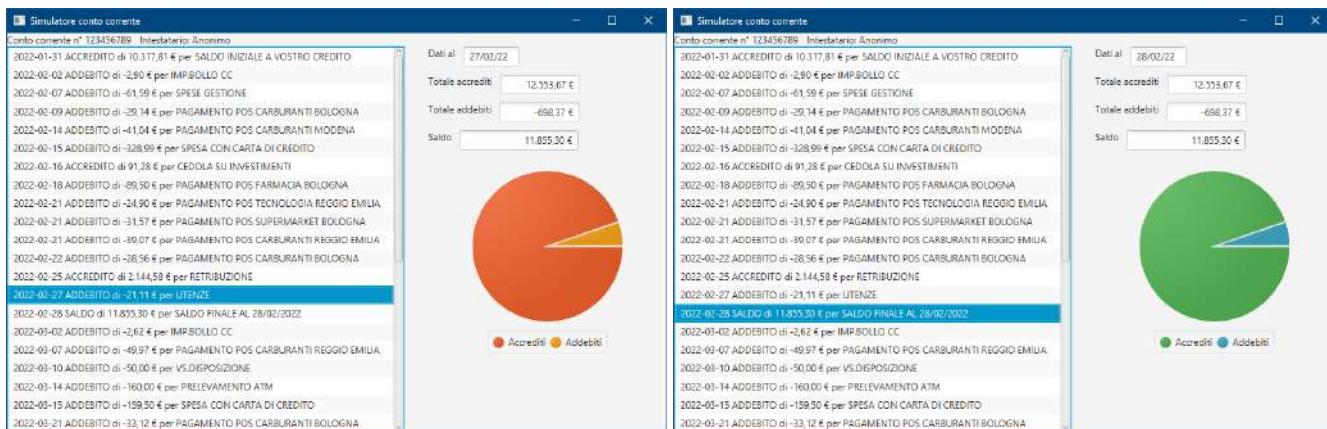
- in alto, una etichetta riporta numero e intestatario del conto corrente
- a sinistra, una **ListView** mostra i movimenti del conto corrente

- a destra, una serie di etichette e campi di testo (non editabili e posti verticalmente in una VBox) mostrano i dati del conto corrente a una certa data, formattata secondo le convenzioni italiane in stile SHORT. Sotto, un *grafico a torta* mostra il *rapporto fra accrediti e addebiti* alla data sopra indicata.

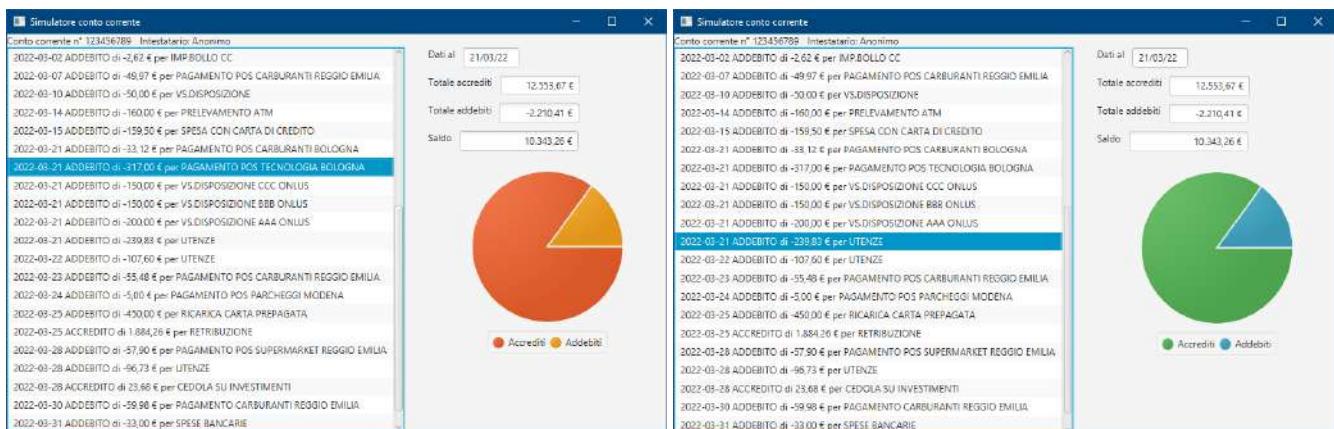
L'utente può interagire unicamente selezionando le varie righe sulla sinistra: immediatamente, la data sulla destra viene aggiornata alla data contabile del movimento selezionato e, subito a seguire, vengono di conseguenza aggiornati anche il saldo e i totali accrediti/addebiti, nonché il grafico a torta.



Figg. 2 / 3: la GUI dopo la selezione della prima riga (a sinistra) e di una riga successiva (a destra)



Figg. 4 / 5: il saldo e i totali non cambiano selezionando una riga di saldo intermedio, che viene però verificata



Figg. 6 / 7: nel caso di più movimenti nella stessa data contabile, i totali non variano, indipendentemente da quale riga si selezioni

Il MainPane è fornito parzialmente realizzato: è presente buona parte dell'impostazione strutturale, mentre sono da completare la configurazione di alcuni componenti e la gestione degli eventi.

La classe **MainPane** (da completare) estende **BorderPane** e prevede:

- 1) in alto, una **HBox** con dentro due etichette affiancate
- 2) a sinistra, una **ListView** di larghezza 500 e con righe impostate di adeguata altezza, per leggibilità
- 3) a destra, una **VBox** con le varie etichette e campi di testo (inserite a coppie in altrettante piccole **HBox** per garantire un gradevole allineamento) e, sotto, il grafico a torta.

La **parte da completare** riguarda:

- 1) la configurazione iniziale dei formattatori di valuta e data e della data corrente
- 2) l'aggancio alla **ListView** dell'opportuno listener encapsulato nel metodo ausiliario **myHandler**
- 3) la logica di gestione dell'evento, encapsulata nel metodo privato **myHandler**
- 4) la costruzione del grafico a torta, di dimensioni 250x250, demandata al metodo ausiliario **mkPieChart**

In particolare, la gestione dell'evento deve:

- recuperare dal movimento selezionato la data contabile da utilizzare, e impostarla nell'apposito campo di testo, opportunamente formattata
- utilizzare tale data per recuperare dal controller i vari dati (saldo, totale accrediti e addebiti) e mostrarli, opportunamente formattati, negli appositi campi di testo
- aggiornare il grafico a torta, ricalcolandolo coi nuovi totali accrediti e addebiti appena ottenuti

Cose da ricordare

- salva costantemente il tuo lavoro: l'informatica a volte può essere "subdolamente ostile" ..
- in particolare: se ora compila e stai per fare modifiche, salva la versione attuale (non si sa mai)

Checklist di consegna

- Hai fatto un **JAR eseguibile**, che contenga cioè l'indicazione del main?
- Hai controllato che **si compili e ci sia tutto**? [NB: non includere il PDF del testo]
- Hai **rinominato IL PROGETTO**, lo ZIP e il JAR esattamente come richiesto?
- Hai **chiamato** la cartella del progetto esattamente come richiesto?
- **Hai fatto un unico file ZIP (NON .7z, rar o altri formati) contenente l'intero progetto?**
In particolare, ti sei assicurato di aver incluso tutti i file .java (e non solo i .class)?
- **Hai consegnato DUE file distinti, ossia lo ZIP col progetto e il JAR eseguibile?**
- Su EOL, hai premuto il tasto "CONFERMA" per inviare il tuo elaborato?

ESAME DI FONDAMENTI DI INFORMATICA T-2 del 31/1/2022

Proff. E. Denti – R. Calegari – A. Molesini

Tempo a disposizione: 3 ore

NOME PROGETTO ECLIPSE: CognomeNome-matricola (es. RossiMario-0000123456)

NOME CARTELLA PROGETTO: CognomeNome-matricola (es. RossiMario-0000123456)

NOME ZIP DA CONSEGNARE: CognomeNome-matricola.zip (es. RossiMario-0000123456.zip)

NOME JAR DA CONSEGNARE: CognomeNome-matricola.jar (es. RossiMario-0000123456.jar)

Si devono consegnare DUE FILE: *I'intero progetto Eclipse e il JAR eseguibile*

Si ricorda che compiti non compilabili o palesemente lontani da 18/30 NON SARANNO CORRETTI e causeranno la verbalizzazione del giudizio “RESPINTO”

AVVERTENZA

A seguito dell’interazione con alcuni studenti, da cui sono emerse difficoltà – e conseguenti perdite di tempo – nel comprendere appieno quanto richiesto dal testo di un compito, in questo caso la spiegazione del Dominio del Problema è stata *volutamente arricchita da lunghi esempi* (in azzurro).

Ciò non significa che il compito sia “più lungo”, ma solo che la spiegazione è stata integrata da esempi più completi, così da ridurre al minimo il rischio di potenziali incomprensioni. Il testo vero e proprio rimane comunque quello in nero.

È stato richiesto di sviluppare un’applicazione per aiutare gli studenti a predisporre una richiesta di piano di studi, che verifichi il rispetto dei vincoli dell’ordinamento didattico universitario.

DESCRIZIONE DEL DOMINIO DEL PROBLEMA

Ogni studente universitario segue una carriera definita dal proprio *piano di studi*, che elenca le *attività formative* che deve seguire e di cui deve sostenere l’esame. A ogni studente è attribuito, all’atto dell’iscrizione, un piano di studi standard, definito dal *piano didattico* del corso di studio, che potrà successivamente essere personalizzato.

Ogni *attività formativa* è caratterizzata da *denominazione*, *settore scientifico-disciplinare* (una sigla standardizzata come MAT/05, ING-INF/05, FIS/01, etc. che indica la macro-area a cui quella materia appartiene) e dal *numero di crediti* (spesso indicati come *cfu* = crediti formativi universitari) che si acquisiscono superando il relativo esame.

Esempi di attività formative:

ANALISI MATEMATICA T-1, 9 cfu, MAT/05

FONDAMENTI DI INFORMATICA T-1, 12 cfu, ING-INF/05

FISICA GENERALE T, 9 cfu, FIS/01

ELETTRONICA T, 6 cfu, ING-INF/01

etc.

Al fine di assicurare il giusto equilibrio fra materie di base, materie caratterizzanti, materie affini, materie a scelta e altre attività, quando una attività formativa viene inserita in un piano di studi è necessario specificare anche *in quale tipologia* si intenda collocarla. Le tipologie possibili sono stabilite per legge, identificate dalle lettere A-F: nelle singole aree culturali vengono poi spesso suddivise in sotto-ambiti specifici. Ad esempio per Ing. Informatica le tipologie A, B, E sono ulteriormente suddivise rispettivamente in 2, 3, 2 sotto-ambiti, come segue:

- **Tipologia A:** materie di base (suddivisa per Ing. Informatica in due sotto-ambiti, “Matematica, informatica e statistica” e “Fisica e chimica”, che indicheremo per brevità con A1 e A2)
- **Tipologia B:** materie caratterizzanti (suddivisa in tre sotto-ambiti, rispettivamente “Ingegneria elettronica”, “Ingegneria informatica” e “Ingegneria delle telecomunicazioni”, che indicheremo per brevità con B1, B2, B3)
- **Tipologia C:** materie affini o integrative
- **Tipologia D:** materie a scelta libera dello studente
- **Tipologia E:** prova finale e lingue straniere (suddivisa nei due sotto-ambiti “prova finale” e “lingue straniere”, che indicheremo per brevità con E1 ed E2)
- **Tipologia F:** altre attività

Esempio: nel piano didattico standard di Ingegneria Informatica, ANALISI MATEMATICA T-1 e FONDAMENTI DI INFORMATICA T-1 sono inserite in tipologia A (più precisamente nel sotto-ambito A1), FISICA GENERALE T è inserita in tipologia A nel sotto-ambito A2, ELETTRONICA T e RETI LOGICHE T sono inserite in tipologia B (rispettivamente nel sotto-ambito B1 la prima e nel sotto-ambito B2 la seconda); e così via.

Affinché un piano di studi sia valido occorre rispettare una serie di **vincoli**, stabiliti dall'**ordinamento** (il documento che definisce il quadro generale del corso di studio, approvato dal Ministero). In particolare:

- ogni attività formativa può essere presente nel piano **una sola volta**
- il totale dei crediti delle attività formative previste deve essere **almeno pari a 180 cfu**, compresa la prova finale
- per ogni tipologia, il totale crediti delle attività formative previste in tale tipologia deve rientrare in un dato **intervallo min/max di crediti** stabilito dall'ordinamento
- le attività formative etichettate con una data tipologia devono **appartenere a uno dei settori scientifico-disciplinari ammessi** dall'ordinamento per tale tipologia.

Ad esempio, l'ordinamento di ingegneria informatica prevede i seguenti intervalli di crediti e settori ammessi:

- Tipologia A, sotto-ambito "Matematica, informatica e statistica" (A1): min 39, max 51 cfu
settori ammessi: INF/01, ING-INF/05, MAT/02, MAT/03, MAT/05, MAT/06, MAT/08
- Tipologia A, sotto-ambito "Fisica e chimica" (A2): min 9, max 18 cfu
settori ammessi: CHIM/07, FIS/01, FIS/03
- Tipologia B, sotto-ambito "Ingegneria elettronica" (B1): min 6, max 15 cfu
settori ammessi: ING-INF/01, ING-INF/02, ING-INF/07
- Tipologia B, sotto-ambito "Ingegneria informatica" (B2): min 48, max 66 cfu
settori ammessi: ING-INF/05, ING-INF/04
- Tipologia B, sotto-ambito "Ingegneria delle telecomunicazioni" (B3): min 9, max 15 cfu
settori ammessi: ING-INF/02, ING-INF/03
- Tipologia C: min 18, max 30 cfu
settori ammessi: ING-IND/31, ING-IND/35, IUS/14, MAT/07, MAT/09
- Tipologia D: min 12, max 18 cfu
settori ammessi: tutti
- Tipologia E, sotto-ambito "prova finale" (E1): min 3, max 9 cfu
settori ammessi: solo la prova finale
- Tipologia E, sotto-ambito "lingue straniere" (E2): min 6, max 6 cfu
settori ammessi: solo le idoneità di lingua straniera
- Tipologia F: min 6, max 6 cfu
settori ammessi: tutti

Ogni piano di studio, compreso il piano didattico standard, deve rispettare tali vincoli e costituisce quindi una **specifica implementazione** dell'ordinamento: esso stabilisce in modo preciso quante e quali attività formative svolgere, elencandole una ad una e assegnando a ciascuna l'opportuna tipologia.

Ad esempio, Il piano didattico standard di Ingegneria Informatica implementa l'ordinamento come segue:

- Tipologia A, sotto-ambito "Matematica informatica e statistica": 5 attività per complessivi 45 cfu
ANALISI MATEMATICA T-1 (9 cfu, MAT/05), FONDAMENTI DI INFORMATICA T-1 (12 cfu, ING-INF/05), GEOMETRIA E ALGEBRA T (6 cfu, MAT/03), ANALISI MATEMATICA T-2 (6 cfu, MAT/05), FONDAMENTI DI INFORMATICA T-2 (12 cfu, ING-INF/05)
- Tipologia A, sotto-ambito "Fisica e chimica": 1 attività per complessivi 9 cfu
FISICA GENERALE T (9 cfu, FIS/01)
- Tipologia B, sotto-ambito "Ingegneria elettronica": 1 attività per complessivi 6 cfu
ELETTRONICA T (6 cfu, ING-INF/01)
- Tipologia B, sotto-ambito "Ingegneria informatica": 8 attività per complessivi 66 cfu
RETI LOGICHE T (6 cfu, ING-INF/05), CALCOLATORI ELETTRONICI T (6 cfu, ING-INF/05), SISTEMI INFORMATIVI T (9 cfu, ING-INF/05), SISTEMI OPERATIVI T (9 cfu, ING-INF/05), CONTROLLI AUTOMATICI T (9 cfu, ING-INF/04), RETI DI CALCOLATORI T (9 cfu, ING-INF/05), TECNOLOGIE WEB T (9 cfu, ING-INF/05), INGEGNERIA DEL SOFTWARE T (9 cfu, ING-INF/05)
- Tipologia B, sotto-ambito "Ingegneria delle telecomunicazioni": 1 attività per complessivi 9 cfu
FONDAMENTI DI TELECOMUNICAZIONI T (9 cfu, ING-INF/03)

- Tipologia C: 3 attività per complessivi ECONOMIA E ORGANIZZAZIONE AZIENDALE T (6 cfu, ING-IND/35), ELETTRONICA T (6 cfu, ING-IND/31), MATEMATICA APPLICATA T (6 cfu, MAT/07)	18 cfu
- Tipologia D: 1 o più attività a scelta libera, per complessivi	12 cfu
- Tipologia E, sotto-ambito “prova finale”: PROVA FINALE (3 cfu)	3 cfu
- Tipologia E, sotto-ambito “lingue straniere”: LINGUA INGLESE B-2 (6 cfu)	6 cfu
- Tipologia F: 1 attività a scelta fra la due seguenti, in alternativa fra loro, per complessivi TIROCINIO T (6 cfu) – oppure – LAB. DI AMMINISTRAZIONE DI SISTEMI (6 cfu, ING-INF/05)	6 cfu

Si può facilmente constatare che tutti i vincoli risultano rispettati, in quanto:

- a) ogni attività formativa appartiene a un settore permesso per la tipologia in cui è collocata
- b) il totale crediti di ogni tipologia rientra nell’intervallo min/max permesso dall’ordinamento
- c) nessuna attività compare due volte
- d) il totale crediti è esattamente 180

Infatti, ad esempio, relativamente al punto a):

- GEOMETRIA E ALGEBRA T (6 cfu, **MAT/03**) è inserita nella tipologia A1, che secondo l’ordinamento ammette i settori INF/01, ING-INF/05, MAT/02, **MAT/03**, MAT/05, MAT/06, MAT/08: dunque l’inserimento è valido (non avrebbe potuto essere collocata in altre tipologie, dato che nessun’altra di esse ammette il settore MAT/03, tranne la D (scelta libera) e la F (altre attività), che li ammettono tutti.)
- FONDAMENTI DI INFORMATICA T-1 (12 cfu, ING-INF/05) è anch’essa inserita nella tipologia A1, che secondo l’ordinamento prevede fra i settori ammessi appunto ING-INF/05; tale settore è presente però anche nella tipologia B2 (attività caratterizzanti, sotto-ambito “ingegneria informatica”) quindi questa attività avrebbe potuto anche essere collocata in tale tipologia. (La scelta in questi casi è del Consiglio che approva il regolamento: per un corso di fondamenti, la tipologia A è considerata in generale più opportuna)
- RETI LOGICHE T-1 (6 cfu, ING-INF/05) è inserita invece in tipologia B2, che da ordinamento ammette i due settori ING-INF/05 e ING-INF/04; avrebbe potuto anche essere inserita in tipologia A1, per lo stesso discorso di cui sopra.
- etc.

Relativamente al punto b):

- Se si sommano i crediti delle cinque attività etichettate come A1 si ottengono 45 cfu, un valore compreso nell’intervallo 39-51 stabilito dall’ordinamento per la tipologia A1
- i crediti dell’unica attività etichettata come A2 (FISICA GENERALE T) sono 9 cfu, un valore compreso nell’intervallo 9-18 stabilito dall’ordinamento per la tipologia A2
- etc.

La legge consente tuttavia allo studente di chiedere al Consiglio del corso di studi di **personalizzare il proprio piano di studi** rispetto alla carriera standard, proponendo sostituzioni di attività normalmente previste con altre.

Nel farlo è però indispensabile rispettare sempre e comunque i vincoli dell’ordinamento: scopo dell’applicazione richiesta è far sì che, mentre lo studente compila la proposta per il proprio piano di studi, sia possibile verificare il rispetto dei vincoli sopra esplicitati, in modo da evitare che venga proposto un piano di studi illegale.

L’elenco delle attività formative disponibili, con i relativi dati, è fornito nel file **AttivitàFormative.txt**.

Ai fini di questo compito, per semplicità, l’ordinamento e il piano didattico standard del corso di studio *Ingegneria Informatica* sono invece prodotti in modo cablato da appositi metodi nelle classi già fornite (v. oltre): non è quindi necessario leggerli da file.

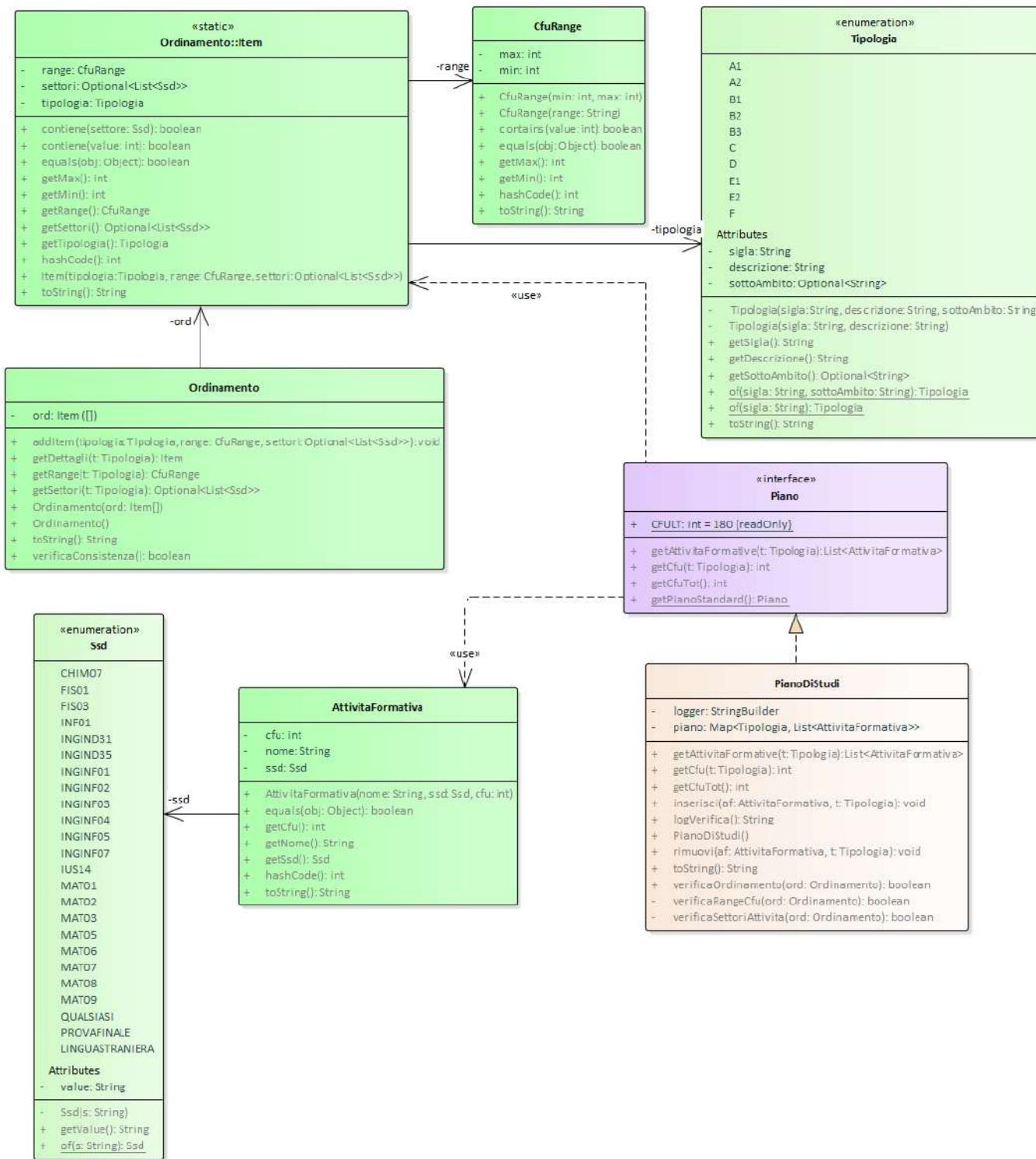
TEMPO STIMATO PER SVOLGERE L’INTERO COMPITO: 1h50 – 2h20

Parte 1

(punti: 21)

Modello dei dati (package pianodistudi.model)

[TEMPO STIMATO: 55-70 minuti] (punti 13)



SEMANTICA:

- l'enumerativo **Tipologia** (fornito) elenca le 10 possibili tipologie (da A ad F con i relativi sotto-ambiti): per comodità, per le tipologie che ammettono più sotto-ambiti sono stati definiti più valori, etichettati con la stessa lettera seguita da un numero crescente (A1, A2; B1, B2, B3; etc.); a ogni enumerativo è associata la corrispondente descrizione testuale per esteso. Tali elementi sono recuperabili tramite appositi metodi.
 Sono inoltre presenti due metodi factory *of* che, data la sigla corrispondente (es. "A") ed eventualmente il sotto-ambito quando previsto (es. "Fisica e chimica") restituiscono il corrispondente valore dell'enumerativo (es. A2), in modo da velocizzare le conversioni stringa/enumerativo. → vedere i test per esempi

- b) l'enumerativo **Ssd** (fornito) elenca i possibili settori scientifico-disciplinari: oltre alle sigle standard di legge sono state aggiunte, per comodità, alcune sigle “finte” utili per le tipologie D,E,F, ovvero QUAISIASI, PROVAFINAL e LINGUASTRANIERA.

Anche in questo caso è presente un metodo `factory of` che, data la sigla del settore, con qualunque mix di maiuscole/minuscole (es. “Ing-Inf/05” o “ING-INF/05” o altri mix) restituisce il corrispondente valore dell'enumerativo (es. INGINFO5), che per ovvie ragioni di sintassi Java non può contenere trattini o barrette: così le conversioni stringa/enumerativo risultano velocizzate. Dualmente, il metodo `getValue` restituisce il valore (stringa) corrispondente. → vedere i test per esempi

- c) la classe **AttivitàFormativa** (fornita) rappresenta l'attività formativa con le proprietà discusse nel *Dominio del Problema*: sono presenti i classici metodi per recuperare gli elementi, produrre un'opportuna stringa descrittiva, nonché `equals` e `hashCode`.
- d) la classe **CfuRange** (fornita) rappresenta un intervallo di crediti min-max, estremi inclusi: i costruttori (uno a due argomenti interi, l'altro da stringa della forma “numero-numero”), lanciano eccezione in caso di argomenti negativi o assurdi (min>max). Sono presenti i classici metodi accessori, `toString`, `equals` e `hashCode`. Il metodo di utilità `contains` verifica se l'intero fornito come argomento è compreso all'interno dell'intervallo min-max rappresentato. → vedere i test per esempi
- e) la classe **Ordinamento** (fornita) rappresenta un ordinamento. Il costruttore produce un ordinamento inizialmente vuoto, a cui vanno via via aggiunti gli elementi desiderati tramite il metodo `addItem`, che ha per argomenti la terna <tipologia, range di cfu, lista settori ammessi>. Inoltre, due metodi consentono di recuperare: → vedere i test per esempi
- `getRange`: il range di crediti permesso per una data **Tipologia**
 - `getSettori`: la lista dei settori (**Ssd**) permessi per una data **Tipologia** (NB: tale lista è opzionale perché nelle tipologie come D ed F l'ordinamento non specifica settori, in quanto vanno bene tutti; nel nostro modello esiste comunque anche la costante QUAISIASI che può essere usata per rappresentare questa situazione)

NB: questa classe è predisposta per supportare, in futuro, un metodo `verificaConsistenza` destinato a verificare che l'ordinamento rispetti altri vincoli di legge. Ai fini di questo compito, dato che l'unico ordinamento utilizzabile è cablato, tale verifica non è necessaria e il metodo non è implementato.

- f) l'interfaccia **Piano** (fornita) astrae il concetto di piano di studi catturandone le proprietà essenziali tramite i tre metodi `getCfu`, `getCfuTot`, `getAttivitàFormative`; espone altresì la costante pubblica CFULT = 180. Il metodo statico `factory getPianoStandard` restituisce una particolare istanza di **Piano** che rappresenta il piano didattico standard del corso di studio in Ingegneria Informatica.
- g) la classe **PianoDiStudi** (**da completare**) implementa **Piano** nel caso generale: a tal fine mantiene internamente l'elenco delle attività formative in esso presenti, divise per tipologia, in una comoda mappa <Tipologia, List<AttivitàFormative>>, così che sia rapido recuperare le attività formative di una data tipologia. Inoltre:
- il costruttore senza argomenti produce inizialmente un piano vuoto;
 - la coppia di metodi `inserisci/rimuovi` (**da implementare**, punti 4) consente di aggiungere/togliere un'attività formativa in una data tipologia; in particolare, `inserisci` deve controllare che l'attività formativa da inserire non sia già presente nel piano (né in quella tipologia, né in altre), mentre `rimuovi` deve controllare che l'attività formativa da togliere sia già presente nel piano in quella specifica tipologia – altrimenti, si deve lanciare **IllegalArgumentException**

Il metodo `verificaOrdinamento` (**da implementare**, punti 9) viene chiamato quando si desidera verificare che il piano rispetti i vincoli di un certo ordinamento fornito come argomento: più precisamente, tale metodo deve verificare: (→ vedere anche gli esempi nei test)

- che il totale crediti previsti dal piano per ciascuna tipologia rientri nel range min-max previsto dall'ordinamento per tale tipologia (ad esempio, nel caso di Ing. Informatica, che il totale dei crediti delle attività formative di tipologia A1 sia compreso fra 39 e 51; e così via per le altre)
- che le attività formative previste dal piano per ciascuna tipologia appartengano tutte a settori ammessi dall'ordinamento per tale tipologia (ad esempio, nel caso di Ing. Informatica, che le attività formative elencate nella tipologia A1 siano di settori permessi per tale tipologia, come MAT/05, MAT/03, ING-INF/05, etc.)

Il metodo `verificaOrdinamento` registra in un apposito `StringBuilder` interno, svuotato a ogni chiamata, i dettagli sui controlli che effettua, così che sia possibile ricostruire a posteriori le verifiche fatte e avere una spiegazione dell'esito della verifica. Il contenuto di tale `StringBuilder` è recuperabile tramite il metodo `logVerifica`.

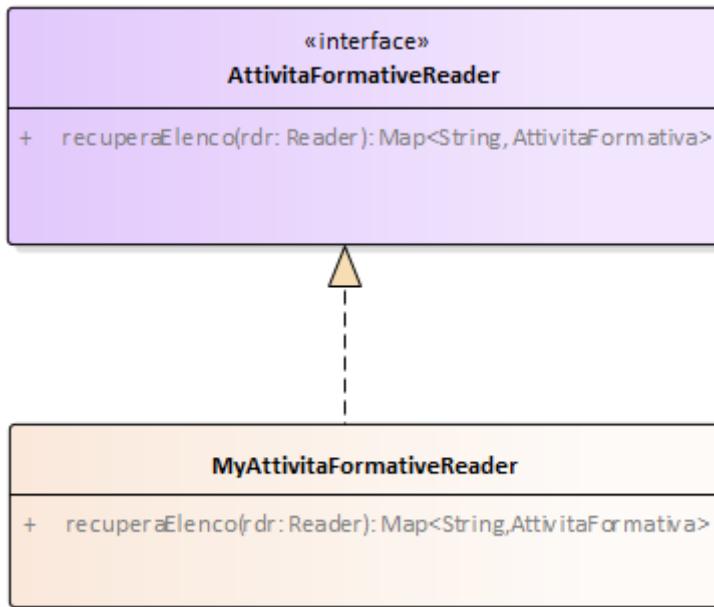
Persistenza (`pianodistudi.persistence`)

[TEMPO STIMATO: 25-30 minuti] (punti 8)

Il file di testo specifica una attività formativa per riga. Ogni riga contiene nell’ordine, separati da tabulazioni:

- il codice numerico univoco dell’attività formativa
- il nome dell’attività (può contenere spazi, apostrofi e ogni altro carattere utile)
- il settore scientifico-disciplinare (una sigla standardizzata), o la specifica SENZASETTORE
- il numero *intero* di crediti corrispondenti

27991	ANALISI MATEMATICA T-1	MAT/05	9
28004	FONDAMENTI DI INFORMATICA T-1	ING-INF/05	12
29228	GEOMETRIA E ALGEBRA T	MAT/03	6
26337	LINGUA INGLESE B-2	senzasettore	6
27993	ANALISI MATEMATICA T-2	MAT/05	6
...			



SEMANTICA:

- a) L’eccezione `BadFormatException` (fornita) esprime l’idea di file formattato in modo scorretto
- b) L’interfaccia `AttivitaFormativaReader` (fornita) dichiara il metodo `recuperaElenco`, che legge da un `Reader` (ricevuto come argomento) i dati di tutte le attività formative elencate nell’omonimo file, restituendole sotto forma di mappa (*nome, attività formativa*), tecnicamente una `Map<String, AttivitaFormativa>`
- c) La classe `MyAttivitaFormativaReader` (**da realizzare**) implementa `AttivitaFormativaReader`: non prevede costruttori, si limita a implementare il metodo `recuperaElenco` come sopra specificato. In caso di problemi di I/O deve essere propagata l’opportuna `IOException`, mentre in caso di Reader nullo o altri problemi di

formato dei file deve essere lanciata una opportuna ***BadFormatException***, il cui messaggio dettaglia l'accaduto.

In particolare, il reader deve verificare, lanciando ***BadFormatException*** in caso contrario, che

- i. ogni riga contenga il giusto numero di elementi;
- ii. il settore scientifico-disciplinare sia *valido*, ossia sia accettato dal metodo factory di ***Ssd***;
- iii. il numero di crediti sia intero.

Si segnala che la verifica di cui al punto ii) viene concretamente già effettuata dal metodo ***of*** di ***Ssd***, mentre che il numero di crediti sia ≥ 1 è già verificato dal costruttore di ***AttivitaFormativa***.

Parte 2

(punti: 9)

Controller (*pianodistudi.controller*)

(punti 0)

Il Controller (fornito) è organizzato secondo il diagramma UML nella figura seguente: esso lavora su un ***Ordinamento*** e un elenco delle attività formative disponibili (ricevuti come argomenti all'atto della costruzione) e incorpora un ***PianoDiStudi***, che gestisce secondo l'input proveniente dall'interfaccia grafica.



SEMANTICA:

- il costruttore riceve un ***Ordinamento*** e una mappa `<String, AttivitaFormativa>` analoga a quella restituita dal metodo ***recuperaElenco*** di ***AttivitaFormativeReader***;
- sono disponibili accessori come ***getOrdinamento***, ***getListAF*** (che restituisce la lista ordinata alfabeticamente di tutte le attività formative) e ***getAttivitaFormativaPerNome*** (che cerca e restituisce una ***AttivitaFormativa*** dato il suo nome), ***getAttivitaFormativePerTipologia*** (che restituisce la lista delle ***AttivitaFormative*** previste dall'ordinamento per una data tipologia)
- la coppia di metodi ***inserisci/rimuovi*** consente di aggiungere/togliere al piano di studi interno al controller un'attività formativa nella/dalla tipologia specificata; analogamente la coppia di metodi ***getCfu/getCfuTot*** consente di recuperare tali informazioni dal piano di studi interno al controller
- la coppia di metodi ***rispettaOrdinamento/logVerifica*** rispettivamente effettua sul piano di studi interno al controller la verifica dell'ordinamento, e ne recupera il log (usando gli analoghi metodi di ***PianoDiStudi***)
- il metodo ausiliario ***getPianoDidatticoStandard*** restituisce il piano didattico standard di Ingegneria Informatica, nella forma di mappa che associa a ogni tipologia l'elenco delle attività formative previste

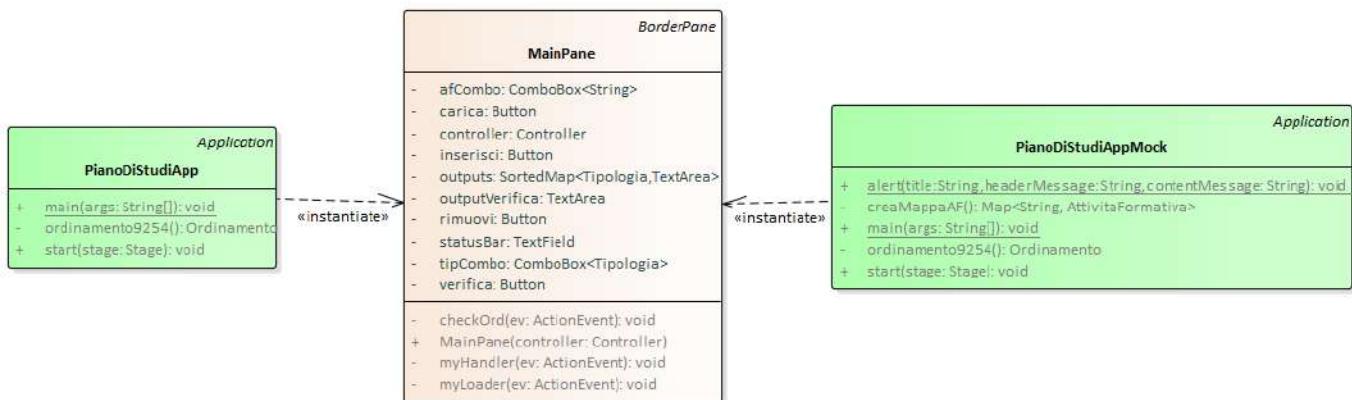
(per semplicità, per i corsi a scelta di tipo D ne sceglie due di default, mentre per la tipologia F inserisce sempre il tirocinio: l'utente potrà poi modificare queste scelte nella GUI)

- La classe contiene anche il **metodo statico ausiliario** `alert`, utile per mostrare avvisi all'utente.

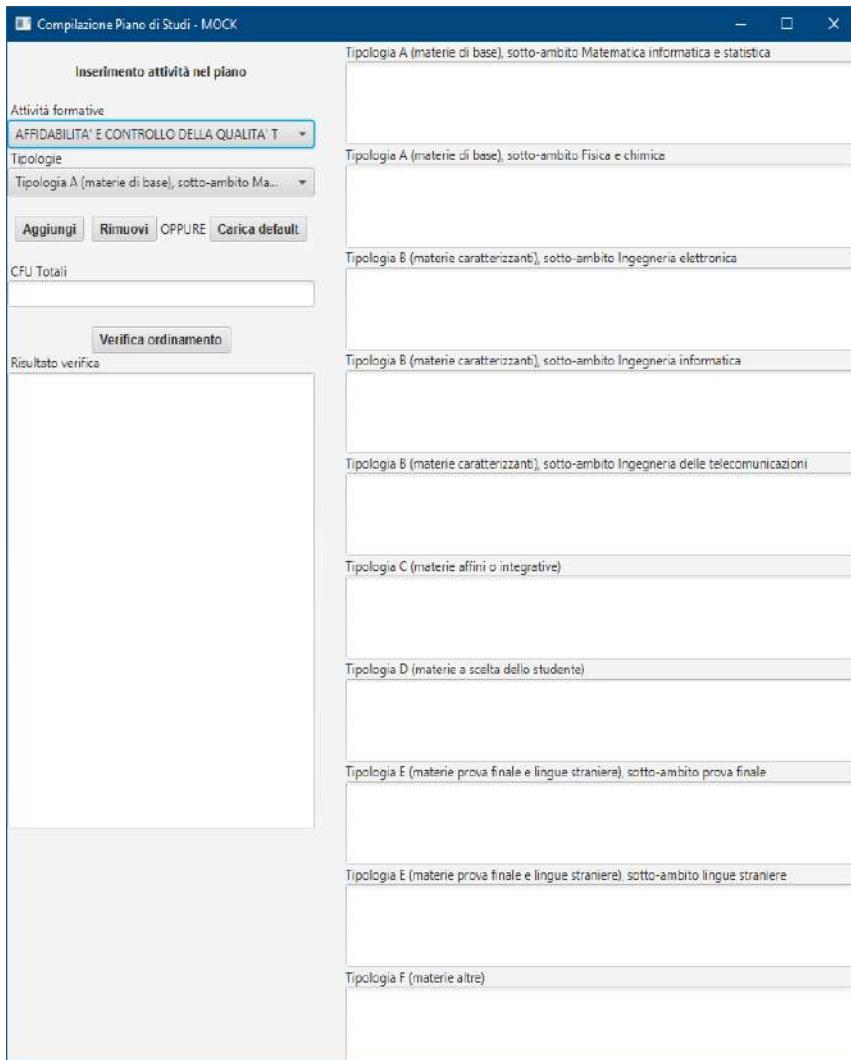
Interfaccia utente (pianodistudi.ui)

[TEMPO STIMATO: 30-40 minuti] (punti 9)

L'interfaccia utente è illustrata nelle figure seguenti e segue il modello sotto illustrato:



La classe **PianoDiStudiApp** (fornita) costituisce l'applicazione JavaFX che si occupa di aprire i file, creare il controller e incorporare il **MainPane**. Per consentire di collaudare la GUI anche in assenza / in caso di malfunzionamento della parte di persistenza, è possibile avviare l'applicazione mediante la classe **PianoDiStudiAppMock**.



Concretamente, l'interfaccia grafica si presenta come a lato:

- sulla destra, 10 aree di testo mostrano le attività formative attualmente presenti nel piano di studi, suddivise per tipologia;

- sulla sinistra sono presenti i pulsanti e gli altri elementi di controllo, nonché un'area di testo destinata a mostrare il risultato della verifica di ordinamento.

In alto, una prima combo elenca tutte le attività formative disponibili, in ordine alfabetico; subito sotto, un'altra combo elenca le 10 tipologie coi vari sotto-ambiti.

Due pulsanti consentono di inserire o togliere l'attività selezionata dalla tipologia selezionata, mentre il terzo carica il piano di studi di default per velocizzare la compilazione (Fig. 1).

A seguire, un campo di testo (non editabile) mostra il totale del cfu delle attività attualmente inserite nel piano.

Il pulsante *Verifica ordinamento* serve a scatenare la verifica che il piano attualmente mostrato rispetti l'ordinamento: il risultato della verifica è mostrato nell'area sottostante.

Il MainPane è fornito parzialmente realizzato: è presente la parte strutturale, mentre manca la parte di popolamento combo e gestione degli eventi.

La classe **MainPane (da completare)** estende **BorderPane** e prevede:

- 1) a sinistra in alto, due **ComboBox** da popolare rispettivamente con *l'elenco alfabetico ordinato delle attività formative* (la prima) e con *l'elenco ordinato delle tipologie* (la seconda): per entrambe deve essere preselezionato il primo elemento (Fig. 1)
- 2) sempre sulla sinistra, i vari *pulsanti* come sopra descritto, più alcune *Label*, seguiti da un **TextField** (non editabile) in cui viene mostrato il totale del cfu delle attività attualmente inserite nel piano, e infine la **TextArea** (non editabile) destinata a mostrare il risultato della verifica di ordinamento.
- 3) sulla destra, dieci **TextArea** mostrano le attività formative attualmente inserite nel piano.

La **parte da completare** riguarda:

- 1) il popolamento delle due **combo**, con predisposizione dell'elemento preselezionato;
- 2) il metodo **myHandler**, da chiamare in risposta all'evento di pressione dei due pulsanti **Inserisci/Rimuovi**, che deve inserire o rimuovere l'attività formativa attualmente selezionata dalla tipologia attualmente selezionata (Fig. 2); se ciò non risulta possibile (perché il controller emette eccezione), dev'essere emesso apposito messaggio di errore tramite **alert** (Figg. 3 e 4).
- 3) il metodo **checkOrd**, da chiamare in risposta all'evento di pressione del pulsante **Verifica ordinamento**: l'esito della verifica dev'essere mostrato nell'area all'uopo prevista (Fig. 5).

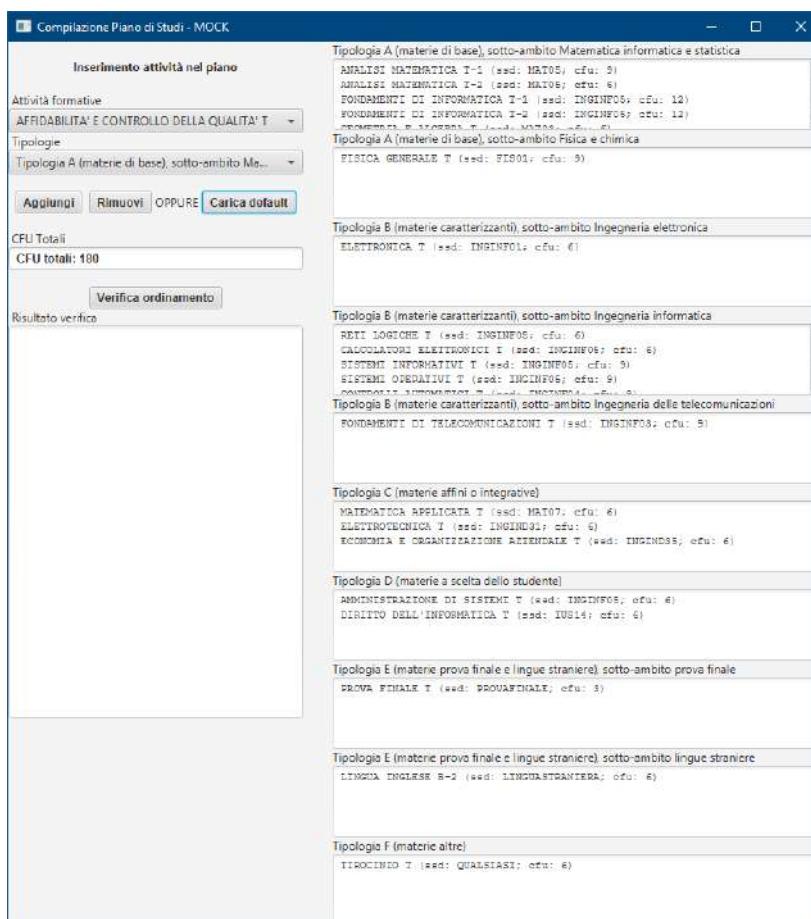


Fig. 1: la GUI dopo la pressione del pulsante *Carica default*

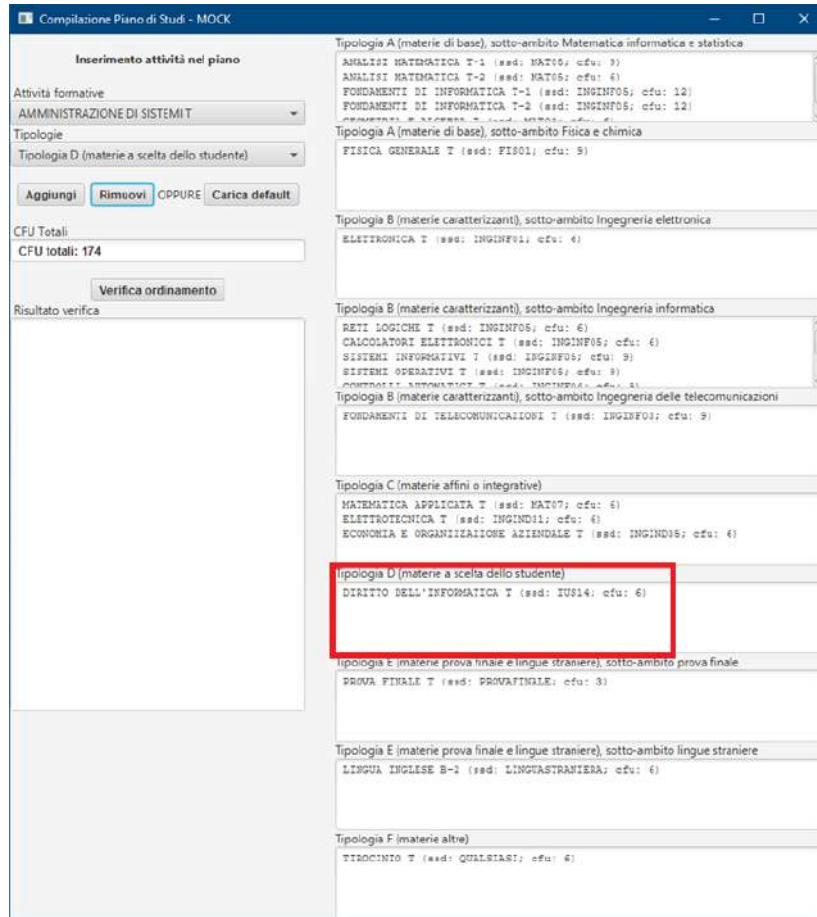


Fig. 2: la GUI dopo la pressione del pulsante *Rimuovi* per l'attività “Amministrazione di sistemi T” dalla tipologia D

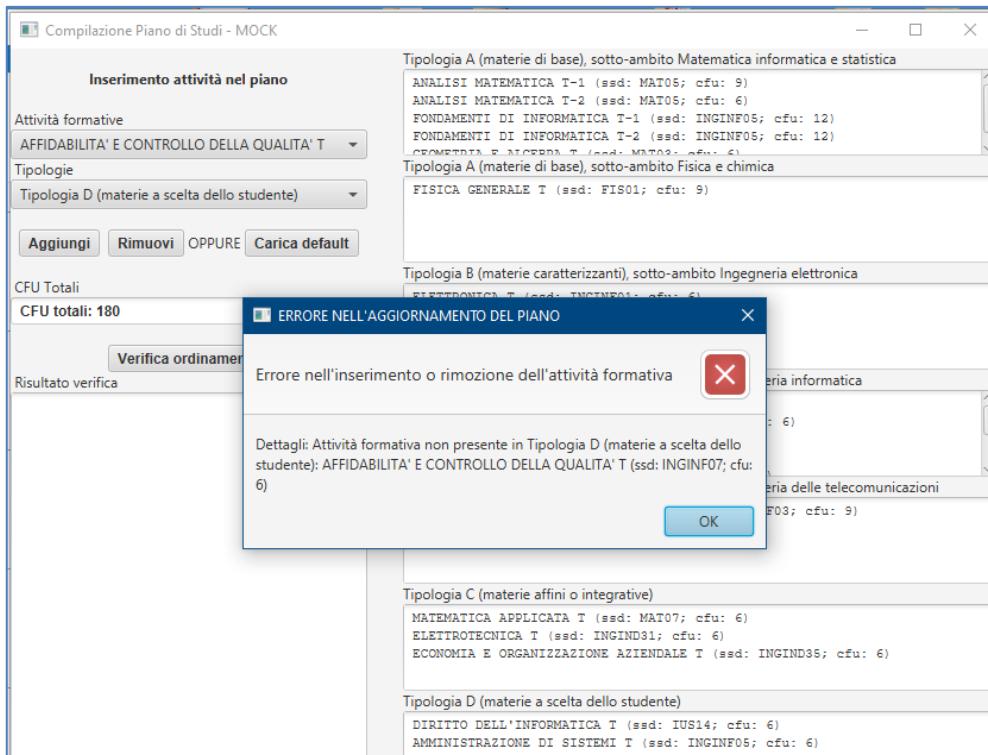


Fig. 3: messaggio d'errore tentando di rimuovere una attività da una tipologia in cui non era presente

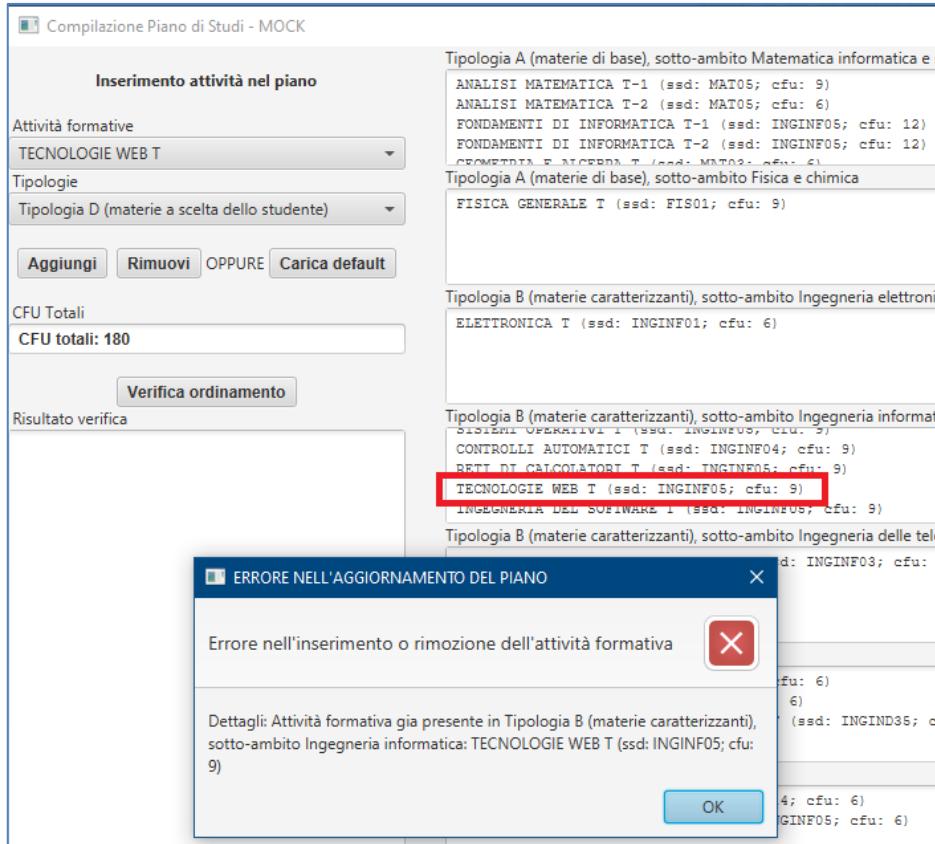


Fig. 4: messaggio d'errore tentando di aggiungere una attività già presente

Fig. 5: alcuni possibili esiti di verifiche di ordinamento

Cose da ricordare

- salva costantemente il tuo lavoro: l'informatica a volte può essere "subdolamente ostile" ..
- in particolare: se ora compila e stai per fare modifiche, salva la versione attuale (non si sa mai)

Checklist di consegna

- Hai fatto un **JAR eseguibile**, che contenga cioè l'indicazione del main?
- Hai controllato che **si compil e ci sia tutto?** [NB: non includere il PDF del testo]
- Hai **rinominato IL PROGETTO**, lo ZIP e il JAR esattamente come richiesto?
- Hai **chiamato** la cartella del progetto esattamente come richiesto?
- **Hai fatto un unico file ZIP (NON .7z, rar o altri formati) contenente l'intero progetto?**
In particolare, ti sei assicurato di aver incluso tutti i file .java (e non solo i .class)?
- **Hai consegnato DUE file distinti, ossia lo ZIP col progetto e il JAR eseguibile?**
- Su EOL, hai **premuto** il tasto "CONFERMA" per inviare il tuo elaborato?

ESAME DI FONDAMENTI DI INFORMATICA T-2 del 14/1/2022

Proff. E. Denti – R. Calegari – A. Molesini

Tempo a disposizione: 3 ore

NOME PROGETTO ECLIPSE: CognomeNome-matricola (es. RossiMario-0000123456)

NOME CARTELLA PROGETTO: CognomeNome-matricola (es. RossiMario-0000123456)

NOME ZIP DA CONSEGNARE: CognomeNome-matricola.zip (es. RossiMario-0000123456.zip)

NOME JAR DA CONSEGNARE: CognomeNome-matricola.jar (es. RossiMario-0000123456.jar)

Si devono consegnare DUE FILE: *l'intero progetto Eclipse e il JAR eseguibile*

Si ricorda che compiti *non compilabili o palesemente lontani da 18/30* NON SARANNO CORRETTI e causeranno la verbalizzazione del giudizio “RESPINTO”

Al fine di tarare al meglio una possibile riforma, il Ministro delle Finanze dell’Elbonia ha richiesto un’applicazione per confrontare l’effetto della modifica delle aliquote per le imposte dirette delle persone fisiche.

DESCRIZIONE DEL DOMINIO DEL PROBLEMA

In Elbonia la tassazione delle persone fisiche è *progressiva*, ossia si applicano aliquote percentuali crescenti al crescere del reddito del contribuente. Fino ad ora, sono state previste cinque fasce di reddito con le seguenti aliquote:

- **fino a 15.000 euro:** aliquota del 23%
- **oltre 15.000 euro e fino a 28.000 euro:** aliquota del 27%
- **oltre 28.000 euro e fino a 55.000 euro:** aliquota del 38%
- **oltre 55.000 euro e fino a 75.000 euro:** aliquota del 41%
- **oltre 75.000 euro:** aliquota del 43%

Per attenuare l’impatto sulle categorie più deboli è inoltre prevista una ***no-tax area***, ossia una quota di reddito comunque non soggetta a tassazione, di **8.174 euro**; per questo motivo il ***reddito imponibile*** (effettivamente soggetto a tassazione) è minore di quello ***lordo*** complessivo.

Su questi valori si applica poi il sistema progressivo sopra descritto, partendo dalla fascia più alta applicabile e passando via via alla precedente per la quota-parte di reddito che rimane da tassare.

ESEMPIO: contribuente con un reddito di 44.000 euro.

- 1° passo: scorporo della no-tax area → reddito imponibile effettivo = € 44.000 - 8.174 = € 35.826
- 2° passo: applicazione aliquote progressive partendo dalla fascia più alta applicabile e passando via via alla precedente per la quota parte di reddito che rimane da tassare

- il reddito imponibile, di € 35.826, ricade nella fascia di reddito fra € 28.000 ed € 55.000 (fascia più alta applicabile) → la quota parte da tassare in questa fascia è la parte eccedente € 28.000, ossia € 7.826
→ imposta = 38% di € 7.826 = € 2.973,88
- la restante parte di imponibile, € 28.000, ricade ora nella fascia precedente del 27%: si applica quindi lo stesso ragionamento, tassando al 27% la quota parte di reddito compresa fra € 15.000 ed € 28.000 (ossia € 13.000)
→ imposta = 27% di € 13.000 = € 3.510,00
- l’ultima parte di imponibile, € 15.000, ricade nella fascia iniziale del 23% → imposta = 23% di € 15.000 = € 3.450

Imposta totale dovuta = 2.973,88 + 3.510,00 + 3.450,00 = 9.933,88 euro (aliquota media: 9.933,88/44.000 = 22,58%)

Scopo dell’applicazione è permettere di confrontare facilmente quanto varierebbe la tassazione di un generico contribuente (di reddito qualsiasi) al variare delle fasce di reddito e rispettive aliquote.

Per semplicità, in questa applicazione si confronteranno solo DUE scenari, denominati “fasce 2021” e “fasce 2022”.

Il file di testo **FasceAttuali.txt** contiene le fasce di reddito pre-riforma, con le relative aliquote e ampiezza della no-tax area, mentre il file di testo **FasceRiforma.txt** contiene le stesse informazioni in una ipotesi di riforma (fino a 15mila euro: aliquota del 23%; oltre 15mila euro e fino a 28mila euro: aliquota del 25%; oltre 28mila euro e fino a 50mila euro: aliquota del 35%; oltre 50mila euro: aliquota de 43%).

TEMPO STIMATO PER SVOLGERE L'INTERO COMPITO: 1h50 – 2h20

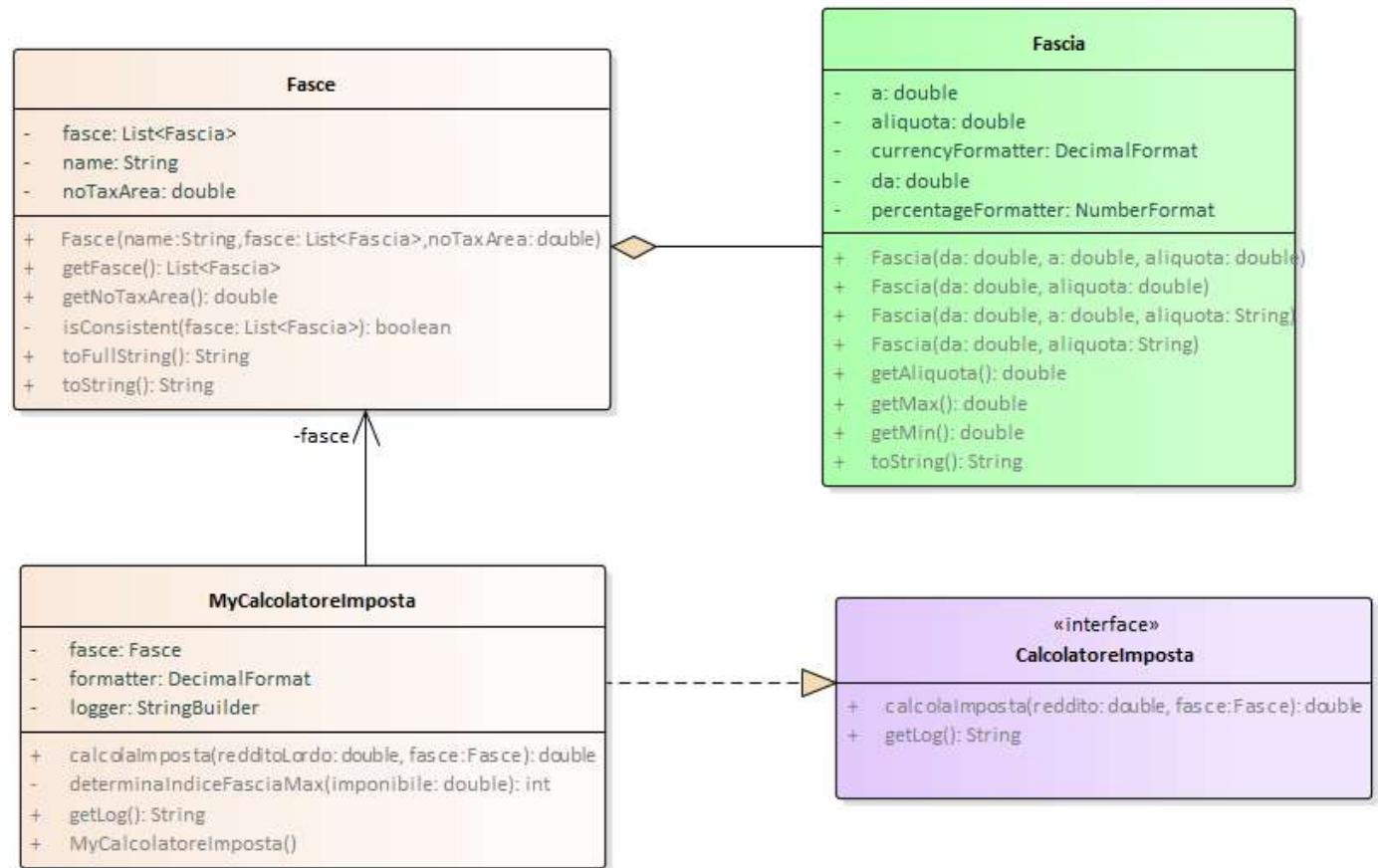
Parte 1

(punti: 20)

Modello dei dati (`taxcomparator.model`)

[TEMPO STIMATO: 50-60 minuti] (punti 13)

Il modello dei dati deve essere organizzato secondo il diagramma UML più sotto riportato.



SEMANTICA:

- la classe **Fascia** (fornita) rappresenta una fascia di reddito, definita dalle tre proprietà importo inferiore (incluso), importo superiore (escluso) e aliquota applicabile; sono disponibili vari costruttori e gli *accessor* a queste proprietà nonché una opportuna `toString`;
- la classe **Fasce** (da completare) incorpora una lista di fasce di reddito e il valore della no-tax area (un double); espone il costruttore e gli *accessor* a tali proprietà nonché una opportuna `toString`; il metodo ausiliario **isConsistent** (da implementare), richiamato dal costruttore, ha il compito di garantire la consistenza delle fasce, che devono essere
 - in ordine di reddito crescente (in particolare, la prima fascia inizia dal reddito 0)
 - adiacenti (soglia max di una fascia = soglia min della successiva)
- l'interfaccia **CalcolatoreImposta** (fornita) espone i due metodi `calcolaImposta`, che effettua il calcolo dell'imposta come da algoritmo specificato nel dominio del problema, e `getLog`, che restituisce la descrizione dettagliata del processo di calcolo

Esempio di Log del processo di calcolo

```

Imponibile lordo = € 44.000, no-tax area = € 8.174, imponibile netto = € 35.826
Fascia da € 28.000 a € 55.000, aliquota 38%
Imponibile corrente = € 35.826, imposta = € 2.973,88, imponibile restante = € 28.000
Fascia da € 15.000 a € 28.000, aliquota 27%

```

Imponibile corrente = € 28.000, imposta = € 3.510, imponibile restante = € 15.000
Fascia da € 0 a € 15.000, aliquota 23%
Imponibile corrente = € 15.000, imposta = € 3.450, imponibile restante = € 0%

- d) la classe ***MyCalcolatoreImposta*** (da realizzare) implementa tale interfaccia. È di particolare rilevanza che tutti i valori di reddito e imposta siano correttamente formattati in Euro col simbolo di valuta davanti (anziché dietro) e due cifre decimali, come da esempio sopra. (SUGGERIMENTO: studiare l'implementazione di ***Fascia***)

Persistenza (*taxcomparator.persistence*)

[TEMPO STIMATO: 30-40 minuti] (punti 8)

Come già anticipato, i due file di testo **FasceAttuali.txt** e **FasceRiforma.txt** contengono due distinti scenari di tassazione, ma sono organizzati secondo la medesima struttura:

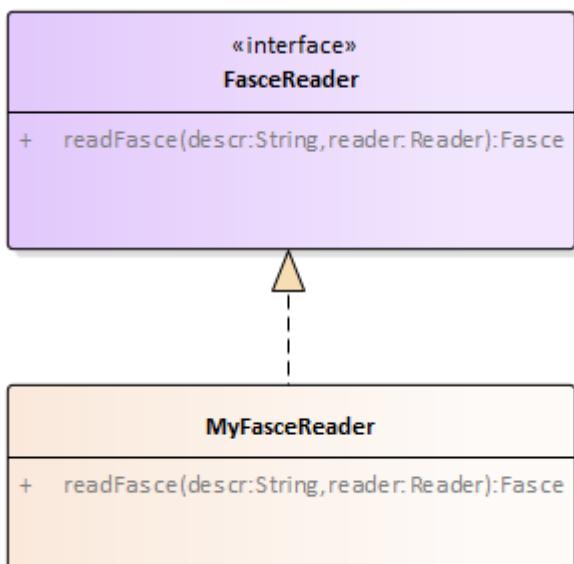
- la prima riga descrive l'ampiezza della no-tax area, nel formato “no-tax area: *valore*” (v. esempio sotto). L'espressione “no-tax area” può essere scritta indifferentemente con maiuscole e/o minuscole, comunque mischiate: dopo il carattere ‘:’ e prima del valore numerico possono essere presenti una quantità arbitraria di spazi e/o tabulazioni. Il valore numerico è espresso secondo le convenzioni culturali italiane.
- le righe successive descrivono le fasce di reddito, in ordine di reddito crescente, secondo il formato:

ESEMPIO <i>Fasce.txt</i>		
no-tax area:	8.174	
0-----15.000:	23%	
15.000-28.000:	27%	
28.000-55.000:	38%	
55.000-75.000:	41%	
75.000--oltre:	43%	

LimiteInferiore-LimiteSuperiore: aliquotaPercentuale

dove *LimiteInferiore* e *LimiteSuperiore* sono valori numerici espressi secondo le convenzioni culturali italiane, tranne che per l'ultima fascia, in cui *LimiteSuperiore* è la stringa “oltre”, scritta indifferentemente con maiuscole e/o minuscole; *aliquotaPercentuale* è una percentuale, formattata come tale.

- Specifiche sui separatori:
 - la quantità di “lineette” ‘-’ fra i due limiti numerici non è fissa, l'unico vincolo è che ce ne sia almeno una
 - fra il carattere ‘:’ e la percentuale può essere presente una quantità arbitraria di tabulazioni (ma non spazi); più in generale, qualunque mix di ‘-’, ‘:’ e tabulazioni costituisce un separatore lecito fra gli tutti gli elementi di queste righe.
- Specifiche sulle fasce: è garantito che le fasce siano adiacenti nel senso sopra specificato, ossia che ogni fascia (superiore alla prima) abbia come limite inferiore il limite superiore della fascia precedente, e che la prima fascia abbia come limite inferiore 0, in modo da “non lasciare buchi” nella sequenza delle fasce.



L'interfaccia ***FasceReader*** (fornita) dichiara il metodo di lettura ***readFasce*** che riceve come argomento una descrizione testuale (il nome convenzionale dello scenario di tassazione che viene letto) e un ***Reader*** (già aperto): suo compito è leggere un file nel formato sopra specificato, restituendo un oggetto ***Fasce*** completamente configurato

La classe ***MyFasceReader*** (da realizzare) implementa tale interfaccia effettuando puntuali controlli sul formato del file e lanciando ***BadFormatException*** (fornita) con opportuno messaggio d'errore in caso di errori nel formato del file, mentre eventuali ***IOException*** devono essere lasciate fluire all'esterno. Non occorrono costruttori poiché non viene mantenuto uno stato interno.

Parte 2

[TEMPO STIMATO: 30-40 minuti]

(punti: 9)

Controller (taxcomparator.controller)

(punti 0)



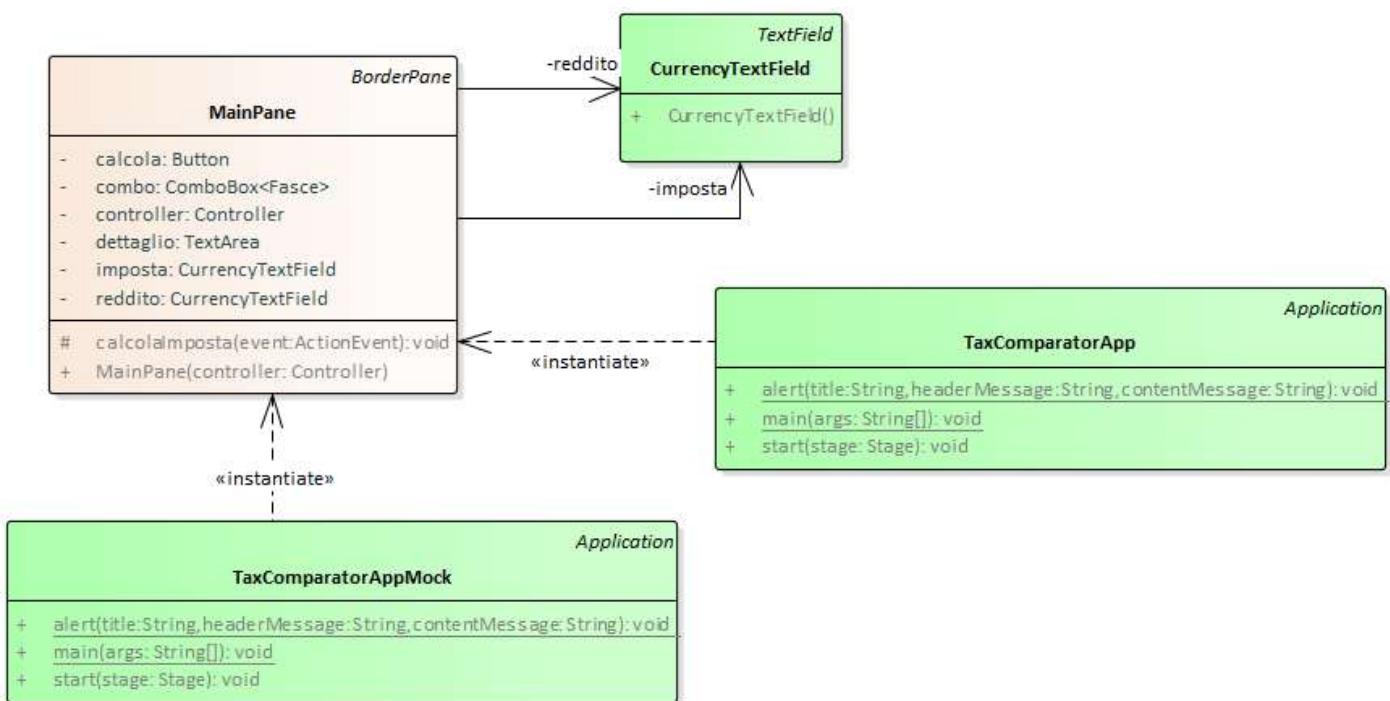
La classe **Controller** (fornita) rappresenta il controller dell'applicazione:

- il costruttore riceve un **CalcolatoreImposta** e la lista delle **Fasce** disponibili per il calcolo; quest'ultima è recuperabile tramite il metodo **getFasceDisponibili**
- la coppia di metodi **getFasce** / **setFasce** permette di recuperare o impostare le **Fasce** da usare per il calcolo
- il metodo **getLog** restituisce la stringa che descrive il processo di calcolo
- il metodo **calcolalImposte** effettua il calcolo dell'imposta relativa al reddito imponibile lordo ricevuto
- il metodo statico **alert** può essere utile per mostrare all'utente messaggi di errore.

Interfaccia utente (taxcomparator.ui)

[TEMPO STIMATO: 30-40 minuti] (punti 9)

L'interfaccia utente è illustrata nelle figure seguenti e segue il modello sotto illustrato:

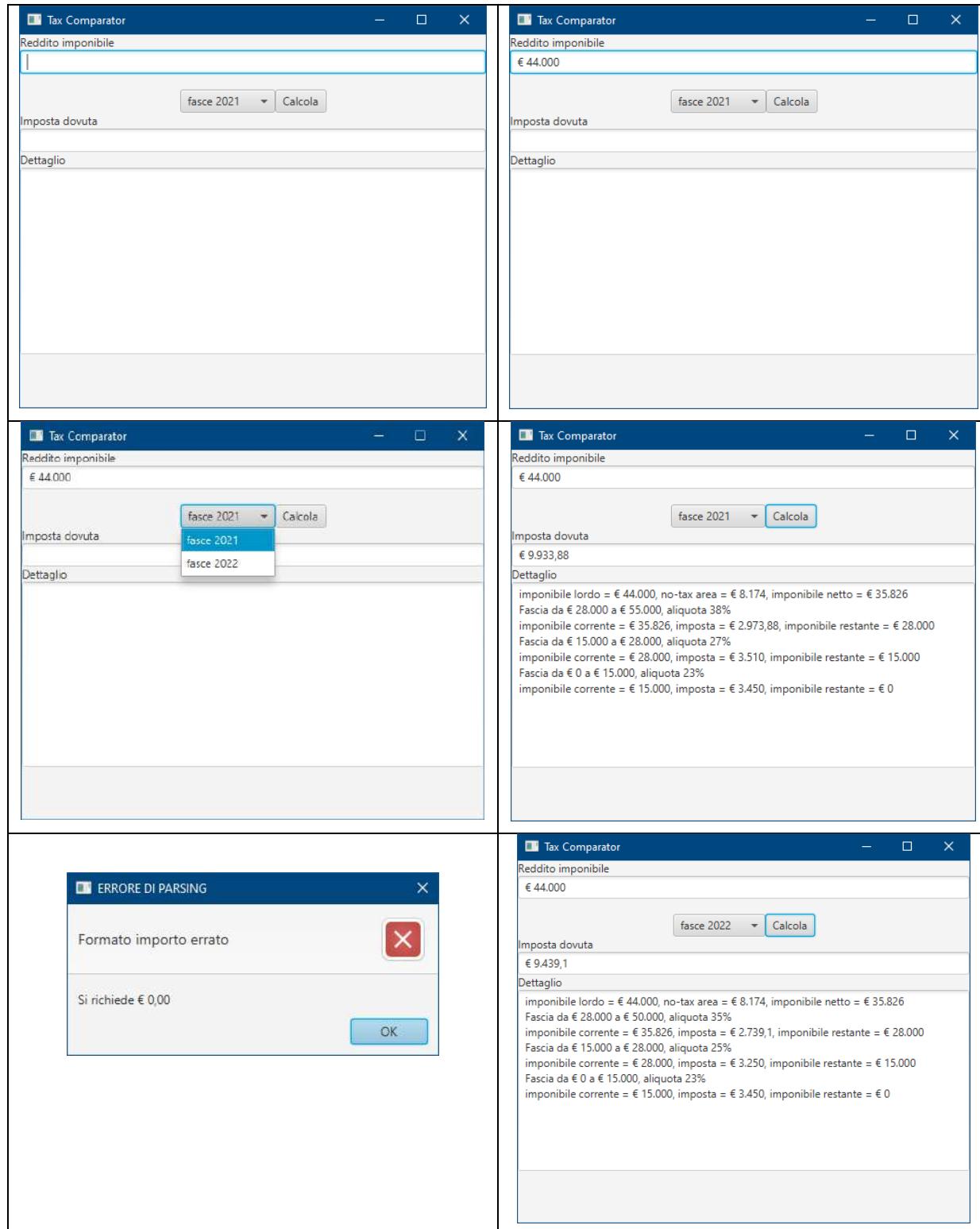


La classe **MainPane (da completare)**, il cui costruttore prende in ingresso un **Controller**, è composta da due speciali campi di testo di tipo **CurrencyTextField** (fornita), una combo, un pulsante e un'area di testo (oltre a varie etichette),

come sotto illustrato. **CurrencyTextField** è una specializzazione del campo di testo con la proprietà di *formattare automaticamente in euro, secondo le convenzioni italiane*, gli importi numeri in esso digitati o visualizzati.

Operativamente, l'utente deve digitare il reddito imponibile lordo nel campo di testo in alto, e scegliere dalla combo le fasce da usare (importante: di default dev'essere sempre selezionata la prima, la selezione non dev'essere inizialmente vuota!): premendo il pulsante Calcola, viene svolto il calcolo, mostrando il valore dell'imposta calcolata nel campo di testo apposito, e il dettaglio del calcolo nell'area sottostante.

Nel caso in cui l'utente digiti un valore non numerico o mal formattato, dev'essere visualizzato apposito messaggio di errore nel pop-up apposito (ottenibile dal metodo **alert** del **Controller**).



Cose da ricordare

- salva costantemente il tuo lavoro: l'informatica a volte può essere "subdolamente ostile" ..
- in particolare: se ora compila e stai per fare modifiche, salva la versione attuale (non si sa mai)

Checklist di consegna

- Hai fatto un **JAR eseguibile**, che contenga cioè l'indicazione del main?
- Hai controllato che **si compil e ci sia tutto?** [NB: non includere il PDF del testo]
- Hai **rinominato IL PROGETTO**, lo ZIP e il JAR esattamente come richiesto?
- Hai **chiamato** la cartella del progetto esattamente come richiesto?
- **Hai fatto un unico file ZIP (NON .7z, rar o altri formati) contenente l'intero progetto?**
In particolare, ti sei assicurato di aver incluso tutti i file .java (e non solo i .class)?
- **Hai consegnato DUE file distinti, ossia lo ZIP col progetto e il JAR eseguibile?**
- Su EOL, hai **premuto** il tasto "CONFERMA" per inviare il tuo elaborato?

ESAME DI FONDAMENTI DI INFORMATICA T-2 del 14/9/2021

Proff. E. Denti – R. Calegari – A. Molesini

Tempo a disposizione: 3 ore

NOME PROGETTO ECLIPSE: CognomeNome-matricola (es. RossiMario-0000123456)

NOME CARTELLA PROGETTO: CognomeNome-matricola (es. RossiMario-0000123456)

NOME ZIP DA CONSEGNARE: CognomeNome-matricola.zip (es. RossiMario-0000123456.zip)

NOME JAR DA CONSEGNARE: CognomeNome-matricola.jar (es. RossiMario-0000123456.jar)

Si devono consegnare DUE FILE: *l'intero progetto Eclipse e il JAR eseguibile*

Si ricorda che compiti *non compilabili o palesemente lontani da 18/30* NON SARANNO CORRETTI e causeranno la verbalizzazione del giudizio “RESPINTO”

L’azienda *EDLift* ha richiesto lo sviluppo di un’applicazione per simulare il funzionamento dei suoi ascensori, alcuni dei quali hanno un comportamento quanto meno.. curioso ☺

DESCRIZIONE DEL DOMINIO DEL PROBLEMA.

Un ascensore serve un dato *edificio*, costituito da un certo numero di *piani*, sia sopra terra (identificati da numeri positivi) sia sotto terra (identificati da numeri negativi); per convenzione, il piano terra è il piano 0.

Ogni piano è dotato di un *pulsante di chiamata*, che serve per chiamare l’ascensore, e di un *display informativo*, che indica dove si trova l’ascensore in quel momento e fornisce altre informazioni sullo stato dell’ascensore.

Un ascensore può essere di diversi tipi:

- **Basic** - serve una singola chiamata per volta: se è libero e viene chiamato a un piano, inizia a muoversi verso quel piano senza effettuare fermate intermedie, ignorando/rifiutando ogni altra chiamata intervenuta nel frattempo. Questa modalità di funzionamento è tipica dei condomini privati e dei piccoli alberghi.
- **Multipiano** - cerca di ottimizzare gli spostamenti servendo più utenti per volta: perciò se, mentre si sta muovendo dal piano Pa verso il piano Pb, viene chiamato a un piano intermedio Pi, esso effettua la fermata intermedia al piano Pi, riprendendo poi il suo viaggio verso il piano Pb di destinazione originaria. Questa installazione è tipicamente presente negli hotel di medie/grandi dimensioni, ospedali, etc.
- **Salutista** (prodotto esclusivo di Edlift!) – simile al basic, promuove però uno stile di vita sano, mirato a combattere l’eccessiva pigrizia. Pertanto, a) non accetta chiamate per piani adiacenti (quello sopra e sotto) a quello a cui già ci si trova, invitando a usare le scale; b) nelle altre chiamate si ferma sempre un piano prima di quanto richiesto, così da costringere l’utente a fare sempre almeno un piano a piedi.

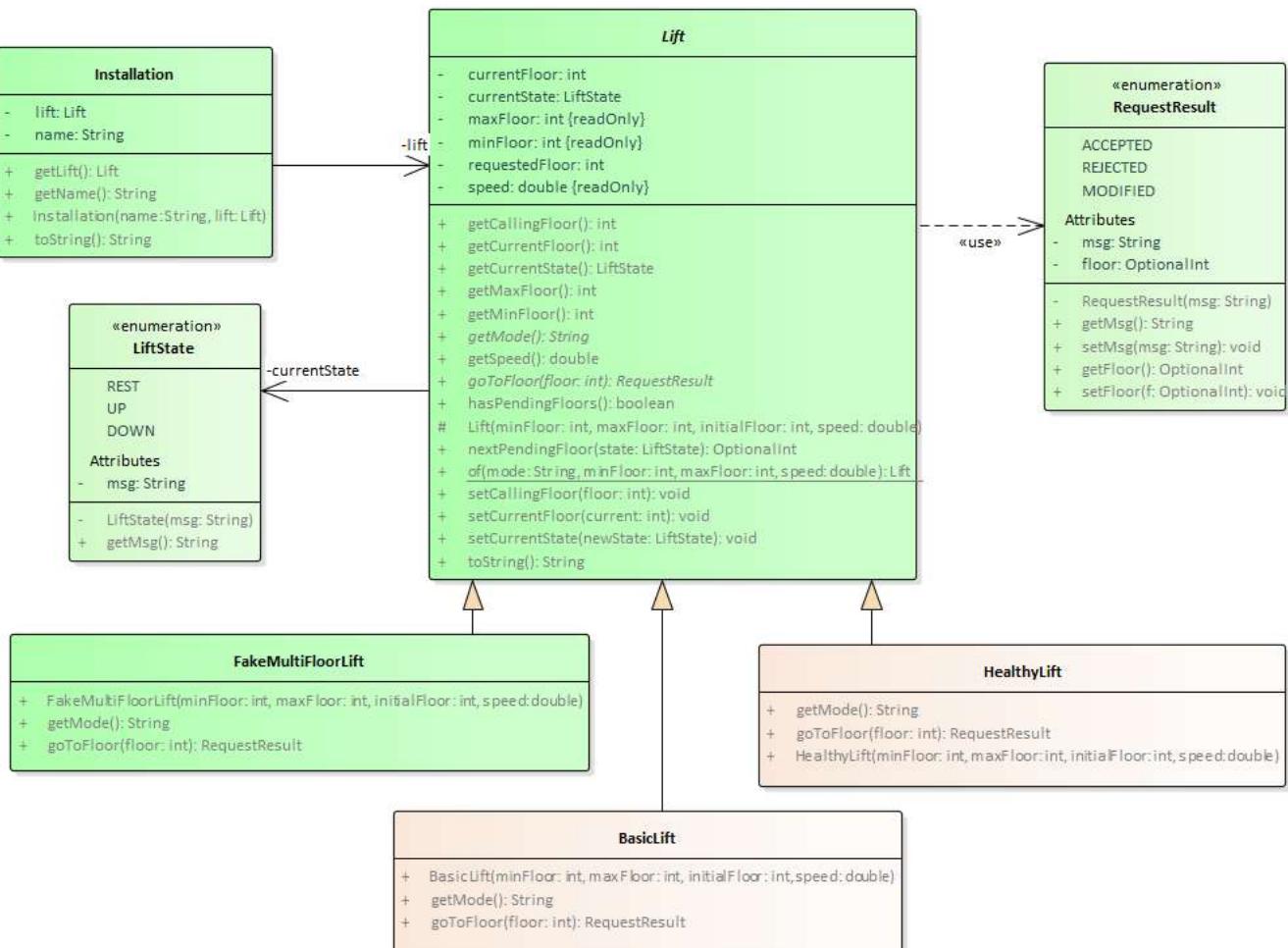
In questo esame saranno implementati soltanto l’ascensore base e quello salutista, pur predisponendo l’architettura e il reader per la futura estensione ad altre tipologie di ascensori.

Il file di testo [Installazioni.txt](#) contiene la descrizione di vari ascensori, specificando per ciascuno:

- il nome dell’edificio in cui è installato
- la modalità di funzionamento (base, multipiano, salutista)
- il numero di piani e loro numerazione (es. 6 piani da -1 a 4, 5 piani da -2 a 2, 4 piani da 0 a 3, etc.)
- la velocità dell’ascensore in metri/secondo (**non utilizzata però nell’algoritmica di questo compito**)

TEMPO STIMATO PER SVOLGERE L’INTERO COMPITO: 1h40 – 2h15

Il modello dei dati deve essere organizzato secondo il diagramma UML più sotto riportato.



SEMANTICA:

- la classe-dati **Installation** (fornita) è un mero contenitore che accoppia una descrizione testuale a un **Lift**, con appositi accessori e **toString**
- l'enumerativo **LiftState** (fornito) definisce i tre stati possibili – **REST** (fermo al piano), **UP** e **DOWN**, con corrispondente messaggio incapsulato, accessibile tramite apposito accessor;
- l'enumerativo **RequestResult** (fornito) definisce i tre risultati possibili, **ACCEPTED**, **REJECTED** e **MODIFIED**: tutti permettono di impostare sia un messaggio incapsulato sia un valore **OptionalInt**, accessibili tramite accessor;
- la classe astratta **Lift** (fornita) fattorizza le caratteristiche comuni a tutti gli ascensori, ma intenzionalmente NON contiene alcuna logica di gestione della simulazione: ciò sarà compito del controller.
 - proprietà rilevanti: *piano minimo*, *piano massimo*, *piano corrente* (ovviamente compreso fra gli altri due, estremi inclusi), *velocità*, *stato interno* (di tipo **LiftState**) e *piano chiamante* (ossa il piano da cui viene effettuata una chiamata)
 - il costruttore (protetto) accetta quattro argomenti: piano minimo, piano massimo, piano a cui l'ascensore si trova inizialmente e velocità; provvede altresì a impostare lo stato interno dell'ascensore al valore iniziale REST (ascensore fermo) e il piano chiamante uguale al piano iniziale
 - gli accessori pubblici **getMinFloor**, **getMaxFloor**, **getCurrentFloor**, **getSpeed** restituiscono lo stato attuale di tali proprietà; la coppia **getCurrentState**/**setCurrentState** consente di accedere allo stato corrente, mentre la coppia **getCallingFloor**/**setCallingFloor** consente di accedere al piano chiamante

- l'accessore astratto `getMode` restituisce una stringa indicativa dello specifico tipo di ascensore: la sua implementazione è a cura delle sottoclassi concrete
- il metodo `toString` restituisce una rappresentazione essenziale dell'ascensore
- la factory internalizzata `Lift.of` crea l'opportuno tipo di ascensore, in base al modo specificato dal primo argomento (stringa): sono previsti i modi “**BASIC**”, “**MULTI**” e “**HEALTHY**”, che determinano la costruzione rispettivamente di un **BasicLift**, di un **MultiFloorLift** e di un **HealthyLift**. La factory provvede altresì a stabilire il piano iniziale a cui l'ascensore si trova di default.
- il metodo astratto `goToFloor` esprime il desiderio di muoversi verso il piano indicato: le sue implementazioni concrete implementeranno la logica di movimento dello specifico ascensore
- la coppia di metodi `hasPendingFloors` e `nextPendingFloor` cattura l'idea di eventuale “lista di chiamate pendenti” da gestire secondo una qualche logica: il primo metodo è vero se esistono chiamate pendenti ancora da servire, il secondo estrae e restituisce da tale lista il “prossimo” piano da servire (un **OptionalInt**) secondo la logica dello specifico. Da notare che `nextPendingFloor` per definizione non è idempotente, in quanto estrae un elemento dall'elenco delle chiamate pendenti, che pertanto viene alterato. L'implementazione di default non gestisce alcuna lista di chiamate: pertanto `hasPendingFloors` restituisce sempre false, mentre `nextPendingFloor` restituisce sempre un optional vuoto.

Per “far partire” l'ascensore basta impostare il piano desiderato con `setCallingFloor` e lo stato dell'ascensore a UP o DOWN con `setCurrentState`, in base alla direzione di movimento desiderata. L'effettiva “messa in azione” dell'ascensore è poi delegata al controller (fornito), che contiene tutta la logica di simulazione (vedere test).

- la classe ***BasicLift (da realizzare)*** concretizza ***Lift*** nel caso specifico dell'ascensore base: **punti: 3**
 - il costruttore accetta gli stessi argomenti del costruttore di ***Lift***
 - non è prevista alcuna lista di chiamate pendenti: pertanto, non occorre ridefinire `hasPendingFloors` e `nextPendingFloor` in quanto il loro risultato di default (falso nel primo caso, optional vuoto nel secondo) è adeguato allo scopo
 - la logica di servizio delle chiamate espressa da `goToFloor` deve dapprima verificare gli argomenti, lanciando ***IllegalArgumentException*** se il piano di destinazione è fuori range, poi:
 - se l'ascensore è fermo, impostare il piano di destinazione desiderato e restituire **ACCEPTED**
 - altrimenti, restituire **REJECTED**
- la classe ***HealthyLift (da realizzare)*** concretizza ***Lift*** nel caso specifico dell'ascensore salutista **punti: 6**
 - il funzionamento è analogo a quello dell'ascensore base, inclusa la scelta di non gestire chiamate pendenti; l'unica differenza è nella logica di funzionamento del metodo `goToFloor`
 - Il metodo `goToFloor` deve anche in questo caso prima verificare gli argomenti, lanciando ***IllegalArgumentException*** se il piano di destinazione è fuori range, poi:
 - come sopra, se l'ascensore non è fermo, rifiuta subito la chiamata restituendo **REJECTED**
 - se, invece, l'ascensore è fermo:
 - a) calcola la differenza in piani fra il piano richiesto e quello corrente
 - b) se essa è non superiore a 1, rifiuta la chiamata; se è superiore, calcola il nuovo piano di arrivo tenendo conto del requisito di far fare sempre un piano a piedi, lo imposta con `setCallingFloor` e restituisce il risultato **MODIFIED** integrando in esso un apposito messaggio e il valore del piano di destinazione calcolato (metodi `setMsg`/`setResult`)

Persistenza (package edlift.persistence)

[TEMPO STIMATO: 40-50 minuti] punti: 10

Come già anticipato, il file di testo [Installazioni.txt](#) contiene la descrizione di ascensori installati in un vari edifici. Ogni ascensore è descritto da un record di tre righe, che specificano rispettivamente:

- il nome dell'installazione
- il tipo di ascensore, il numero di piani e la loro numerazione (es. 6 piani da -1 a 4, 5 piani da -2 a 2, etc.)
- la velocità dell'ascensore in metri/secondo

Tali informazioni sono espresse nel seguente formato (tutte le parole chiave sono case-insensitive):

- la prima riga, introdotta dalla parola **ASCENSORE**, dà il nome dell'installazione: dopo la parola ASCENSORE vi è almeno uno spazio, seguito dalla descrizione (che può contenere qualunque carattere) fino a fine riga
- la seconda riga, introdotta dalla parola **TIPO**, specifica il modo di funzionamento ("BASIC", "MULTI" o "HEALTHY"), il numero di piani e il loro intervallo, nel formato **TIPO MODO A N PIANI** da **MIN** a **MAX**
- la velocità dell'ascensore in metri/secondo, nel formato **VELOCITA N m/s**

Esempio di file *Installazioni.txt*

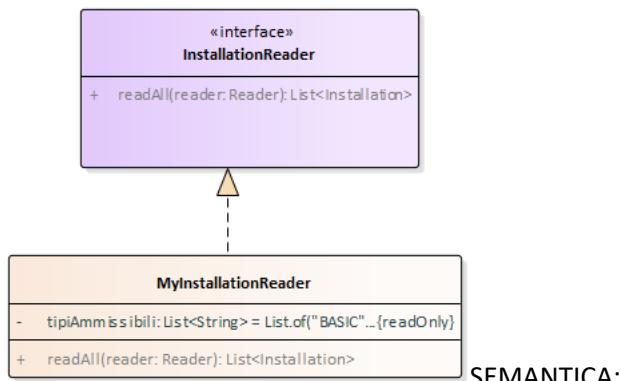
```
ASCENSORE HOTEL MIRALAGO
TIPO MULTI A 7 PIANI da -2 a 4
VELOCITA 1 m/s
```

```
ASCENSORE Condominio Girasoli
TIPO BASIC A 8 PIANI da -1 a 6
VELOCITA 0.9 m/s
```

```
ASCENSORE Palazzina Ferrari
TIPO BASIC A 5 PIANI da 0 a 4
VELOCITA 0.15 m/s
```

```
ASCENSORE Grattacielo Salutisti
TIPO HEALTHY A 10 PIANI da -2 a 7
VELOCITA 1 m/s
```

L'architettura software è illustrata nel diagramma UML che segue:



SEMANTICA:

- a) l'interfaccia **InstallationReader** (fornita) dichiara il metodo **readAll** che restituisce una lista di **Installation**, lanciando, oltre alla "naturale" **IOException**, una **BadFormatException** nel caso di errore nel formato del file;
- b) la classe **MyInstallationReader** (**da realizzare**) concretizza **InstallationReader** implementando **readAll** secondo il formato del file sopra descritto, effettuando accurate verifiche di formato ed emettendo, nel caso, **BadFormatException** con dettagliato messaggio d'errore

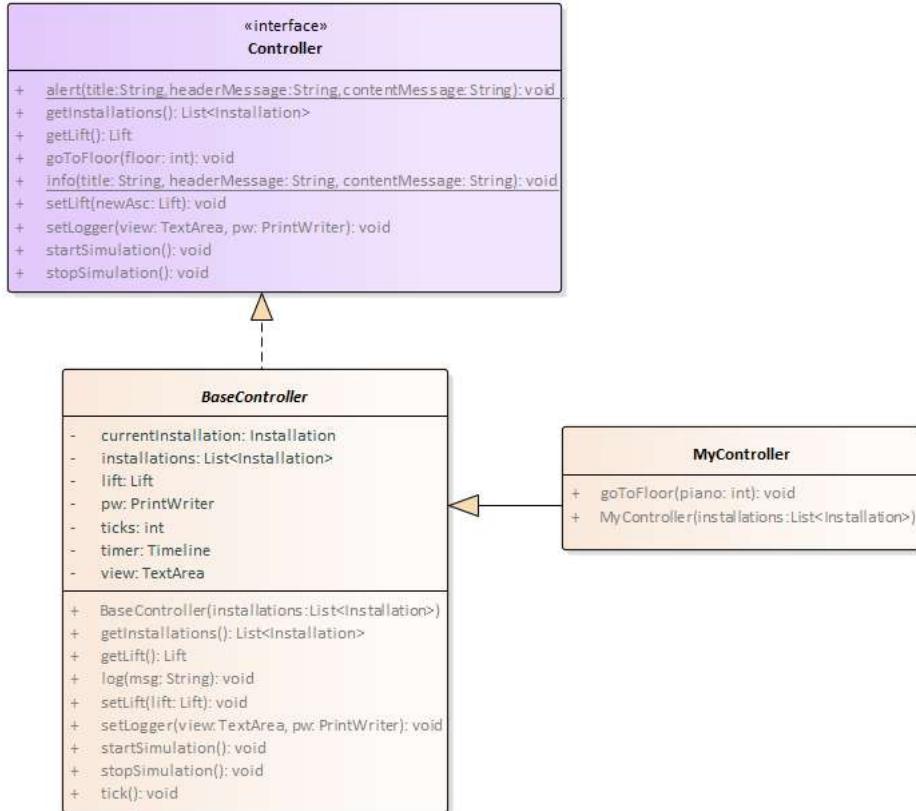
Parte 2

punti: 11

Controller (package edlift.controller)

punti: 0

La parte di controllo, strutturata nella triade interfaccia (**Controller**), implementazione base astratta (**BaseController**) e implementazione specifica (**MyController**), è fornita già implementato secondo il diagramma UML in figura



SEMANTICA

a) l'interfaccia **Controller** dichiara i metodi:

- **getInstallations** restituisce la lista delle installazioni (proveniente dal reader)
- **startSimulation**/**stopSimulation** avviano/fermano la simulazione
- **setLogger** imposta i dispositivi di uscita: in particolare il primo dev'essere la `TextArea` della GUI, il secondo un `PrintWriter` (tipicamente `System.out` adeguatamente incapsulato) per l'output su console
- **setLift**/**getLift** rispettivamente impostano/recuperano l'ascensore attualmente oggetto della simulazione
- **alert** e **info** (statici) per emettere avvisi all'utente tramite finestre di dialogo a comparsa

b) la classe astratta **BaseController** implementa quasi totalmente la logica della simulazione

- il costruttore riceve le lista delle **Installation** simulabili
- il metodo **log** emette un messaggio sui dispositivi di output preventivamente impostati con **setLogger**
- a cadenza di un secondo, tramite il metodo privato **tick**, aggiorna lo stato dell'ascensore secondo una logica invariante rispetto allo specifico tipo di ascensore, espressa tramite una macchina a stati: emette, tramite il metodo **log**, opportuni messaggi che permettono di seguirne il funzionamento

c) la classe concreta **MyController** implementa l'unico metodo rimasto astratto, **goToFloor**, anch'esso in modo generale rispetto allo specifico tipo di ascensore.

Interfaccia utente (package edlift.ui)

[TEMPO STIMATO: 40-50 minuti]

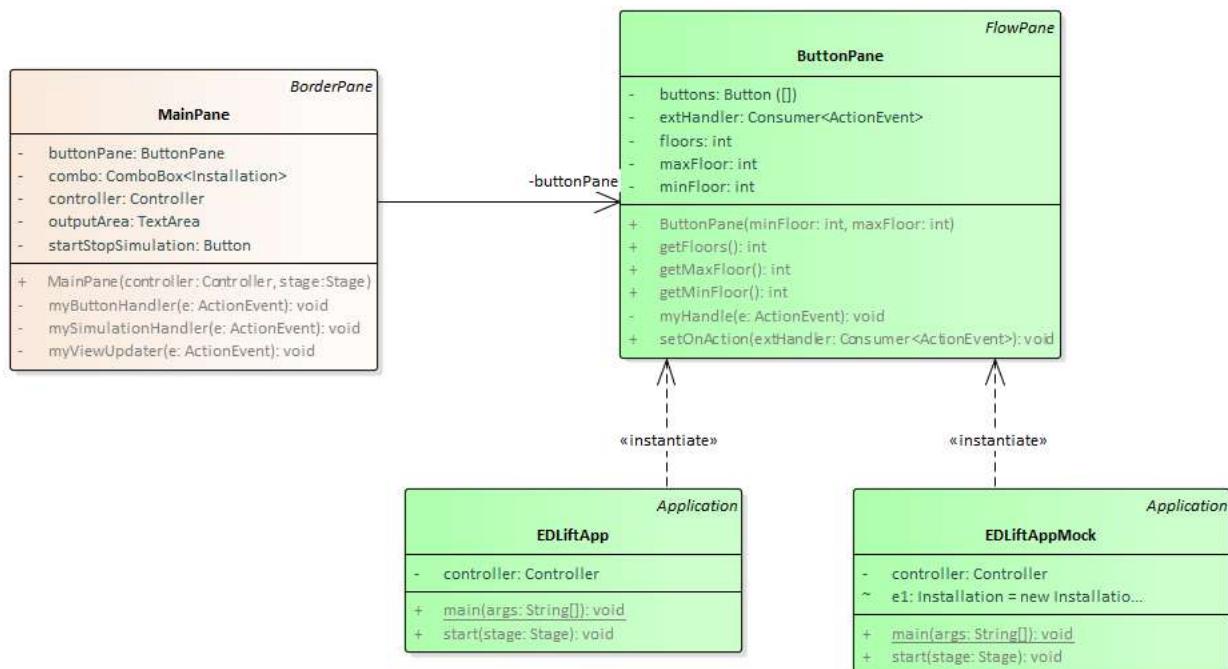
punti: 11

L'interfaccia (Figg. 1-2-3-4) mostra il pannello che governa la simulazione: una combo – preventivamente popolata con tutte le installazioni disponibili – consente di scegliere l'ascensore da simulare: ciò causa l'istanziazione dell'opportuna buttoniera (in basso), con cui l'utente può simulare la chiamata dell'ascensore ai vari piani. Sulla destra, una textarea disposta verticalmente funge da dispositivo di uscita.

Un apposito pulsante *Start/Stop Simulation* – il cui testo cambia da “*Start simulation*” a *Stop simulation*” a seconda dello stato della simulazione stessa – consente di avviare o fermare la simulazione.

La classe ***EDLiftApp*** (fornita) costituisce l'applicazione JavaFX che si occupa di aprire i file, creare il controller e incorporare il ***MainPane***. Per consentire di collaudare la GUI anche in assenza / in caso di malfunzionamento della parte di persistenza, è possibile avviare l'applicazione mediante la classe ***EDLiftAppMock***.

L'architettura segue il modello sotto illustrato:



SEMANTICA:

- La classe ***ButtonPanel*** (fornita) costituisce la pulsantiera: il costruttore riceve i piani minimo e massimo, e provvede a istanziare il giusto numero di pulsanti con la opportune etichette. Nel complesso la pulsantiera è gestibile come se fosse un bottone singolo: il gestore dell'evento va impostato tramite il metodo *setOnAction*, il cui argomento *ActionEvent* consente di risalire allo specifico bottone premuto
- La classe ***EDLiftApp*** (e la sua consorella ***-Mock***) costituisce l'entry point dell'applicazione.
- La classe ***MainPane* (da completare)** rappresenta la GUI dell'applicazione
 - a sinistra, una combo popolata con tutte le installazioni consente di scegliere quale ascensore simulare; la scelta dell'ascensore deve causare l'istanziazione dell'opportuna buttoniera, da collocare in basso **Quando si cambia ascensore la view dev'essere aggiornata, nel seguente modo:**
 - sostituendo la buttoniera con una nuova adatta ai piani del nuovo edificio
 - svuotando la textarea che rappresenta il dispositivo di uscita
 - fermando e subito riattivando la simulazione, come se fosse stato premuto due volte il pulsante Start/Stop simulation, così che il controller possa azzerare il clock della simulazione
 - sotto alla combo, il pulsante Start/Stop Simulation agisce sul controller per avviare/fermare la simulazione; il testo del pulsante si deve modificare di conseguenza (Figg.1-2-3-4)

- al centro, una textarea rappresenta il dispositivo di uscita, da collegare al controller mediante setLogger.

FUNZIONAMENTO PREVISTO: inizialmente, si sceglie l'installazione desiderata, poi si avvia la simulazione: ciò causa l'apparire dei primi messaggi di output. Premendo uno dei pulsanti di chiamata, l'ascensore si comporta di conseguenza, accettando o rifiutando o modificando la chiamata in base alla propria logica di funzionamento.

Nel caso degli ascensori Base e Salutista, non sono accettate chiamate mentre l'ascensore è in moto (Fig 2): quello salutista può rifiutare una chiamata anche se proveniente da un piano adiacente (Fig. 3), o può accettarla modificando il piano di destinazione in base alla sua logica (Fig. 3).

In futuro verrà implementato anche l'ascensore Multipiano (NON MOSTRATO).

La parte da realizzare riguarda:

1. il popolamento della combo
2. la gestione eventi dei tre elementi, ovvero:
 - il metodo *mySimulationHandler* che gestisce il pulsante Start/Stop simulation
 - il metodo *myViewUpdater* che aggiorna la view a seguito del cambio di ascensore nella combo
 - il metodo *myButtonHandler* che gestisce la buttoniera

Cose da ricordare

- salva costantemente il tuo lavoro: l'informatica a volte può essere "subdolamente ostile" ..
- in particolare: se ora compila e stai per fare modifiche, salva la versione attuale (non si sa mai)

Checklist di consegna

- Hai fatto un **JAR eseguibile**, che contenga cioè l'indicazione del main?
- Hai controllato che **si compili e ci sia tutto?** [NB: non includere il PDF del testo]
- Hai **rinominato IL PROGETTO**, lo ZIP e il JAR esattamente come richiesto?
- Hai **chiamato** la cartella del progetto esattamente come richiesto?
- **Hai fatto un unico file ZIP (NON .7z, rar o altri formati) contenente l'intero progetto?**
In particolare, ti sei assicurato di aver incluso tutti i file .java (e non solo i .class)?
- **Hai consegnato DUE file distinti, ossia lo ZIP col progetto e il JAR eseguibile?**
- Su EOL, hai premuto il tasto "CONFERMA" per inviare il tuo elaborato?

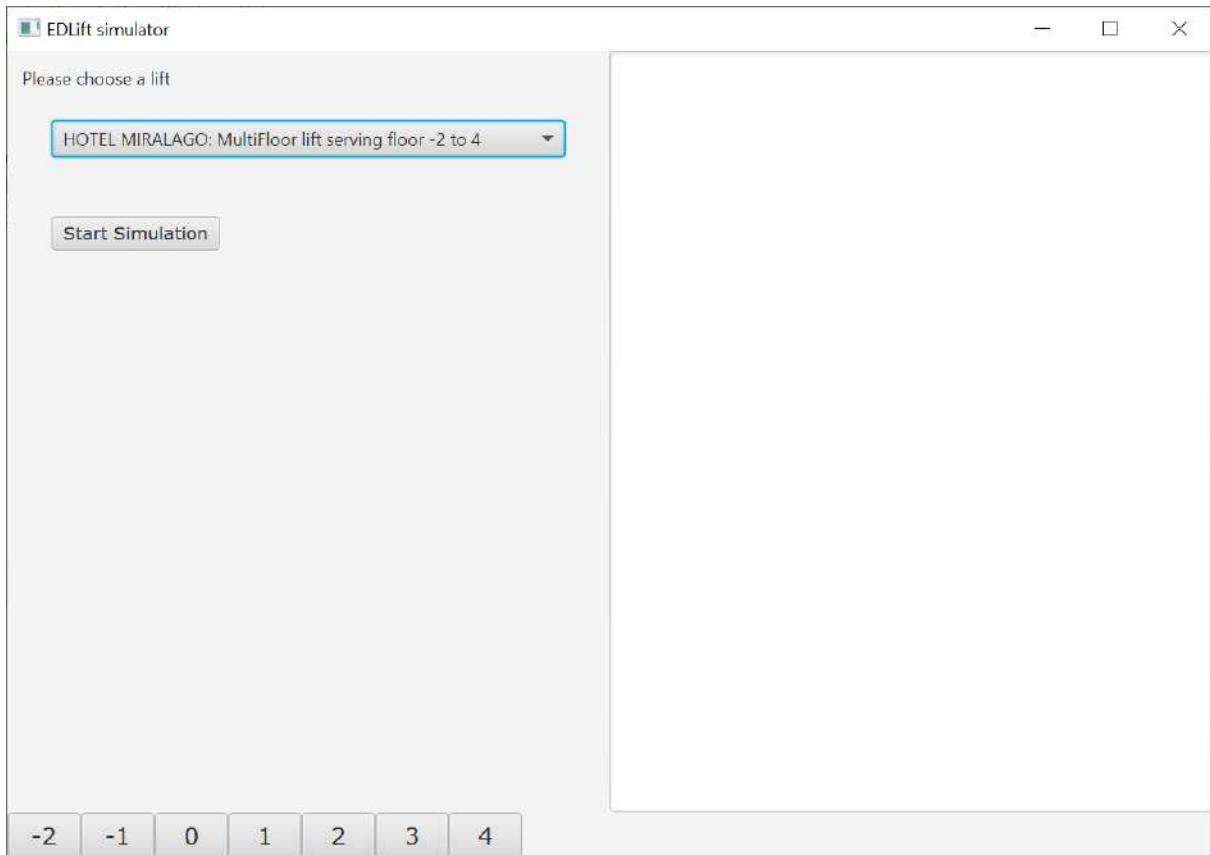


Figura 1

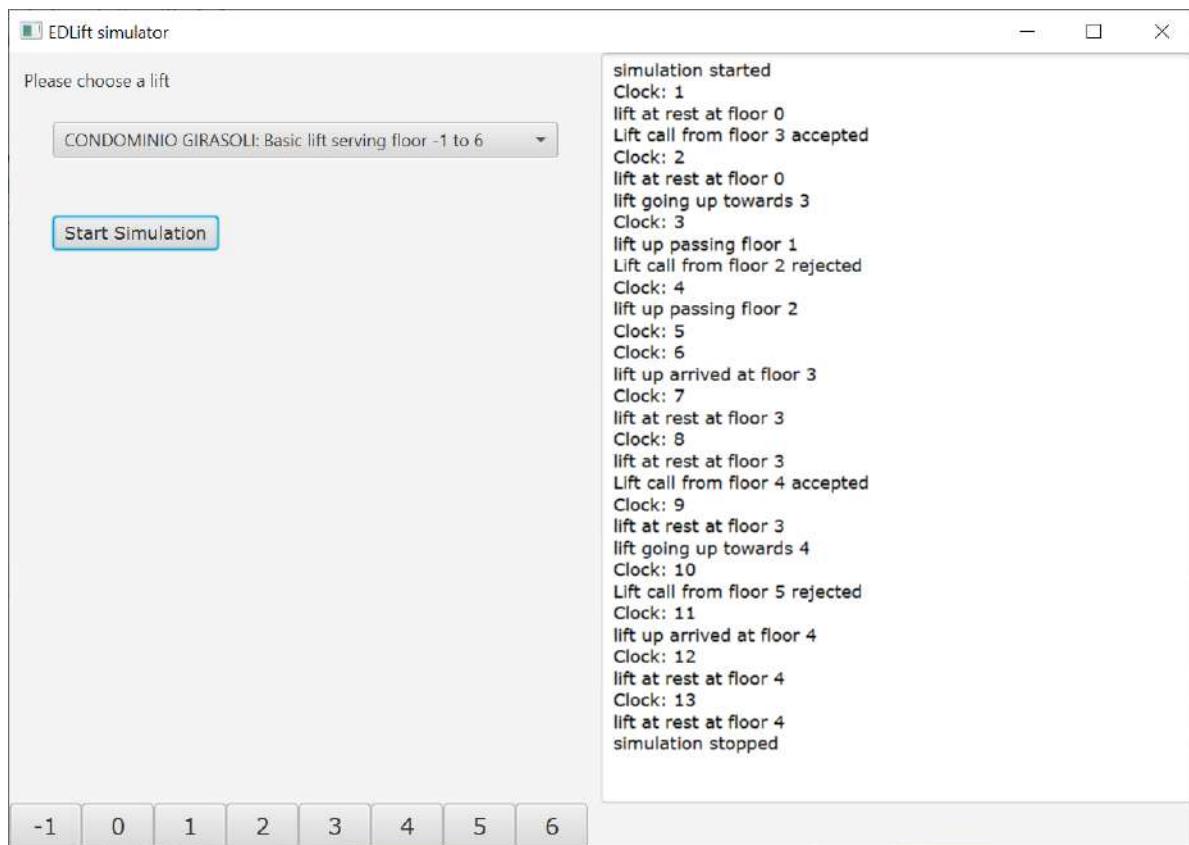


Figura 2

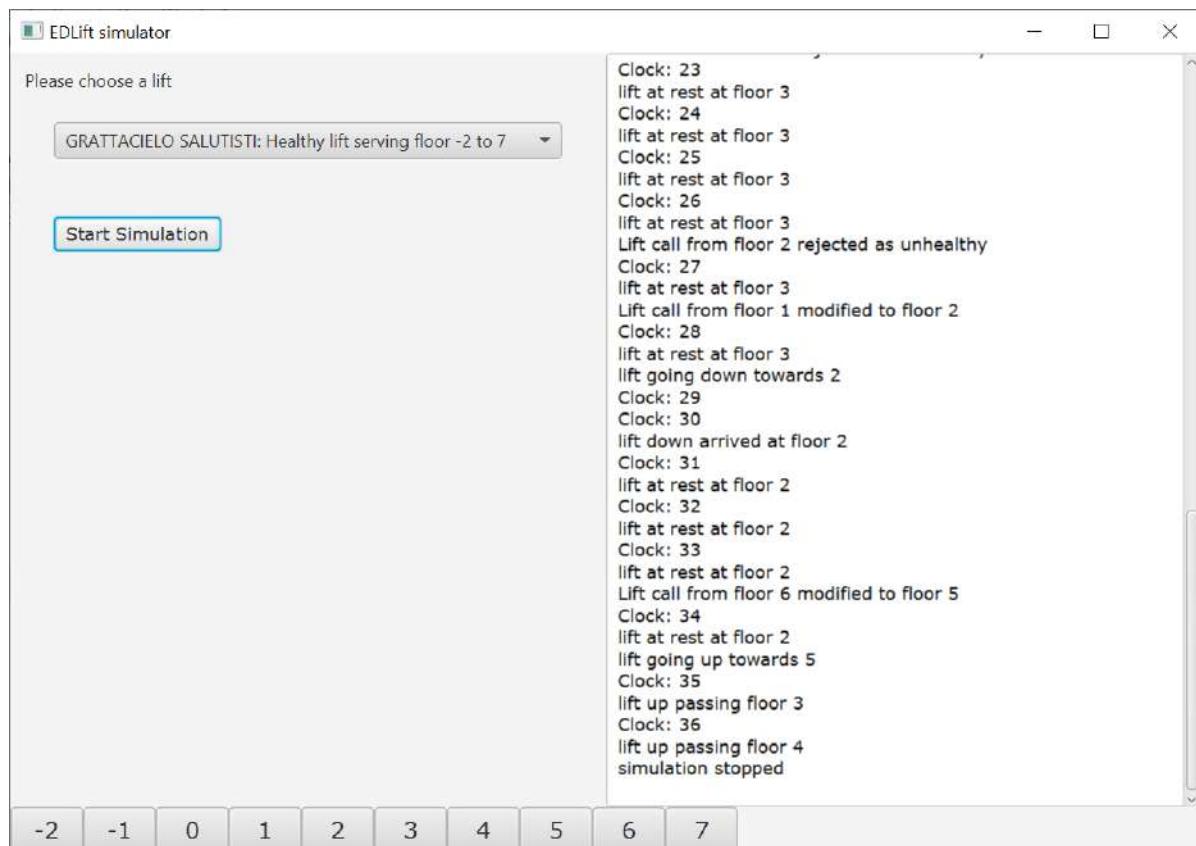


Figura 3

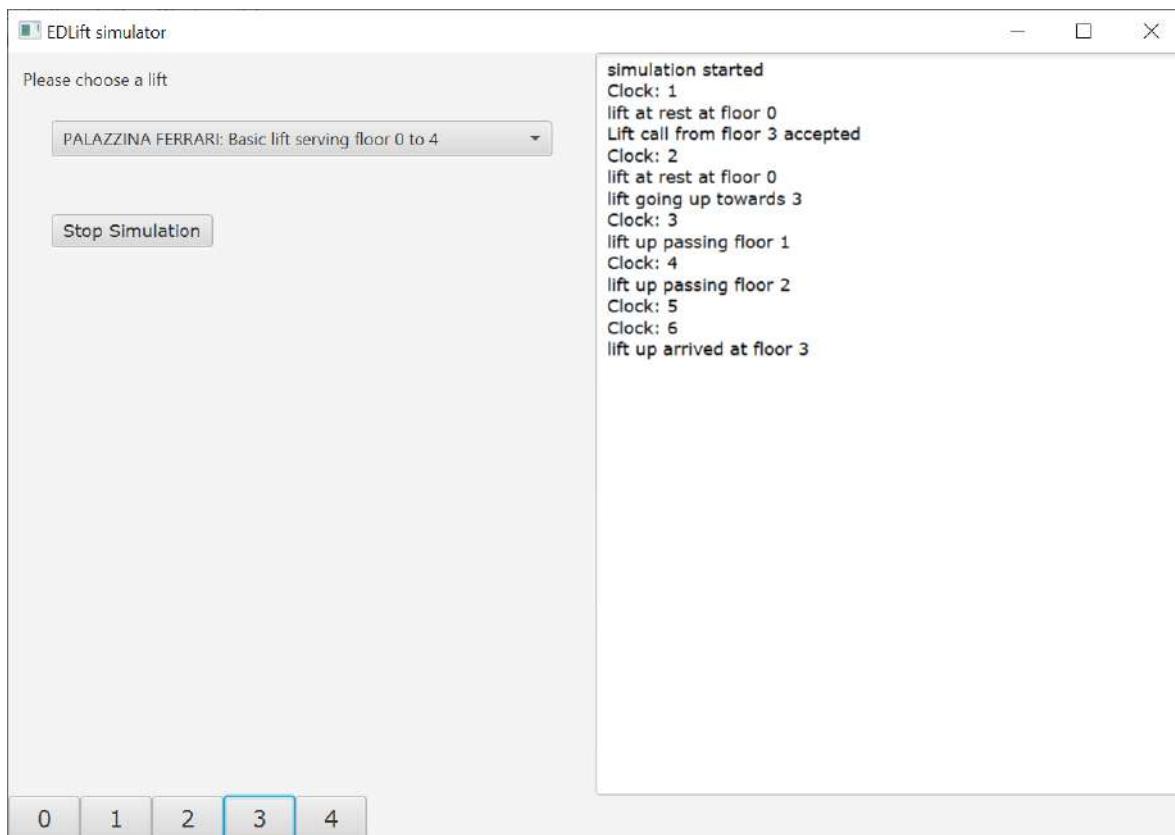


Figura 4

ESAME DI FONDAMENTI DI INFORMATICA T-2 del 22/7/2021

Proff. E. Denti – R. Calegari – A. Molesini

Tempo a disposizione: 3 ore

NOME PROGETTO ECLIPSE: CognomeNome-matricola (es. RossiMario-0000123456)

NOME CARTELLA PROGETTO: CognomeNome-matricola (es. RossiMario-0000123456)

NOME ZIP DA CONSEGNARE: CognomeNome-matricola.zip (es. RossiMario-0000123456.zip)

NOME JAR DA CONSEGNARE: CognomeNome-matricola.jar (es. RossiMario-0000123456.jar)

Si devono consegnare DUE FILE: *I'intero progetto Eclipse e il JAR eseguibile*

Si ricorda che compiti non compilabili o palesemente lontani da 18/30 NON SARANNO CORRETTI e causeranno la verbalizzazione del giudizio “RESPINTO”

È stata richiesta una app per giocare a **Battaglia Navale**, nella versione **solistario**. L'obiettivo è giocare contro il computer, indovinando la posizione delle navi mediante l'uso intelligente delle informazioni fornite.

DESCRIZIONE DEL DOMINIO DEL PROBLEMA

La Battaglia navale in solitario è un gioco di logica e abilità costituito da una *griglia* di 8x8 celle, ciascuna delle quali può contenere o un **elemento di nave** o il **mare**. Inizialmente, quasi tutte le celle della griglia del giocatore sono **vuote**, tranne alcune fornite come base di partenza.

Le navi possono essere di quattro **tipi**:

- *Portaerei* (4 elementi: 2 estremi + 2 centrali)
- *Incrociatori* (3 elementi: 2 estremi + 1 centrale)
- *Cacciatorpedinieri* (2 elementi, entrambi estremi)
- *Sommergibili* (un singolo elemento)

Gli **elementi** che costituiscono le navi possono essere:

- Elementi orizzontali (estremo sinistro, estremo destro)
- Elementi verticali (estremo superiore, estremo inferiore)
- Elemento centrale (un quadrato)
- Elemento singolo (un cerchio = sommersibile)

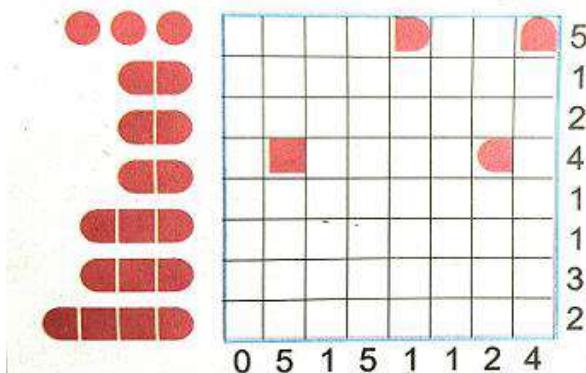
Lo scopo del gioco è capire dove siano le varie navi, sapendo che:

- Il totale di elementi di ogni riga/colonna è riportato a destra/sotto la riga/colonna corrispondente
- Intorno a ogni nave deve esserci del mare: due navi non possono mai toccarsi, neanche in diagonale

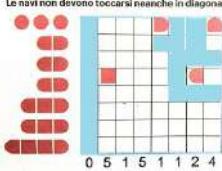
STRATEGIA: il giocatore inizia collocando il mare intorno agli elementi di nave noti e prosegue poi deducendo via via le possibili posizioni degli altri tenendo conto dei totali di riga/colonna forniti.

Per curiosità, sotto viene mostra il corrispondente schema via via risolto.

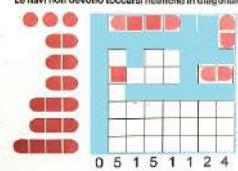
Trovate nello schema la posizione delle navi sotto (portaerei da 4 caselle, incrociatori da 3, torpediniere da 2 e sommergibili da 1), sapendo che i numeri di fianco allo schema indicano la somma di caselle occupate dalle navi nella riga o nella colonna. Le navi non devono toccarsi neanche in diagonale.



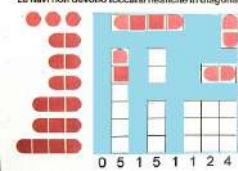
Trovate nello schema la posizione delle navi sotto (portaerei da 4 caselle, incrociatori da 3, torpediniere da 2 e sommergibili da 1), sapendo che i numeri di fianco allo schema indicano la somma di caselle occupate dalle navi nella riga o nella colonna. Le navi non devono toccarsi neanche in diagonale.



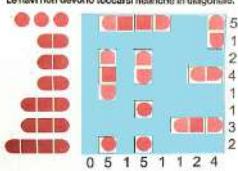
Trovate nello schema la posizione delle navi sotto (portaerei da 4 caselle, incrociatori da 3, torpediniere da 2 e sommergibili da 1), sapendo che i numeri di fianco allo schema indicano la somma di caselle occupate dalle navi nella riga o nella colonna. Le navi non devono toccarsi neanche in diagonale.



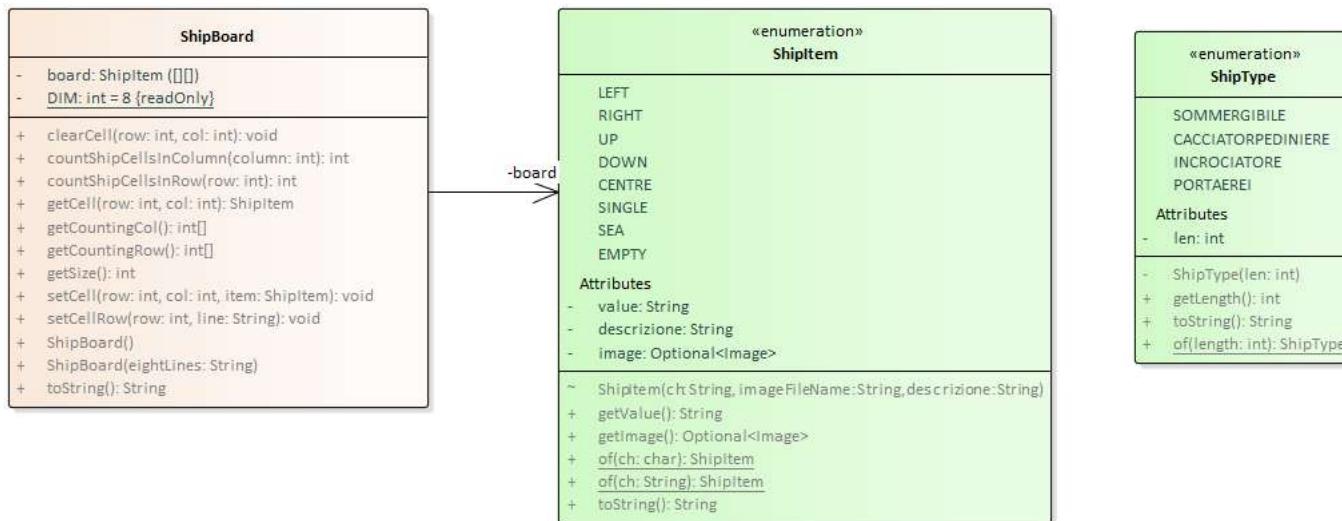
Trovate nello schema la posizione delle navi sotto (portaerei da 4 caselle, incrociatori da 3, torpediniere da 2 e sommergibili da 1), sapendo che i numeri di fianco allo schema indicano la somma di caselle occupate dalle navi nella riga o nella colonna. Le navi non devono toccarsi neanche in diagonale.



Trovate nello schema la posizione delle navi sotto (portaerei da 4 caselle, incrociatori da 3, torpediniere da 2 e sommergibili da 1), sapendo che i numeri di fianco allo schema indicano la somma di caselle occupate dalle navi nella riga o nella colonna. Le navi non devono toccarsi neanche in diagonale.



TEMPO STIMATO PER SVOLGERE L'INTERO COMPITO: 1h40 – 2h10



SEMANTICA:

- L'enumerativo **ShipType** (fornito) definisce i quattro tipi possibili di nave, ciascuno con associata la propria lunghezza recuperabile tramite il metodo `getLength`. Un metodo factory `of` consente di ottenere l'enumerativo “giusto” per una lunghezza data. Un'apposita `toString` completa il tutto.
- L'enumerativo **ShipItem** (fornito) definisce gli otto tipi possibili di elementi, di cui 6 elementi di nave più il mare e il caso della casella vuota. Ognuno è caratterizzato da varie proprietà, in particolare
 - la stringa (singolo carattere) corrispondente, recuperabile tramite il metodo `getValue`
 - l'immagine associata, recuperabile tramite il metodo `getImage`
 - la descrizione testuale corrispondente, restituita da `toString`

Anche in questo caso una coppia di metodi factory `of` consente di ottenere l'enumerativo “giusto” a partire dal carattere dato (uno degli otto possibili).

- la classe **ShipBoard** (fornita parzialmente realizzata ma da completare) rappresenta lo schema di gioco: ne verranno usate due istanze, una per rappresentare la soluzione (immutabile), l'altra per rappresentare la situazione attuale della scacchiera del giocatore (ovviamente modificabile). Per ipotesi, la scacchiera è *sempre* 8x8. La classe mette a disposizione i seguenti metodi:
 - un costruttore* per la scacchiera inizialmente vuota
 - un costruttore* con argomento una stringa di otto righe, corrispondenti al contenuto iniziale della scacchiera; ogni riga è costituita da una sequenza di singoli caratteri separati fra loro da spazi. I caratteri ammessi sono soltanto `<`, `>`, `^`, `v`, `x`, `o` per gli elementi-nave, più `~` per il mare e `#` per denotare la cella vuota. [NB: il carattere `~` si ottiene con ALT+126 nelle tastiere italiane]. Il costruttore, come anche il reader, fa uso del sottostante metodo `setCellRow`.
 - `setCellRow` consente di caricare nello schema un'intera riga di valori, secondo le medesime convenzioni sopra indicate. Il metodo deve controllare i valori di riga/colonna ricevuti, che ovviamente devono essere compresi nel range 0..DIM-1, lanciando **IllegalArgumentException** in caso di non conformità, con adeguata messaggistica.
 - `getCell` restituisce lo **ShipItem** corrispondente all'attuale contenuto della cella relativa agli indici ricevuti come argomento. Come sopra, il metodo deve controllare accuratamente i parametri ricevuti, lanciando **IllegalArgumentException** in caso di non conformità.
 - `getSize` restituisce la dimensione della scacchiera

- `clearCell` imposta a ***ShipItem.EMPTY*** il contenuto della cella relativa agli indici di riga e colonna specificati, che ovviamente devono essere controllati, lanciando ***IllegalArgumentException*** in caso di non conformità.
- `setCell` imposta il contenuto della cella relativa agli indici di riga e colonna specificati al valore di ***ShipItem*** specificato; anche in questo caso ovviamente gli indici devono essere controllati come sopra, lanciando ***IllegalArgumentException*** in caso di non conformità.
- `countShipCellsInRow` e `countShipCellsInColumn` restituiscono rispettivamente il numero di caselle di tipo nave (quindi, *non mare e non vuote*) nella riga o colonna specificata.
- `getCountingRow` e `getCountingCol` restituiscono rispettivamente l'array di interi relativo ai "suggerimenti" verticali (riga extra da posizionare sotto la scacchiera) / orizzontali (colonna extra da posizionare alla destra della scacchiera).
- `toString` restituisce una stringa con la struttura della scacchiera (in righe) intesa come sequenza di valori che etichettano le varie caselle, utilizzando i caratteri associati a ogni ***ShipItem***.

IMPORTANTE: data la presenza di una ***Image*** JavaFX in ***ShipItem***, e quindi di riflesso anche in ***ShipBoard***, per far girare i test è indispensabile aggiungere i soliti argomenti alla run configuration di ogni test

```
-ea --module-path .....\\javafx-sdk-15.0.1\\lib --add-modules javafx.controls
```

Persistenza (*battleship.persistence*)

[TEMPO STIMATO: 35-45 minuti] (punti 10)

Ci sono due file di testo:

- **battlefield.txt** contiene la soluzione, ossia la disposizione di navi e mare che il giocatore deve indovinare
- **initialfield.txt** contiene la configurazione iniziale della scacchiera del giocatore.

Entrambi sono formattati secondo lo stesso schema, ovvero con esattamente otto righe costituite ciascuna da una sequenza di otto singoli caratteri separati fra loro da spazi. I caratteri ammessi sono soltanto <, >, ^, v, x, o per gli elementi-nave, più ~ per il mare e # per denotare la cella vuota. Per ipotesi:

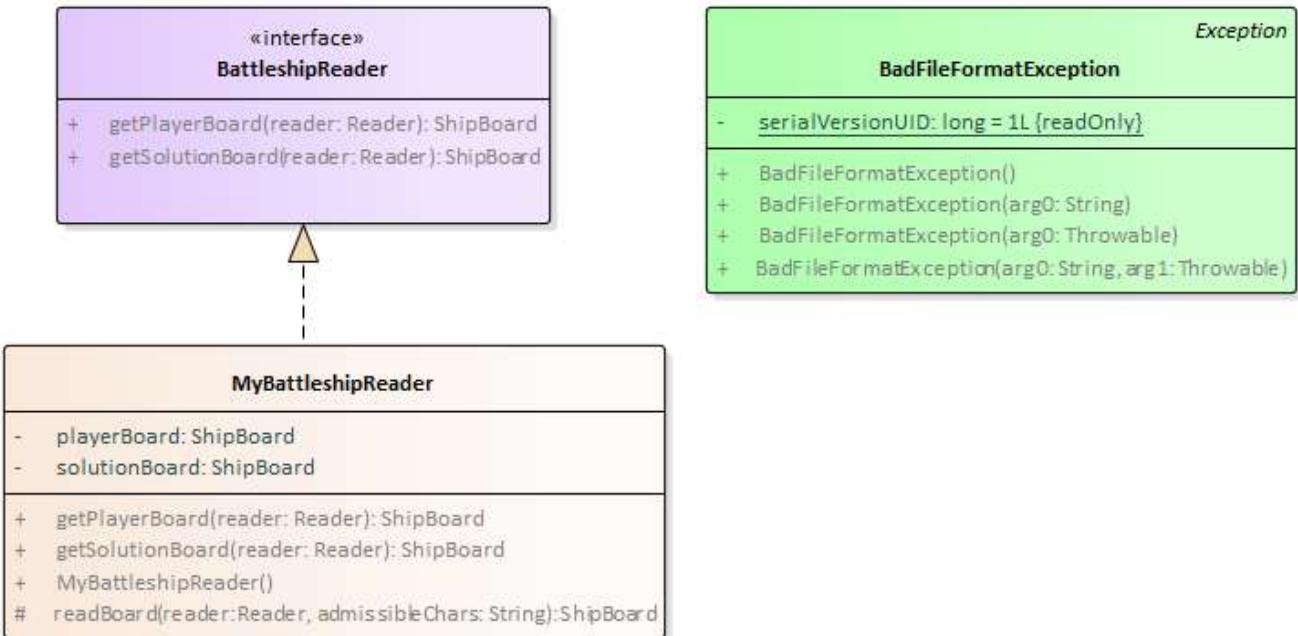
- nel file **battlefield.txt** non vi sono celle vuote (quindi, non è mai presente il carattere #)
- nel file **initialfield.txt** non vi sono solo celle di mare (quindi, non è mai presente il carattere ~).

Esempi:

battlefield.txt	initialfield.txt
~ < x x > ~ ~ ^	# # # # > # # ^
~ ~ ~ ~ ~ ~ ~ v	# # # # # # #
~ ^ ~ ^ ~ ~ ~ ~	# # # # # # #
~ x ~ v ~ ~ < >	# x # # # # < #
~ v ~ ~ ~ ~ ~ ~	# # # # # # #
~ ~ ~ o ~ ~ ~ ~	# # # # # # #
~ ~ ~ ~ ~ < x >	# # # # # # #
~ o ~ o ~ ~ ~ ~	# # # # # # #

Poiché la struttura dei due file è identica, l'architettura prevede un unico reader con due distinti metodi di lettura, che costituiscono due entry point per lo stesso metodo di lettura fisica, con diverso set di caratteri ammissibili.

Il diagramma UML è illustrato alla pagina seguente.



SEMANTICA:

- L'interfaccia **BattleShipReader** (fornita) dichiara i due metodi `getSolutionBoard` e `getPlayerBoard` che restituiscono rispettivamente la scacchiera-soluzione e la scacchiera iniziale lette dal file; al fine di evitare letture ripetute, la prima volta che essi vengono invocati memorizzano nello stato del reader la **ShipBoard** letta, che viene poi restituita a ogni invocazione successiva dello stesso metodo.
- La classe **MyBattleShipReader (da realizzare)** implementa **BattleShipReader**
 - costruttore* di default che si limita a inizializzare lo stato interno, senza ancora effettuare lettura
 - `getSolutionBoard` e `getPlayerBoard` che si appoggiano al metodo protetto `readBoard`, il cui primo argomento è un **Reader** già aperto, il secondo è una stringa che specifica i caratteri ammissibili per quella scacchiera; come già anticipato, ognuno di questi metodi effettua realmente la lettura solo la prima volta che viene invocato, memorizzando il risultato nello stato interno del reader, così da poterlo facilmente restituire alle invocazioni successive senza dover rifare alcuna lettura.

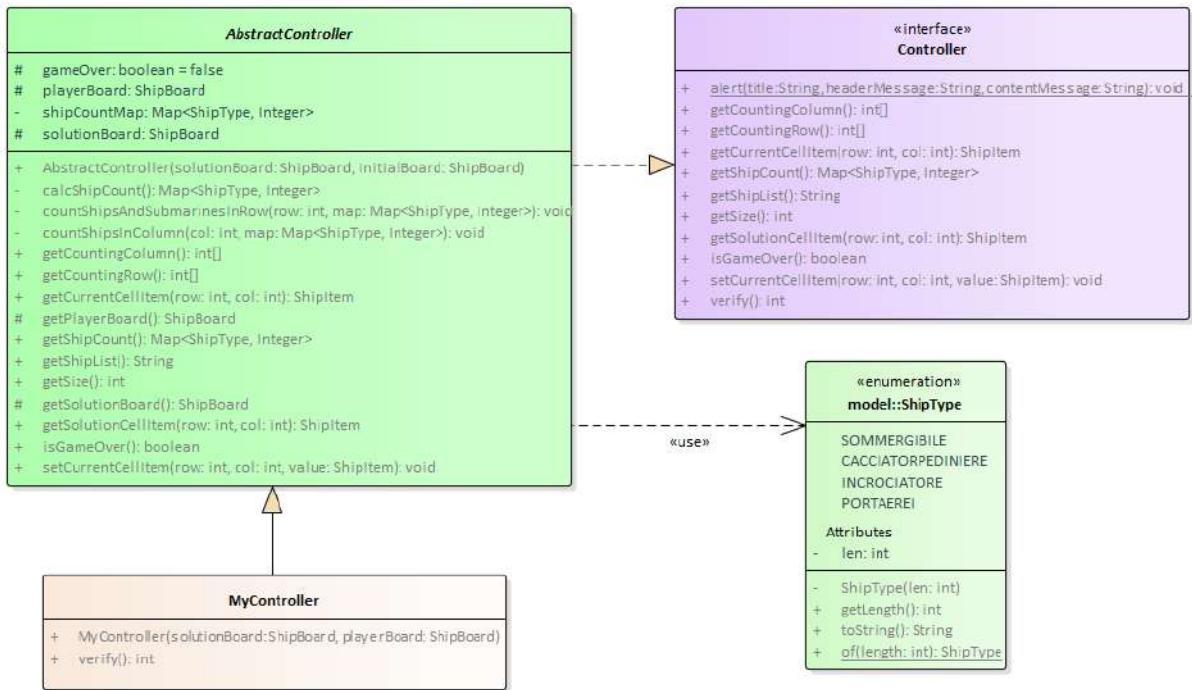
Parte 2

(punti: 13)

Controller (package battleship.controller)

[TEMPO STIMATO: 15-20 minuti] (punti: 5)

Il Controller è organizzato secondo il diagramma UML in figura.



SEMANTICA:

- a) L'interfaccia **Controller** (fornita) dichiara dieci metodi, otto dei quali costituiscono semplici entry point ad omonimi metodi di **ShipBoard**:

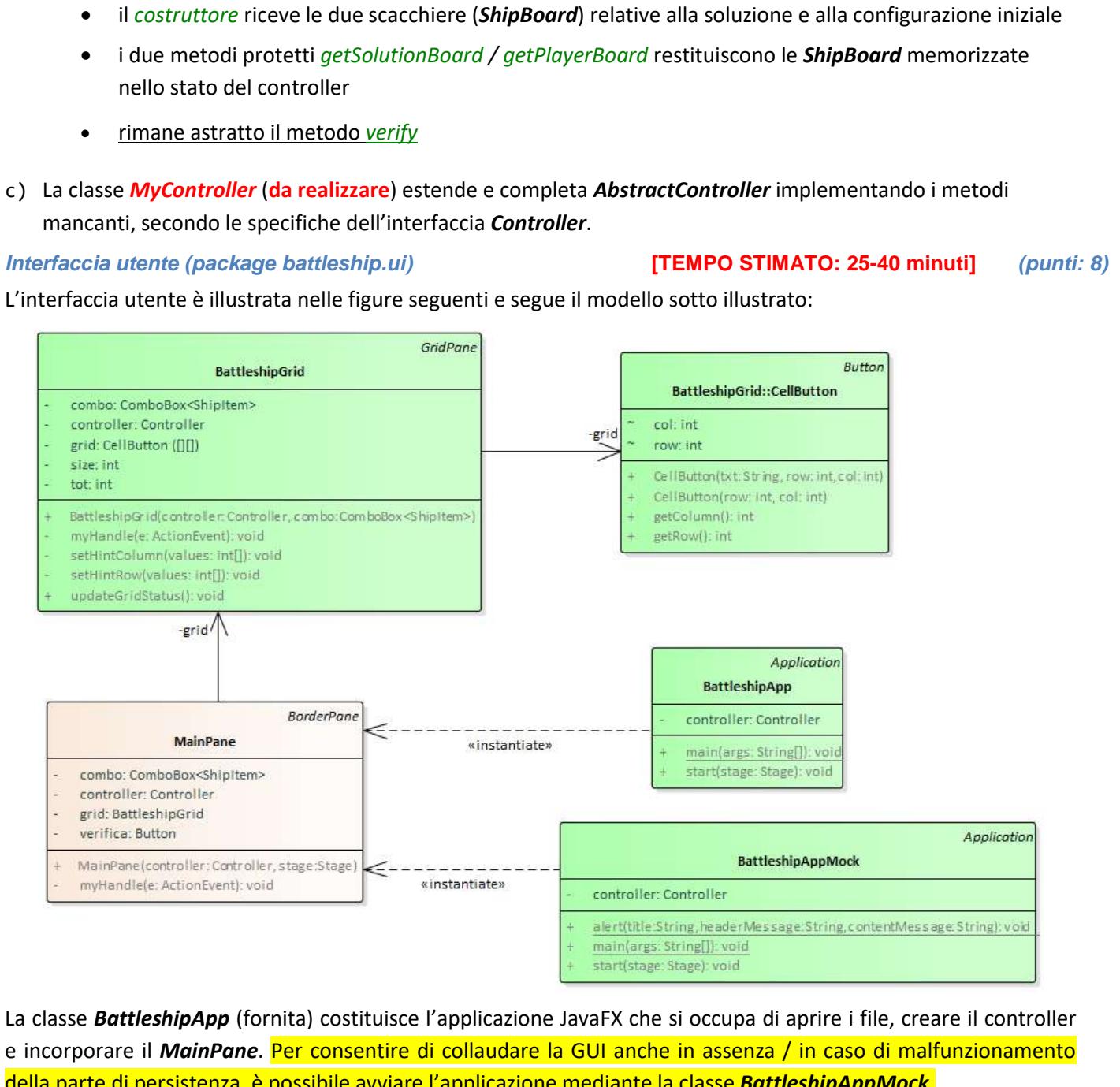
- **getSize** restituisce la dimensione della scacchiera
- **getCountingRow / getCountingCol** si rimappano sui quasi-omonimi metodi di **ShipBoard**
- **getCurrentCellItem / setCurrentCellItem** si rimappano sui quasi-omonimi metodi di **ShipBoard** relativamente alla scacchiera giocatore
- **getSolutionCellItem** si rimappano sul quasi-omonimo metodo di **ShipBoard** relativamente alla scacchiera soluzione
- **isGameOver** restituisce lo stato del controller relativamente all'eventuale raggiungimento della fine del gioco (nessuna cella vuota, tutte le celle della scacchiera giocatore identiche a quelle della soluzione)
- **getShipList** restituisce una stringa che elenca numero e tipo della navi presenti nella soluzione

Sono invece peculiari del controller i due metodi:

- **verify**, che confronta lo stato attuale della scacchiera giocatore con la soluzione, contando e restituendo il numero di celle non vuote diverse (ossia, sbagliate): contemporaneamente, aggiorna lo stato interno del controller relativamente al campo-dati **gameOver**, che diviene *true* solo se nella scacchiera giocatore non vi è più alcuna cella vuota e tutte le celle sono identiche a quelle della soluzione.
- **getShipCount**, che restituisce una mappa <**ShipType**, **Integer**> che conta quante navi ci sono nella scacchiera-soluzione per ogni tipo di nave

NB: il Controller contiene anche il **metodo statico ausiliario alert**, utile per mostrare avvisi all'utente.

- b) La classe **AbstractController** (fornita) implementa quasi totalmente tale interfaccia



SEMANTICA:

- La classe **BattleshipGrid** (fornita) fornisce un componente pronto per l’uso che mostra e gestisce la griglia di pulsanti che costituiscono la GUI della scacchiera del giocatore, con le due colonne/righe di ausilio a destra e sotto. In particolare:
 - il *costruttore* riceve il **Controller** e un riferimento alla **ComboBox** del **MainPane**, utile per estrarre l’elemento scelto dall’utente per poi impostare al giusto **ShipItem** la casella premuta;
 - il metodo *updateGridStatus* aggiorna la visualizzazione e lo stato interno della griglia: va chiamato dopo ogni modifica che implica una gestione di eventi del **MainPane**
- La classe **MainPane** (da realizzare) estende **BorderPane** e prevede:
 - nel lato sinistro, in verticale, prima una **ComboBox** popolata con gli **ShipItem**, poi alcune **Label** che riportano l’elenco delle navi, indi un pulsante VERIFICA che scatena il confronto fra l’attuale scacchiera giocatore e la soluzione retrostante (che non viene però mai mostrata).
 - nella parte centrale, una **BattleshipGrid**.

Inizialmente, il pannello mostra la configurazione iniziale (Fig. 1). Per inserire un elemento nella griglia, l'utente deve prima selezionarlo dalla combo, poi premere il pulsante-cellula corrispondente (Fig. 2).

Proseguendo nel gioco, in qualunque momento l'utente può premere il pulsante VERIFICA che scatena, tramite il metodo `verify` del controller, la verifica sull'eventuale presenza di celle errate, mostrandone il numero (Figg. 3 e 4).

Nel caso vi siano celle errate, esse NON vengono immediatamente resettate a livello grafico: la gestione dell'evento deve invece far comparire un apposito dialogo (Figg. 3 e 4) che riporta l'esito della verifica. SOLO DOPO che l'utente preme il tasto OK nel dialogo si procede ad aggiornare lo stato della griglia, "sbiancando" le celle errate (Fig. 5). In questo modo è impossibile proseguire nel gioco con configurazioni errate: il sistema convalida e accetta solo configurazioni corrette, guidando via via verso la soluzione.

Il gioco termina quando, dopo aver riempito tutte le caselle, la verifica finale dà esito positivo (Fig. 6): anche in tal caso comunque la scacchiera giocatore resta attiva, permettendo eventualmente all'utente di continuare a modificare celle per divertimento.

La gestione dell'evento relativo al pulsante VERIFICA deve:

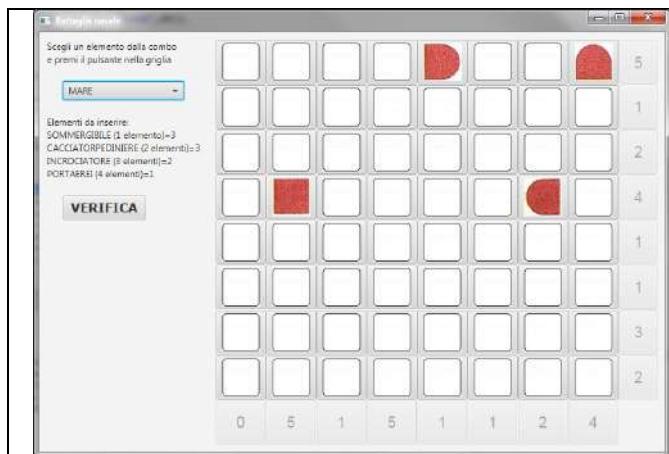
- effettuare la verifica della situazione, tramite il metodo `verify` del controller
- mostrare la finestra di dialogo (utile il metodo statico `alert` del controller) con idonea messaggistica adeguata alla specifica situazione
- aggiornare lo stato della griglia.

Cose da ricordare

- salva costantemente il tuo lavoro: l'informatica a volte può essere "subdolamente ostile" ..
- in particolare: se ora compila e stai per fare modifiche, salva la versione attuale (non si sa mai)

Checklist di consegna

- Hai fatto un **JAR eseguibile**, che contenga cioè l'indicazione del main?
- Hai controllato che **si compili e ci sia tutto?** [NB: non includere il PDF del testo]
- Hai **rinominato IL PROGETTO**, lo ZIP e il JAR esattamente come richiesto?
- Hai **chiamato** la cartella del progetto esattamente come richiesto?
- **Hai fatto un unico file ZIP (NON .7z, rar o altri formati) contenente l'intero progetto?**
In particolare, ti sei assicurato di aver incluso tutti i file .java (e non solo i .class)?
- **Hai consegnato DUE file distinti, ossia lo ZIP col progetto e il JAR eseguibile?**
- Su EOL, hai premuto il tasto "CONFERMA" per inviare il tuo elaborato?



MARE

Elementi da inserire:
SOMMERGIBILE (1 elemento)=3
CACCIA TORPEDINIERE (2 elementi)=3
INCROCIATORE (3 elementi)=2
PORTAEREA (4 elementi)=1

VERIFICA

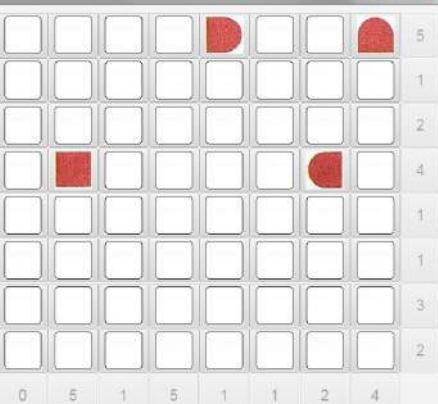
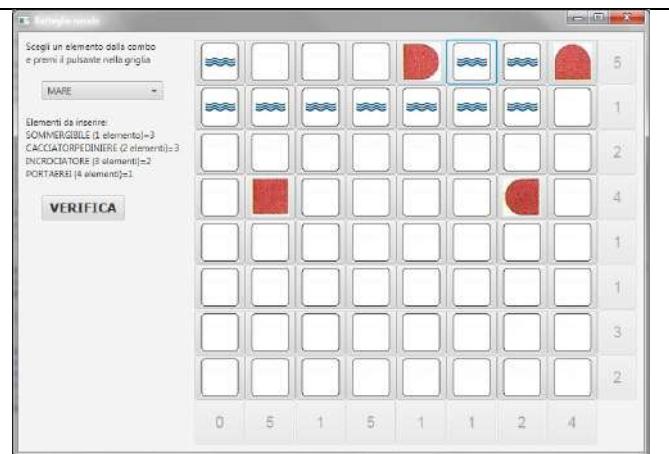


Fig.1



MARE

Elementi da inserire:
SOMMERGIBILE (1 elemento)=3
CACCIA TORPEDINIERE (2 elementi)=3
INCROCIATORE (3 elementi)=2
PORTAEREA (4 elementi)=1

VERIFICA

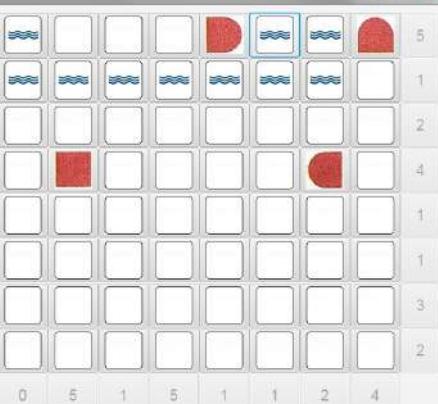
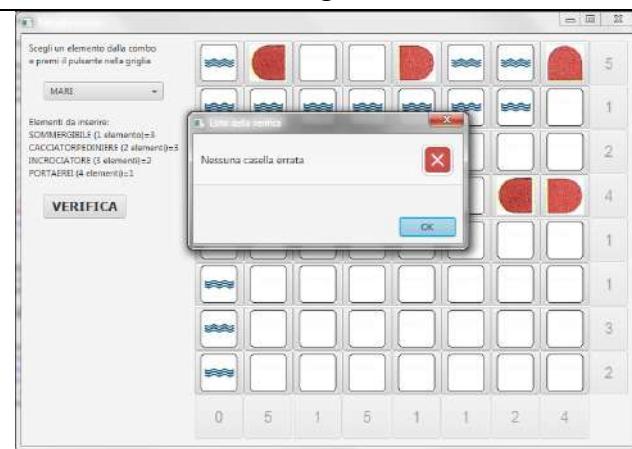


Fig.2



MARE

Elementi da inserire:
SOMMERGIBILE (1 elemento)=3
CACCIA TORPEDINIERE (2 elementi)=3
INCROCIATORE (3 elementi)=2
PORTAEREA (4 elementi)=1

VERIFICA

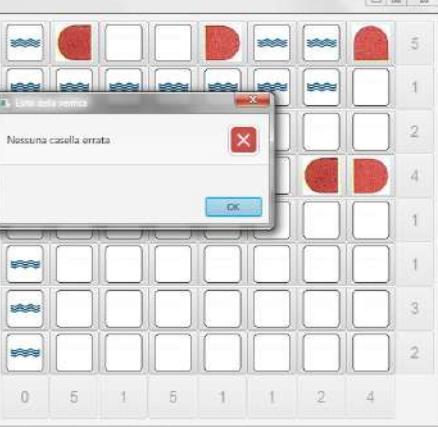
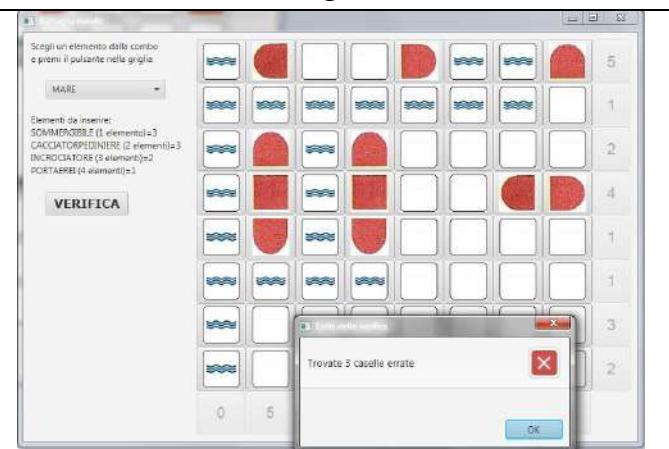


Fig.3



MARE

Elementi da inserire:
SOMMERGIBILE (1 elemento)=3
CACCIA TORPEDINIERE (2 elementi)=3
INCROCIATORE (3 elementi)=2
PORTAEREA (4 elementi)=1

VERIFICA

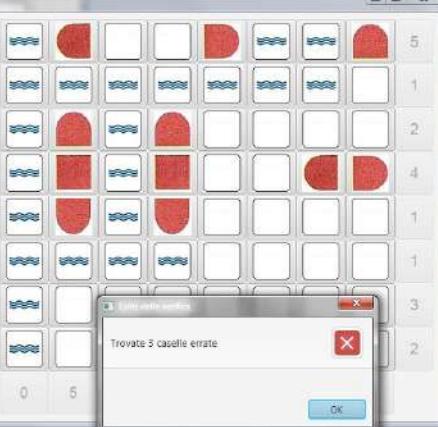
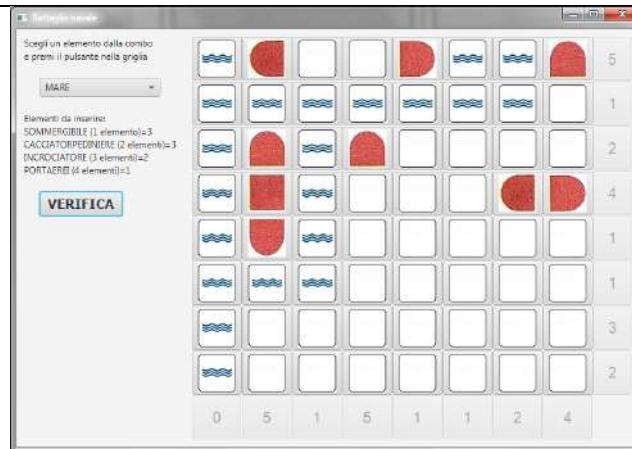


Fig.4



MARE

Elementi da inserire:
SOMMERGIBILE (1 elemento)=3
CACCIA TORPEDINIERE (2 elementi)=3
INCROCIATORE (3 elementi)=2
PORTAEREA (4 elementi)=1

VERIFICA

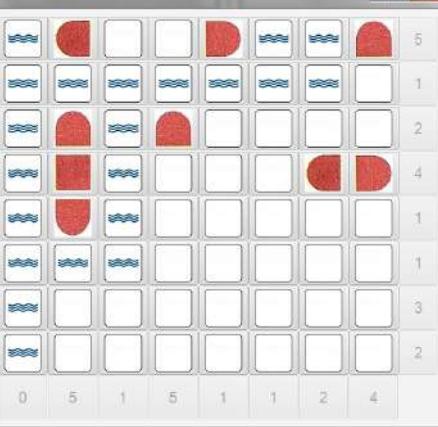
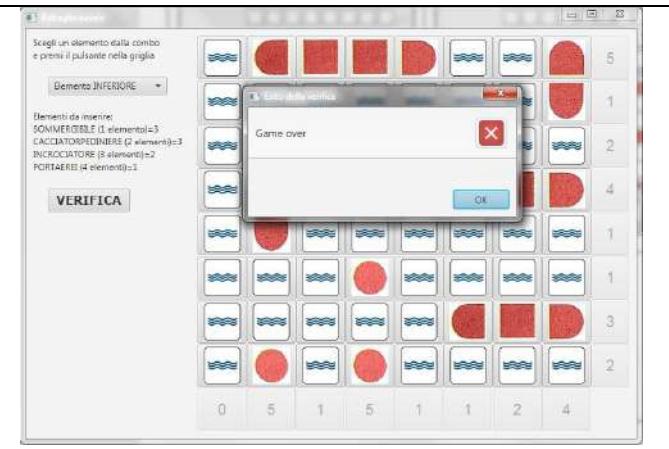


Fig.5



Mare inferiore

Elementi da inserire:
SOMMERGIBILE (1 elemento)=3
CACCIA TORPEDINIERE (2 elementi)=3
INCROCIATORE (3 elementi)=2
PORTAEREA (4 elementi)=1

VERIFICA

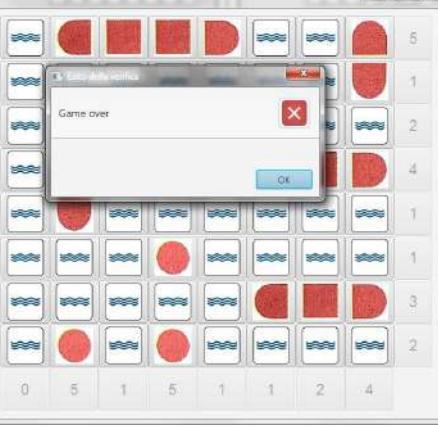


Fig.6

ESAME DI FONDAMENTI DI INFORMATICA T-2 del 6/07/2021

Proff. E. Denti – R. Calegari – A. Molesini

Tempo a disposizione: 3 ore

NOME PROGETTO ECLIPSE: CognomeNome-matricola (es. RossiMario-0000123456)

NOME CARTELLA PROGETTO: CognomeNome-matricola (es. RossiMario-0000123456)

NOME ZIP DA CONSEGNARE: CognomeNome-matricola.zip (es. RossiMario-0000123456.zip)

NOME JAR DA CONSEGNARE: CognomeNome-matricola.jar (es. RossiMario-0000123456.jar)

Si devono consegnare DUE FILE: *I'intero progetto Eclipse e il JAR eseguibile*

Si ricorda che compiti non compilabili o palesemente lontani da 18/30 NON SARANNO CORRETTI e causeranno la verbalizzazione del giudizio “RESPINTO”

Le **Rupestri Ferrovie di Dentinia** (RFD), che gestiscono una antica rete ferroviaria a vapore fra alcune città, hanno richiesto un'applicazione per la ricerca dei possibili percorsi fra stazioni della loro rete: i percorsi devono essere o diretti (senza cambi) o con al più un cambio.



DESCRIZIONE DEL DOMINIO DEL PROBLEMA

Ogni *linea ferroviaria* è descritta da una sequenza di *punti di interesse*, caratterizzati ciascuno dal nome del luogo (una stringa che può contenere spazi) e dalla *progressiva chilometrica* (ossia, la distanza dal capolinea iniziale): per antica consuetudine, quest'ultima – riportata anche sui caselli e sulle stazioni – è espressa nella forma *chilometri+metri*, dove i *metri* sono sempre espressi su tre cifre, mentre i *chilometri* sono espressi da un numero di almeno una cifra.



ESEMPIO (foto): la stazione di Lodi si trova alla progressiva chilometrica 183+802 della linea Bologna-Milano, il cui capolinea iniziale è Bologna Centrale (0+000).

I percorsi fra città appartenenti alla stessa linea (percorsi *diretti*) hanno una lunghezza facilmente desumibile dalla differenza fra le corrispondenti progressive chilometriche.

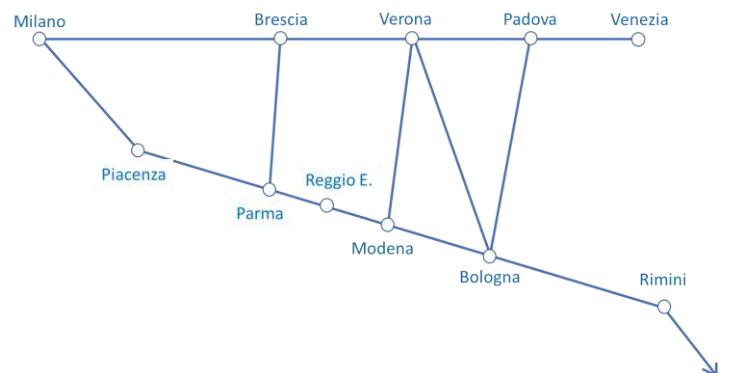
ESEMPIO: la lunghezza del percorso fra Lodi (progressiva 183+802) e Modena (progressiva 36+932) è di 146,87 km.

Alcune stazioni, in cui si intersecano due o più linee, costituiscono *punti di interscambio*: ciò consente di offrire ai viaggiatori soluzioni di viaggio *con un cambio intermedio*, unendo due *segmenti* di linee diverse aventi in comune il nodo di interscambio: in tal caso la lunghezza del percorso è pari alla somma delle lunghezze dei segmenti componenti.

ESEMPIO: il percorso fra Lodi (progressiva 183+802 della linea Bologna-Milano) e Ancona (progressiva 203+996 della linea Bologna-Lecce) è di 387,798 km e comprende due segmenti (LO-BO e BO-AN).

Se la rete comprende più linee, possono essere possibili *più percorsi* fra le medesime città, con o senza cambi.

ESEMPIO: nella rete a lato, fra Reggio Emilia e Verona sono possibili due percorsi, uno via Modena, l'altro via Bologna.



La rete delle **Rupestri Ferrovie di Dentinia**, costituita da sette *linee*, è illustrata sopra: ogni linea è descritta in un singolo file di testo, il cui formato è riportato più oltre.

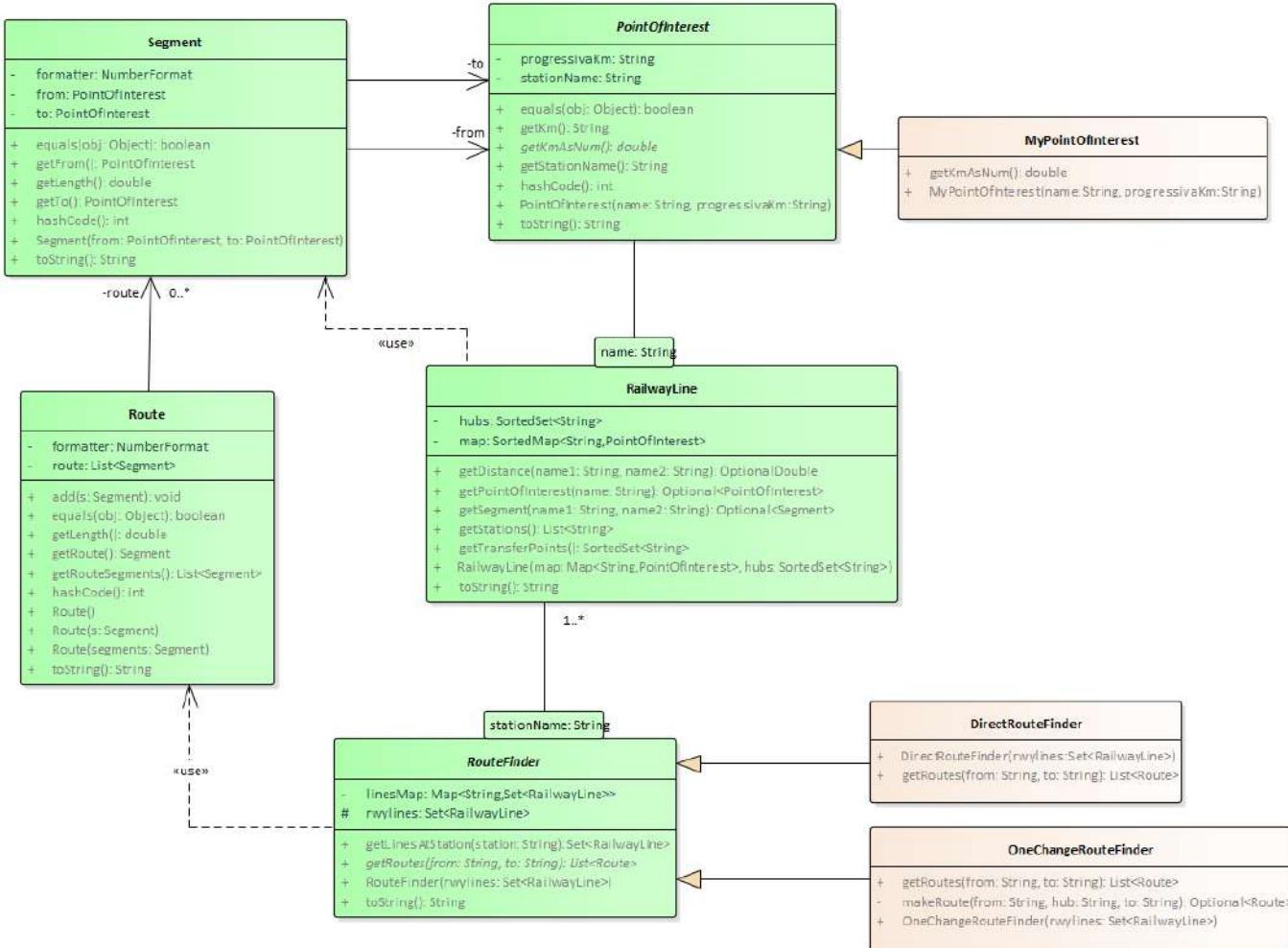
TEMPO STIMATO PER SVOLGERE L'INTERO COMPITO: 1h45 – 2h20

Parte 1

(punti: 23)

Modello dei dati (package rfd.model)

[TEMPO STIMATO: 1h10-1h30] (punti: 18)



SEMANTICA:

- la classe **astratta** **PointOfInterest** (fornita) rappresenta un punto di interesse, caratterizzato da nome e progressiva chilometrica, *senza fare ipotesi sul formato di quest'ultima*: il metodo **getKmAsNum**, che riporta sotto forma di numero il valore della progressiva chilometrica, rimane perciò astratto
- la classe **MyPointOfInterest (da realizzare)** concretizza la precedente assumendo per la progressiva chilometrica **il formato chilometri+metri**, dove i chilometri sono *un intero di almeno una cifra*, mentre i metri sono *sempre espressi su esattamente tre cifre*: *se questo formato è violato, il costruttore deve lanciare **IllegalArgumentException*** con apposito messaggio d'errore. Ovviamente, il metodo **getKmAsNum** segue anch'esso questa convenzione.
- la classe **Segment** (fornita) rappresenta un segmento di linea compreso fra due **PointOfInterest** **distinti** (anche non adiacenti, ossia anche con ulteriori **PointOfInterest** intermedi): i metodi consentono di recuperare gli elementi caratterizzanti, inclusa la lunghezza del tratto come numero reale, nonché di produrre un'opportuna stringa descrittiva. Sono presenti anche **equals** e **hashcode**.
- la classe **RailwayLine** (fornita) rappresenta una linea ferroviaria intesa come insieme di **PointOfInterest**, che il costruttore si aspetta di ricevere sotto forma di mappa indicizzata per nome del punto di interesse. Il secondo argomento del costruttore è l'insieme ordinato dei *nomi* delle stazioni della linea che possono fungere da punti di interscambio. La classe offre svariati metodi per:
 - recuperare la lista dei nomi delle stazioni (metodo **getStations**) o dei punti di interscambio (metodo **getTransferPoints**)
 - recuperare un **PointOfInterest**, se esiste, dato il suo nome (metodo **getPointOfInterest**)

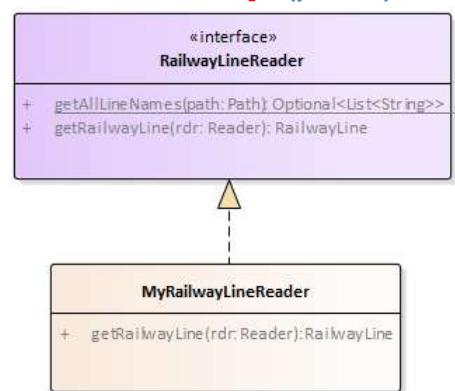
- calcolare la distanza fra due **PointOfInterest** distinti, se esistono sulla linea (metodo `getDistance`)
 - recuperare il **Segment**, se esiste, corrispondente alla tratta compresa fra i due **PointOfInterest** distinti dati (metodo `getSegment`)
 - emettere una stringa descrittiva (metodo `toString`)
- e) la classe **Route** (fornita) rappresenta un *percorso* per un viaggiatore, inteso come *sequenza di Segmenti su linee diverse*. I tre costruttori consentono di costruire la **Route** in tre situazioni tipiche (inizialmente vuota, inizialmente con un solo segmento, o a partire da un numero variabile di segmenti): successivamente è comunque possibile aggiungere via via altri segmenti *in coda* alla sequenza, tramite il metodo `add`. Opportuni accessori consentono di recuperare i vari elementi: anche in questo caso sono presenti `equals`, `hashCode` e `toString`.
- f) La classe astratta **RouteFinder** (fornita) rappresenta l'entità in grado di cercare percorsi fra due città date: il costruttore riceve a tal fine l'insieme di **RailwayLine** che rappresenta la rete societaria. Il metodo (astratto) `getRoutes` cerca l'insieme dei percorsi fra due stazioni date, che devono essere non-null: altrimenti, per ipotesi deve lanciare **IllegalArgumentException** con apposito messaggio. A differenza del precedente, il metodo (concreto) `getLinesAtStation` restituisce l'insieme – eventualmente vuoto - delle **RailwayLine** che servono una data stazione, senza mai lanciare eccezioni.
- g) la classe **DirectRouteFinder (da realizzare)** concretizza **RouteFinder** implementando il metodo **(punti: 5)** `getRoutes` in modo che cerchi i percorsi diretti fra le due città date. **[TEMPO STIMATO: 15-20 minuti]**
- h) la classe **OneChangeRouteFinder (da realizzare)** concretizza **RouteFinder** implementando **(punti: 9)** `getRoutes` in modo che cerchi i percorsi indiretti con un solo cambio (T) fra le due città date A e B: **[TEMPO STIMATO: 45-55 minuti]**
- per ogni linea che serve la stazione di partenza A (**suggerimento: usare l'apposito metodo**) *che non serve anche la stazione di arrivo B* (altrimenti, sarebbe un percorso diretto!)
 - recupera l'insieme dei suoi punti di interscambio (**suggerimento: usare l'apposito metodo**) e, per ciascuno di essi, cerca se esiste un percorso diretto fra tale punto (T) e la destinazione finale (B) **SUGGERIMENTO: utilizzare un DirectRouteFinder per cercare tale (unico) percorso diretto**
 - se tale percorso diretto T-B esiste, recupera l'altro percorso diretto, che esiste certamente, fra la stazione di partenza (A) e il punto di interscambio considerato (T)
 - confeziona quindi una nuova **Route** costituita dai segmenti A-T e T-B, che costituiscono per definizione i primi e unici segmenti delle due sotto-**Route** appena trovate
 - aggiunge la nuova **Route** così sintetizzata all'insieme delle **Route** da restituire.

Persistenza (rfd.persistence)

[TEMPO STIMATO: 20-30 minuti] (punti 5)

Sono presenti tanti file di testo con estensione “.txt” quante le linee ferroviarie, tutti formattati secondo lo stesso schema. Ogni riga contiene nell'ordine *la progressiva chilometrica*, nella forma *chilometri+metri*, poi *una o più tabulazioni*, indi *il nome della stazione* (che può contenere spazi o qualunque altro carattere). Le stazioni che fungono da interscambio con altre linee sono identificabili dal simbolo “+” in coda al nome della stazione. Come anticipato nel “Dominio del Problema”, la progressiva chilometrica prevede per i *chilometri* un valore intero espresso *fra una e tre cifre*, per i *metri* un valore intero espresso *sempre su esattamente tre cifre*. Ad esempio:

216+176	Milano Centrale+
212+397	Milano Lambrate+
208+751	Milano Rogoredo+
...	
36+932	Modena+
25+008	Castelfranco Emilia



17+130	Samoggia
12+735	Anzola dell'Emilia
0+000	Bologna Centrale+

SEMANTICA:

- a) L'interfaccia **RailwayLineReader** (fornita) dichiara il metodo `getRailwayLine`, che legge da un Reader (ricevuto come argomento) i dati di una singola linea ferroviaria, restituendo la corrispondente **RailwayLine**. L'interfaccia contiene altresì il metodo statico `getAllLineNames`, che restituisce la lista dei nomi di file di tipo ".txt" che descrivono le linee ferroviarie contenuti nella cartella passata come argomento. Tale metodo è invocato automaticamente dal main dell'applicazione (vedere Parte 2).
- b) La classe **MyRailwayLineReader** (da realizzare) implementa **RailwayLineReader**: non prevede costruttori, si limita a implementare il metodo `getRailwayLine` come sopra specificato. In caso di problemi di I/O deve essere propagata l'opportuna `IOException`, mentre in caso di Reader nullo o altri problemi di formato dei file deve essere lanciata una opportuna `IllegalArgumentException`, il cui messaggio dettagli l'accaduto. In particolare, il reader deve verificare: 1) che ogni riga sia composta esattamente di 2 elementi; 2) che il nome della stazione non inizi con cifra numerica; 3) che, nel caso di stazione di interscambio, il "+" finale segua il nome della stazione senza altri caratteri intermedi e sia l'ultimo carattere della riga.

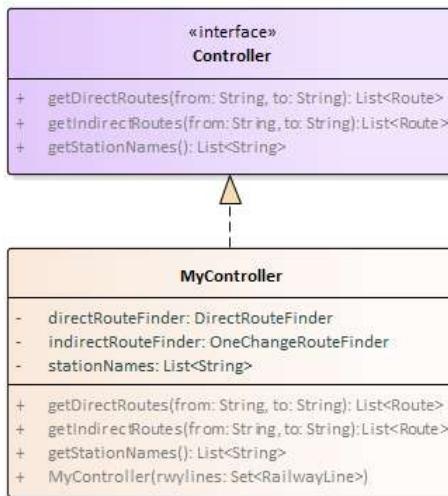
Parte 2)

[TEMPO STIMATO: 15-20 minuti] (punti: 7)

Controller (rfd.controller)

(punti 0)

Il Controller (fornito) è organizzato secondo il diagramma UML in figura.

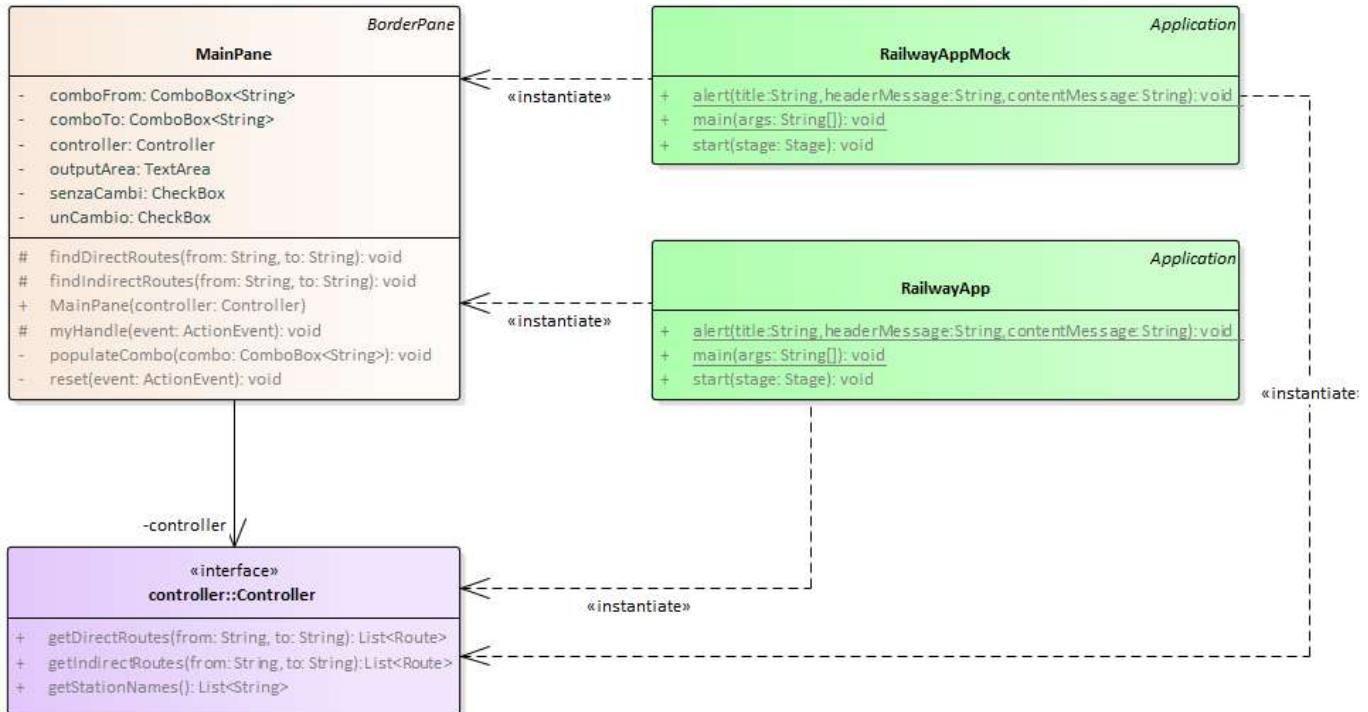


SEMANTICA:

- a) L'interfaccia **Controller** (fornita) dichiara i metodi `getStationNames`, `getDirectRoutes` e `getIndirectRoutes`.
- b) La classe **MyController** (fornita) implementa tale interfaccia, fornendo:
- il costruttore che, dall'elenco delle linee societarie, estrae l'elenco dei nomi delle stazioni e crea internamente i finder necessari;
 - implementa i tre metodi delegando, nel caso delle ricerche, il lavoro ai rispettivi *finder*.

NB: la lista dei nomi di stazione restituita da `getStationNames` è già ordinata alfabeticamente.

L'interfaccia utente è illustrata nelle figure seguenti e segue il modello sotto illustrato:



La classe **RailwayApp** (fornita) costituisce l'applicazione JavaFX che si occupa di aprire i file, creare il controller e incorporare il **MainPane**. Per consentire di collaudare la GUI anche in assenza / in caso di malfunzionamento della parte di persistenza, è possibile avviare l'applicazione mediante la classe **RailwayAppMock**.

Entrambe le classi contengono anche il **metodo statico ausiliario alert**, utile per mostrare avvisi all'utente.

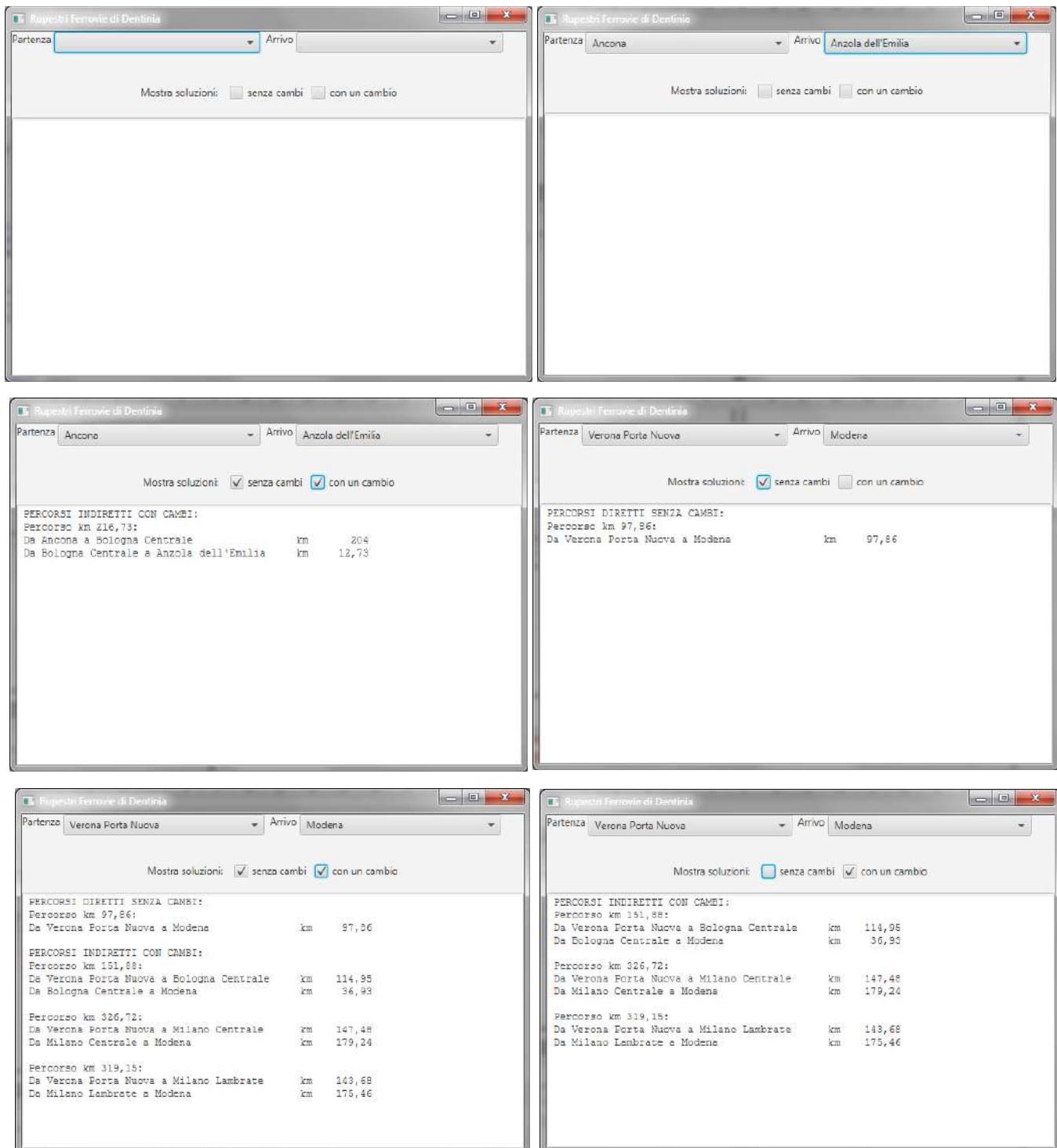
Il **MainPane** è fornito *parzialmente realizzato*: è presente la parte strutturale, mentre manca la parte di popolamento **combo** e gestione degli eventi.

La classe **MainPane** (da completare) estende **BorderPane** e prevede:

- 1) in alto, due **ComboBox** da popolare con l'*elenco alfabetico ordinato di tutte le stazioni di tutte le linee*
- 2) sotto, due **Checkbox** per scegliere la ricerca di percorsi diretti e/o con un cambio
- 3) in basso, una **TextArea** che mostra i percorsi (**Route**) trovati, utilizzando un font non proporzionale tipo Courier New (o analogo) 12 punti

La **parte da completare** comprende l'uso e/o l'implementazione dei seguenti metodi:

- 1) **populateCombo**, che popola la combo con l'*elenco alfabetico ordinato di tutte le stazioni*
- 2) **myHandle**, da chiamare in risposta all'evento di pressione di entrambe le **Checkbox**: si appoggia operativamente ai due metodi ausiliari **findDirectRoutes** e **findIndirectRoutes**.
 - NB: se ci sono sia percorsi diretti che indiretti, devono essere mostrati prima quelli *diretti*
- 3) **findDirectRoutes** e **findIndirectRoutes**, che gestiscono concretamente il caso di percorsi rispettivamente senza cambi / con un cambio, emettendo frasi costruite secondo le seguenti regole (v. figure):
 - prima di tutti i percorsi diretti (risp. indiretti), apposita frase maiuscola di presentazione seguita da un unico *newline*, purché tali percorsi esistano: altrimenti non deve essere visualizzato nulla
 - due *newline* dopo ogni percorso, così che fra i vari percorsi ci sia una riga vuota



Cose da ricordare

- salva costantemente il tuo lavoro: l'informatica a volte può essere "subdolamente ostile" ..
- se ora compila e stai per fare modifiche, salva la versione attuale (non si sa mai)

Checklist di consegna

- Hai fatto un **JAR eseguibile**, che contenga cioè l'indicazione del main?
- Hai controllato che **si compil** e **ci sia tutto**? [NB: non includere il PDF del testo]
- Hai **rinominato IL PROGETTO**, lo ZIP e il JAR esattamente come richiesto?
- Hai **chiamato** la cartella del progetto esattamente come richiesto?
- **Hai fatto un unico file ZIP (NON .7z, rar o altri formati) contenente l'intero progetto?**
In particolare, ti sei assicurato di aver incluso tutti i file .java (e non solo i .class)?
- **Hai consegnato due file distinti**, ossia lo ZIP col progetto e il JAR eseguibile?
- **Su EOL, hai premuto il tasto "CONFERMA"** per inviare il tuo elaborato?

ESAME DI FONDAMENTI DI INFORMATICA T-2 del 15/06/2021

Proff. E. Denti – R. Calegari – A. Molesini

Tempo a disposizione: 3 ore

NOME PROGETTO ECLIPSE: CognomeNome-matricola (es. RossiMario-0000123456)

NOME CARTELLA PROGETTO: CognomeNome-matricola (es. RossiMario-0000123456)

NOME ZIP DA CONSEGNARE: CognomeNome-matricola.zip (es. RossiMario-0000123456.zip)

NOME JAR DA CONSEGNARE: CognomeNome-matricola.jar (es. RossiMario-0000123456.jar)

Si devono consegnare DUE FILE: I'intero progetto Eclipse e il JAR eseguibile

Si ricorda che compiti *non compilabili o palesemente lontani da 18/30* NON SARANNO CORRETTI e causeranno la verbalizzazione del giudizio “RESPINTO”

L'amministratore del complesso residenziale *The Dent* ha richiesto lo sviluppo di un'app per calcolare il costo pro-quota del riscaldamento condominiale a gas, per ogni appartamento, su base sia mensile che annuale.

DESCRIZIONE DEL DOMINIO DEL PROBLEMA

Ogni appartamento è dotato di un *partitore di calore* che mensilmente registra il consumo in KWh dell'appartamento: tali registrazioni vengono poi fornite all'amministratore, su base *annuale*, attraverso un apposito file.

L'amministratore ha inoltre a sua disposizione, in un ulteriore file, i dati di ciascun appartamento, ovvero: *codice identificativo, nome del proprietario* (o inquilino nel caso l'appartamento sia concesso in locazione) e *consumo massimo contrattuale di gas, espresso però in standard m³ di gas metano*.

Infine, dalla bolletta trasmessa dal fornitore l'amministratore è in grado di estrarre i seguenti dati:

- Importo totale: importo totale della bolletta
- Costi fissi: costi imputabili alla gestione dell'utenza
- Costi variabili: costi che dipendono dal consumo di gas
- Consumo totale: totale di m³ di gas riportati in bolletta
- Costo al m³: costo di un singolo m³ di gas
- Costo extra al m³: costo di un singolo m³ di gas oltre una certa soglia (maggiore del precedente)

Il calcolo della quota di costo di ciascun appartamento deve avvenire secondo il seguente algoritmo:

1. si recupera il consumo dello specifico mese registrato dal partitore di calore dell'appartamento
2. si recupera il consumo massimo contrattuale di m³ dello specifico appartamento
3. si verifica preliminarmente se il consumo effettivo registrato dal partitore superi o meno il massimo contrattuale di tale appartamento: in tal caso, la *quota eccedente* dovrà essere tariffata usando il *costo extra* al m³ (più elevato del costo standard) così da disincentivare i consumi eccessivi; altrimenti, tutto il gas sarà tariffato a tariffa standard.
4. si aggiungono i costi fissi, che vanno suddivisi in parti uguali tra i condomini
5. infine, se la somma di tutte le quote così calcolate differisce dal totale della bolletta (sia in eccesso che in difetto), si applica a tutte le quote una *correzione uniforme*: a tal fine si calcola dapprima la differenza tra il costo totale della bolletta e la somma delle quote, poi si suddivide tale differenza in parti uguali tra tutti i condomini.

Nel caso di quota annuale l'algoritmo viene ripetuto per ciascun mese dell'anno, in modo da fornire un calcolo quanto più preciso e accurato possibile.

TEMPO TOTALE STIMATO PER SVOLGERE L'INTERO COMPITO: 2h30

Cose da ricordare

- salva costantemente il tuo lavoro: l'informatica a volte può essere “subdolamente ostile”...
- in particolare: se ora compila e stai per fare modifiche, salva la versione attuale (non si sa mai)

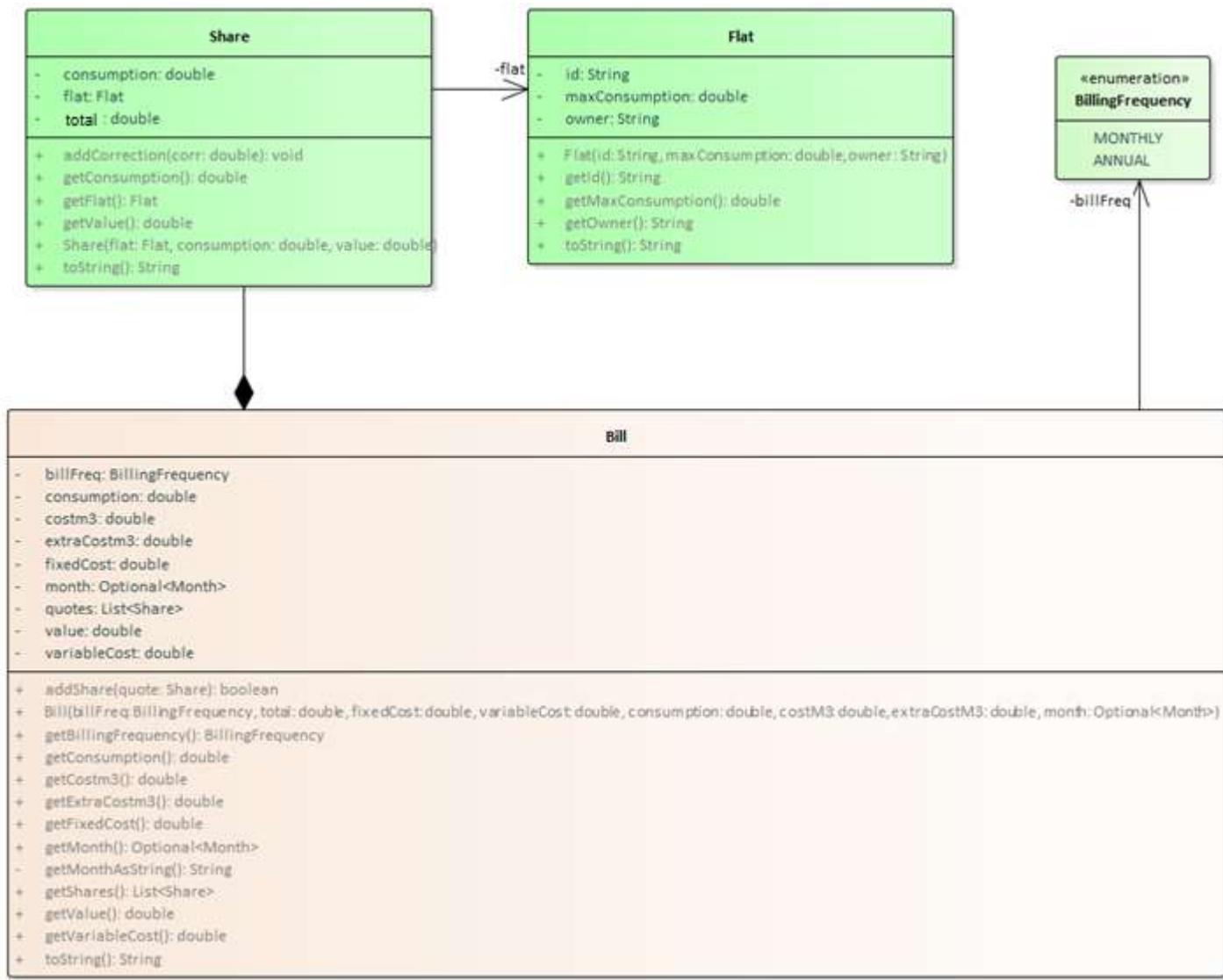
Parte 1

(punti: 14)

Dati (namespace `gasforlife.model`) [TEMPO STIMATO: 30 MINUTI]

(punti: 6)

Il modello dei dati deve essere organizzato secondo il diagramma UML di seguito riportato:



SEMANTICA:

- L'enumerativo **BillingFrequency** (fornito) definisce i due valori **MONTHLY** e **ANNUAL**, rispettivamente per bollette mensili o annuali.
- La classe **Flat** (fornita) modella un singolo appartamento e fornisce *l'identificativo* associato all'appartamento (composto da un codice alfanumerico che identifica il civico, il piano e l'appartamento), il *nome del proprietario* (o dell'inquilino nel caso l'appartamento sia in locazione) ed il *consumo massimo contrattuale mensile* in termini di m³ di gas; appositi accessori consentono di recuperare i valori dei parametri memorizzati dalla classe. Completa la classe un'ovvia implementazione di **toString**.
- La classe **Share** (fornita) modella *ciascuna delle quote* della bolletta associata ad *uno specifico* appartamento. Ogni quota memorizza il consumo e l'importo associati a un dato appartamento; appositi accessori consentono di recuperare i valori dei parametri memorizzati dalla classe, mentre il metodo **addCorrection** permette di aggiornare il valore della quota applicando il “fattore correttivo” specificato. Completa la classe un'ovvia implementazione di **toString**.

d) La classe **Bill** (da realizzare) rappresenta la Bolletta e incapsula i seguenti dati:

- Importo totale: importo totale della bolletta
- Costi fissi: costi imputabili alla gestione dell'utenza
- Costi variabili: costi che dipendono dal consumo di gas
- Consumo: numero totale di m³ indicati nella bolletta
- Costo al m³: costo di un singolo m³ di gas
- Costo extra al m³: costo di un singolo m³ di gas extra quota (maggiore del precedente)
- Mese di riferimento: solo nel caso la bolletta sia mensile (**opzionale**)
- Quote: la lista delle quote (**Share**)

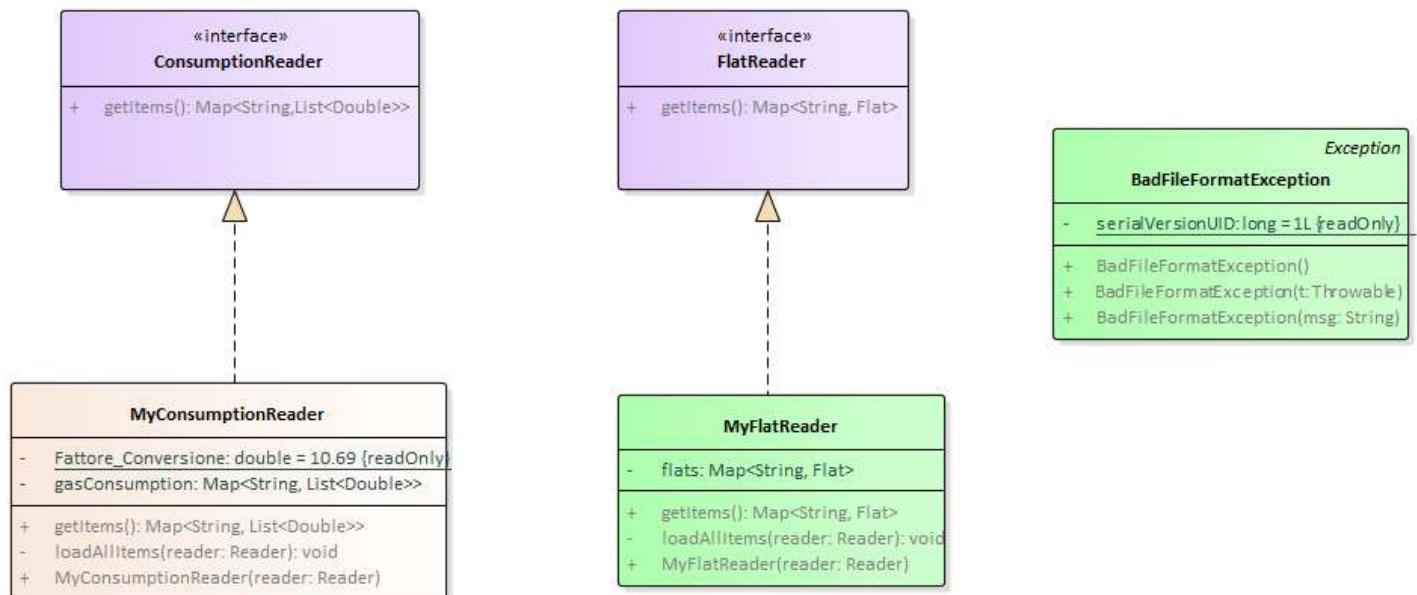
Requisiti:

- Il costruttore riceve tutti gli elementi sopra elencati **tranne l'insieme delle quote**, che verranno calcolate in un secondo momento. Deve controllare che gli argomenti ricevuti non siano null, zero, o numeri negativi, lanciando in tal caso apposita **IllegalArgumentException** con idoneo messaggio.
- Il metodo **addShare** aggiunge al **Bill** una quota (**Share**): restituisce l'esito dell'operazione
- Il metodo **getMonthAsString** deve fornire il nome del mese *secondo la cultura locale italiana*, o la stringa "mese non presente" se esso non è specificato. SUGGERIMENTO: `DateTimeFormatter.ofPattern("MMMM")`
- Il metodo **toString** deve fornire, *uno per riga*, non solo tutti i valori memorizzati nella classe (compreso il mese, nel caso di bolletta mensile) ma anche *tutte le singole quote, ordinate per codice di appartamento* (sempre una per riga).

Persistenza (namespace `gasforlife.persistence`) [TEMPO STIMATO: 40 MINUTI]

(punti: 8)

Questo package definisce il reader per leggere da file gli appartamenti e i consumi.



SEMANTICA:

- a) L'interfaccia **FlatReader** (fornita) dichiara il metodo **getItems** che restituisce una Mappa <String, Flat> con chiave il codice identificativo di ciascun appartamento.
- b) La classe **MyFlatReader** (fornita) implementa **FlatReader**: per ipotesi riceve un **Reader** nel costruttore, che provvede a leggere il file che memorizza i dati relativi ai vari appartamenti.

- c) l'interfaccia **ConsumptionReader** (fornita) dichiara anch'essa un suo metodo **getItems** che restituisce una Mappa <String, List<Double>> in cui la chiave è il codice dell'appartamento, e il valore associato è la lista delle dodici letture mensili del consumo di quell'appartamento.
- d) la classe **MyConsumptionReader** (da realizzare) implementa **ConsumptionReader**: per ipotesi il costruttore riceve un **Reader già aperto** e si occupa della lettura del file memorizzando i dati in un'opportuna Mappa. In caso di problemi di I/O dev'essere lasciata uscire **IOException**, mentre in caso di problemi nel formato delle righe si deve lanciare **BadFormatException** (fornita) con preciso e specifico messaggio d'errore.

FORMATO DEL FILE: ogni riga del file memorizza le dodici letture di un dato appartamento (una per ogni mese).

In ogni riga, il primo elemento è una stringa che rappresenta il *codice identificativo* dell'appartamento, seguita da uno spazio, un ":" e un ulteriore spazio. A seguire vengono le *dodici letture* dei consumi, separate tra loro da una barra (|). È cruciale tenere presente che i valori memorizzati nel file sono espressi in Kwh, mentre la rendicontazione in bolletta è espressa in m³ di gas: pertanto, prima della memorizzazione nella mappa occorre effettuare la *conversione tra le due unità di misura* applicando il fattore di conversione:

$$1 \text{ standard m}^3 \text{ di gas metano} = 10.69 \text{ Kwh}$$

Esempio di file:

1-1A : 1300 1069 1069 780 780 0 0 0 0 780 1400 1400
...
7-1B : 1400 1069 1069 780 780 0 0 0 0 780 1400 1400
7-2A : 930 940 900 680 680 0 0 0 0 680 860 8600
...

Parte 2

(punti: 16)

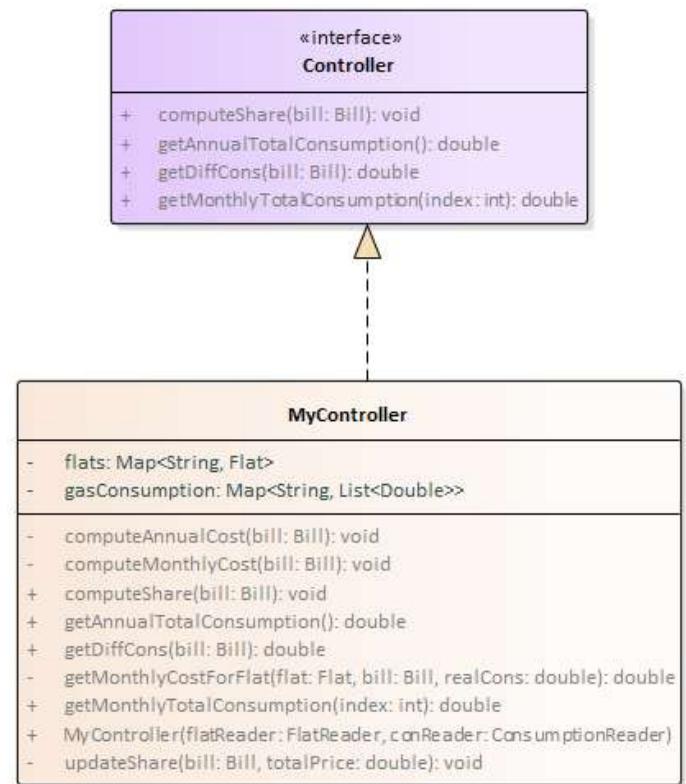
Controller (namespace `gasforlife.controller`) [TEMPO STIMATO: 60 MINUTI]

(punti: 12)

Questo package contiene il controller dell'app.

SEMANTICA:

- a) l'interfaccia **Controller** (fornita) dichiara i metodi:
- **computeShare**: riceve in ingresso un **Bill** e calcola le quote di ogni appartamento memorizzandole all'interno del **Bill** ricevuto, secondo l'algoritmo presentato nell'analisi del dominio.
 - **getAnnualTotalConsumption**: calcola il consumo totale *annuale* di gas per tutto il complesso residenziale.
 - **getMonthlyTotalConsumption**: calcola il consumo totale di gas per tutto il complesso residenziale *per lo specifico mese* ricevuto come argomento.
 - **getDiffCons**: calcola la differenza tra il consumo registrato nella bolletta in ingresso e il consumo effettivo che proviene dalla lettura dei partitori. **ATTENZIONE:** la differenza è calcolata sulla stessa base del **Bill** (quindi, è annuale se **Bill** è annuale, è mensile se **Bill** è mensile).



- b) la classe **MyController** (fornita parzialmente realizzata, ma da completare) implementa tale interfaccia, aggiungendo svariati metodi privati di ausilio (come **updateShare**). In particolare devono essere realizzati:
- il metodo privato **computeMonthlyCost**, che incapsula l'algoritmo di calcolo descritto del Dominio del Problema (mentre l'analogico metodo **computeAnnualCost** è fornito già pronto);

SUGGERIMENTO 1: può essere utile usare il metodo fornito `updateShare`

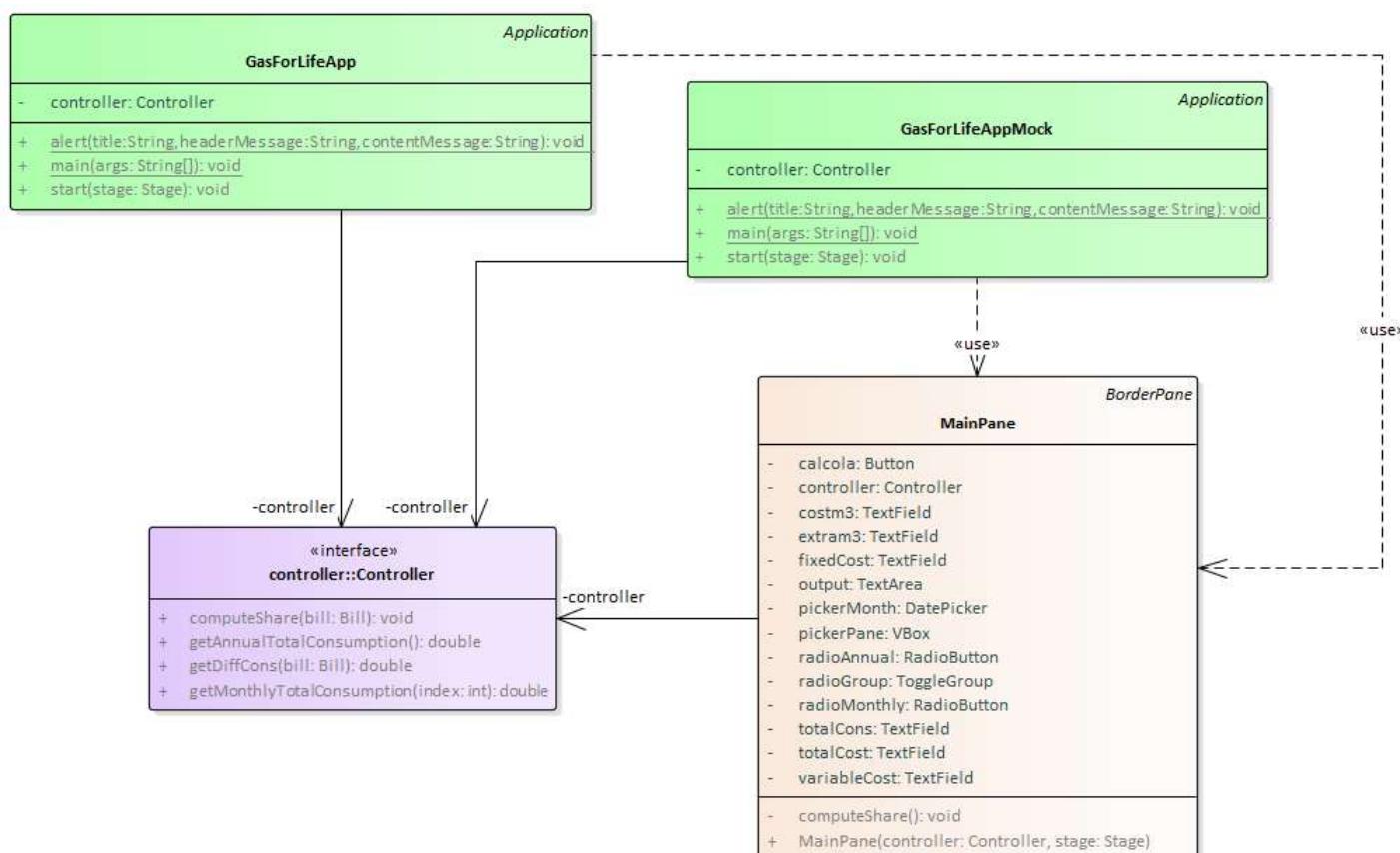
SUGGERIMENTO 2: può essere utile appoggiarsi a un proprio metodo ausiliario `getMonthlyCostForFlat` in cui encapsulare la logica di calcolo del costo mensile per appartamento

- il metodo pubblico `getDiffCons`, destinato a essere invocato dalla GUI nella gestione dell'evento di calcolo, appoggiandosi sui metodi `getAnnualTotalConsumption` e `getMonthlyTotalConsumption` già presenti.

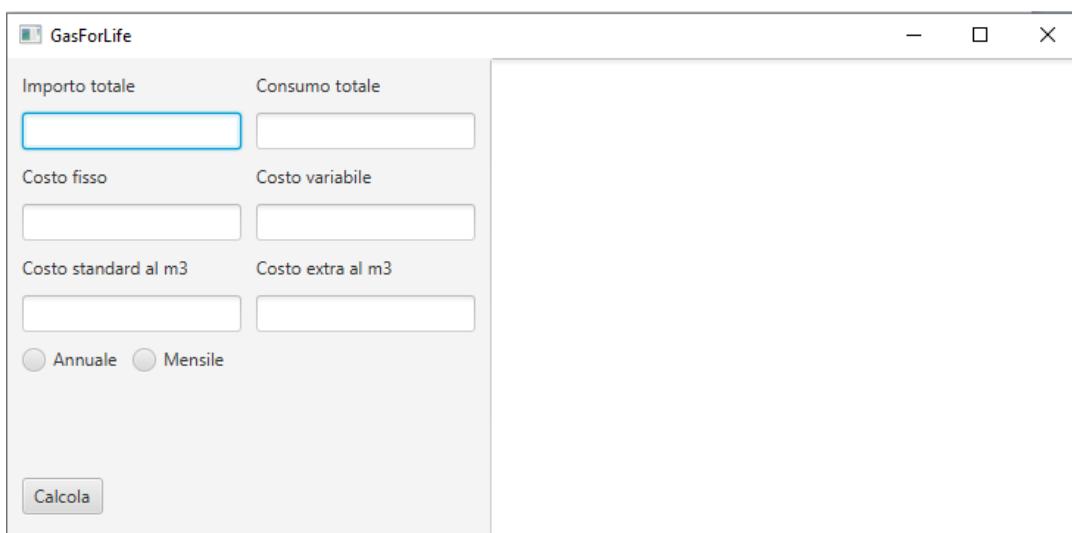
GUI (namespace `gasforlife.ui`) [TEMPO STIMATO: 20 MINUTI]

(punti: 4)

Questo package contiene le classi che rappresentano l'interfaccia grafica. **GasForLifeApp** (fornita) crea la finestra grafica, le classi per la persistenza e il controller. **GasForLifeAppMock** (fornita) funge da mock in caso non sia stata svolta la parte relativa alla persistenza. **MainPane** (fornita parzialmente realizzata, ma da completare) è un **BorderPane** che contiene i widget grafici e permette la gestione dell'evento quando viene premuto il tasto “Calcola”



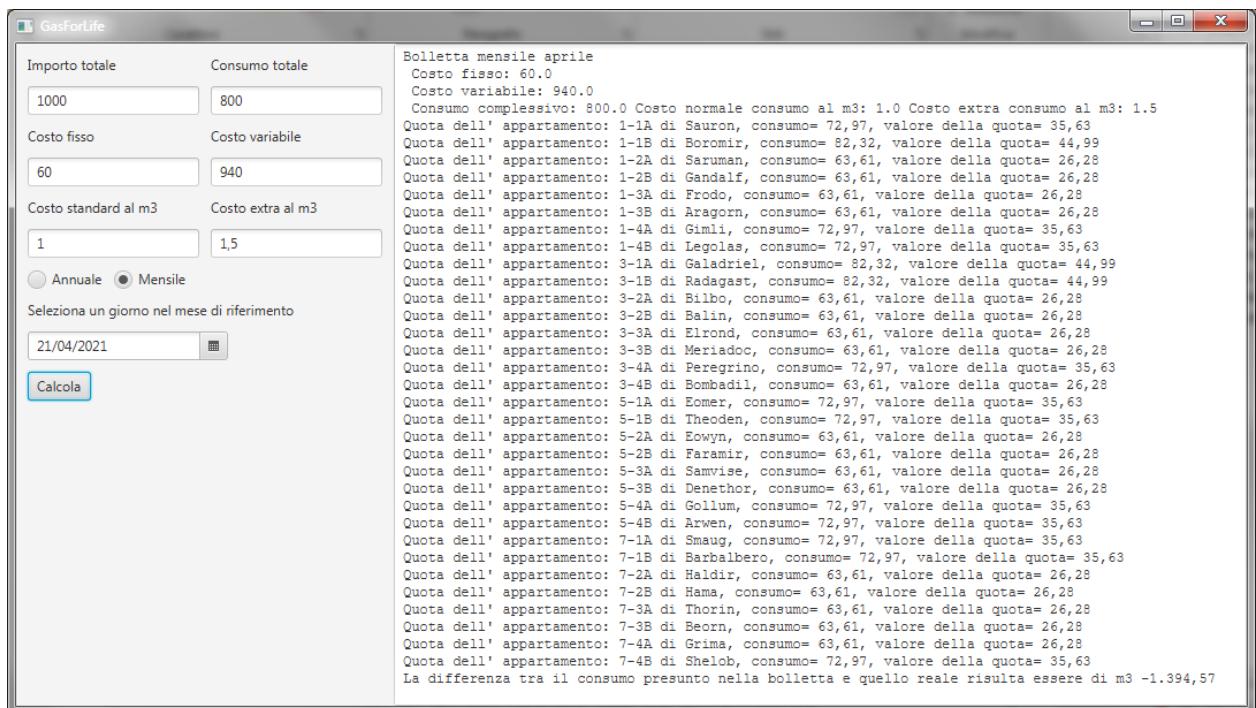
La GUI dev'essere simile (non necessariamente identica) a quella sotto illustrata:



Si tratta di un **BorderPane**, in cui nel lato sinistro sono presenti **sei campi di testo**, sormontati da **apposite label**, che permettono di inserire i dati ricavati dalla Bolletta da ripartire. **Due radio button** (sotto) permettono di specificare se la Bolletta sia **mensile** o **annuale**: quando viene selezionato “mensile” deve comparire un **DatePicker** per selezionare **un giorno all’interno del mese** per il quale si vogliono calcolare le quote. Alla pressione del tasto “Calcola” vengono quindi calcolate le quote, mostrate nella **TextArea** presente sulla parte destra (v. figura sottostante).

Il metodo di gestione dell’evento, **computeShare** (fornito parzialmente realizzato ma **da completare**), dapprima recupera i e valida i parametri immessi nei campi di testo (parte fornita già pronta), poi agisce come segue:

- recupera dal **RadioButton** la frequenza di calcolo (mensile o annuale)
- istanzia il corrispondente **Bill**
- *delega al controller il calcolo delle quote* ed emette sulla **TextArea** il risultato (completo di tutte le informazioni relative alla bolletta, incluse le quote di ciascun appartamento)
- *recupera dal controller l’eventuale differenza di consumi* ed emette sulla **TextArea** la frase finale relativa all’eventuale *differenza tra consumo stimato in bolletta e consumo effettivo, opportunamente formattata con due cifre decimali*.



NB: il metodo statico ausiliario **alert** consente di mostrare una finestra di dialogo utile a segnalare errori all’utente.

Cose da ricordare

- salva costantemente il tuo lavoro: l’informatica a volte può essere “subdolamente ostile”..
- in particolare: se ora compila e stai per fare modifiche, salva la versione attuale (non si sa mai)

Checklist di consegna

- Hai fatto un **JAR eseguibile**, che contenga cioè l’indicazione del main?
- Hai controllato che **si compilii e ci sia tutto**? [NB: non includere il PDF del testo]
- Hai **rinominato IL PROGETTO**, lo ZIP e il JAR esattamente come richiesto?
- Hai **chiamato** la cartella del progetto esattamente come richiesto?
- **Hai fatto un unico file ZIP (NON .7z, rar o altri formati)** contenente l’intero progetto?
In particolare, ti sei assicurato di aver incluso tutti i file .java (e non solo i .class)?
- **Hai consegnato DUE file distinti, ossia lo ZIP col progetto e il JAR eseguibile?**
- Su EOL, hai premuto il tasto “CONFERMA” per inviare il tuo elaborato?

ESAME DI FONDAMENTI DI INFORMATICA T-2 del 4/2/2021

Proff. E. Denti – R. Calegari – A. Molesini

Tempo a disposizione: 3 ore

NOME PROGETTO ECLIPSE: CognomeNome-matricola (es. RossiMario-0000123456)

NOME CARTELLA PROGETTO: CognomeNome-matricola (es. RossiMario-0000123456)

NOME ZIP DA CONSEGNARE: CognomeNome-matricola.zip (es. RossiMario-0000123456.zip)

NOME JAR DA CONSEGNARE: CognomeNome-matricola.jar (es. RossiMario-0000123456.jar)

Si devono consegnare DUE FILE: *l'intero progetto Eclipse e il JAR eseguibile*

Si ricorda che compiti *non compilabili o palesemente lontani da 18/30* NON SARANNO CORRETTI e causeranno la verbalizzazione del giudizio “RESPINTO”

Si vuole sviluppare un'app per simulare il gioco della Ghigliottina del noto show preserale “L'Eredità”

DESCRIZIONE DEL DOMINIO DEL PROBLEMA

Scopo del gioco è indovinare una parola a partire da altre parole (*indizi*, tipicamente 5), che si collegano ad essa per analogie, modi di dire, etc. Tali indizi però devono essere conquistati: a ogni manche, al concorrente vengono proposte due parole, di cui una sola è l'indizio giusto. Appena il concorrente fa la sua scelta, scatta la Ghigliottina: se la parola scelta è l'indizio corretto, il montepremi rimane intoccato, altrimenti viene dimezzato. L'indizio corretto viene poi comunque mostrato e lasciato visibile. A seconda delle scelte (e della fortuna) del concorrente si può quindi dimezzare da 0 a 5 volte.

Una volta accumulati i 5 indizi, il concorrente ha un minuto di tempo per pensare alla parola che li accomuna tutti: allo scadere del tempo deve scriverla su un cartoncino. Successivamente si procede alla verifica: se la parola scritta risulta la risposta esatta, il concorrente vince il montepremi; altrimenti, non vince nulla.



ESEMPIO

Si supponga che vengano via via proposte al concorrente le seguenti coppie di parole:

1. Lordo / Lardo (indizio corretto: Lardo)
2. Zampino / Zampone (indizio corretto: Zampino)
3. Tetto / Veranda (indizio corretto: Tetto)
4. Lenta / Frettolosa (indizio corretto: Frettolosa)
5. Gino Paoli / Mina (indizio corretto: Gino Paoli)

A seconda delle scelte che via via effettua, il concorrente può dimezzare zero o più volte il montepremi iniziale (si supponga siano € 100.000), giocando quindi per una cifra compresa fra €100.000 e € 3.125.

Allo scadere del tempo, il concorrente scrive la parola che ritiene accomuni i cinque indizi: se, alla successiva verifica, essa risulta quella corretta, il concorrente vince il montepremi; altrimenti non vince nulla.

In questo caso, la risposta esatta è “Gatta”, poiché:

1. Lardo → proverbio: “Tanto va la gatta al lardo che ci lascia lo zampino”
2. Zampino → proverbio: “Tanto va la gatta al lardo che ci lascia lo zampino”
3. Tetto → titolo di film e opera teatrale: “La gatta sul tetto che scotta”
4. Frettolosa → proverbio: “La gatta frettolosa fa i gattini ciechi”
5. Gino Paoli → titolo di canzone: “La gatta”

Nell'applicazione da sviluppare, per semplicità, non vi sarà alcun timer a scandire lo scorrere del tempo dopo la scelta degli indizi: il giocatore avrà tutto il tempo che vuole per pensare e scrivere la sua risposta. Si potrà poi verificare il risultato premendo un apposito pulsante (“SVELA”): un dialogo informerà del risultato (vedere figure in coda al testo).

NB: le animazioni sono fornite già realizzate nell'apposito pannello *GigliottinaPanel*.

Cose da ricordare

- salva costantemente il tuo lavoro: l'informatica a volte può essere “subdolamente ostile”..
- in particolare: se ora compila e stai per fare modifiche, salva la versione attuale (non si sa mai)

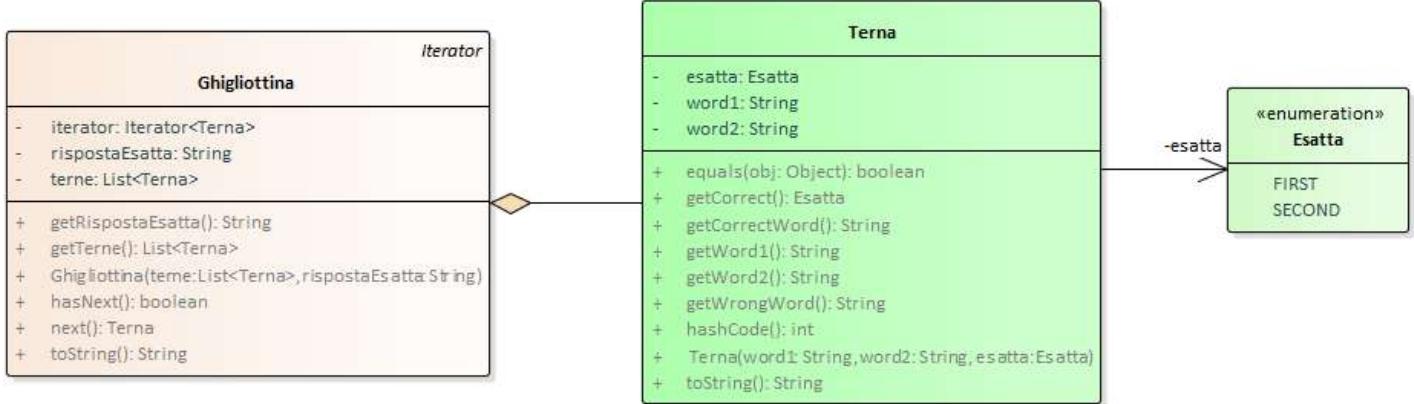
Parte 1

(punti: 18)

Dati (namespace `ghigliottina.model`)

(punti: 5)

Il modello dei dati deve essere organizzato secondo il diagramma UML di seguito riportato:



SEMANTICA:

- L'enumerativo **Esatta** (fornito) definisce i due valori FIRST e SECOND per denotare quale delle due parole proposte sia l'indizio corretto
- La classe **Terna** (fornita) modella una proposta di due parole come possibile indizio, con indicazione della relativa risposta esatta. Ad esempio, con riferimento all'esempio sopra, l'alternativa fra "Lordo" e "Lardo", in cui l'indizio corretto è la seconda parola ("Lardo"), sarà rappresentato dalla terna ("Lordo", "Lardo", **SECOND**). Il costruttore riceve le due parole e l'indicazione di quale sia l'indizio corretto (tramite uno dei due valori dell'enumerativo); appositi accessori consentono di recuperare le due parole sia in base all'ordine con cui compaiono nella terna (**getFirst/getSecond**), sia in base alla loro esattezza (**getCorrectWord/getWrongWord**); è anche possibile recuperare l'indicazione di quale sia quella esatta come valore dell'enumerativo (**getCorrect**). Completano la classe alcune implementazioni ovvie di **toString**, **equals** e **hashCode**.
- La classe **Ghigliottina** (da realizzare) rappresenta il gioco e pertanto incapsula:
 - una lista di **Terna** (tipicamente saranno 5, ma occorre essere generali) con le proposte di indizi
 - la risposta esatta (una stringa)

Il costruttore deve controllare accuratamente la qualità degli argomenti ricevuti, in particolare che essi non siano null né vuoti (nel caso della risposta esatta, è richiesto che non sia blank ossia composta unicamente di spazi, tabulazioni, etc.): in tal caso dovrà essere lanciata apposita **IllegalArgumentException**, con idoneo messaggio.

Non è invece richiesta alcuna particolare implementazione per **toString**, **equals** e **hashCode**.

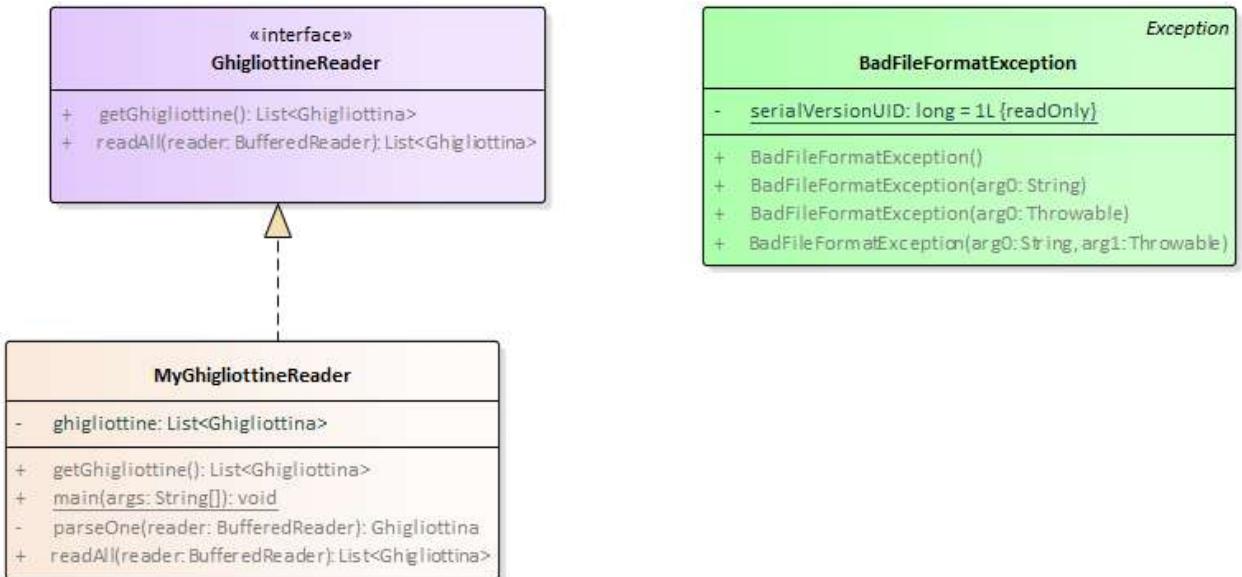
Persistenza (namespace `ghigliottina.persistence`)

(punti: 13)

Questo package definisce il reader per leggere da file le possibili ghigliottine.

SEMANTICA:

- l'interfaccia **GhigliottineReader** (fornita) dichiara i due metodi **readAll** e **getGhigliottine**: il primo legge da un (buffered)reader una serie di **Ghigliottina** e le restituisce sotto forma di lista; il secondo restituisce semplicemente la lista già letta (o lancia **NullPointerException** se la lettura non è ancora avvenuta)
- la classe **MyGhigliottineReader** (da realizzare) implementa tale interfaccia: si suggerisce di strutturare la lettura su due metodi, appoggiandosi a un metodo privato ausiliario **readOne** per la lettura di una singola **Ghigliottina**, così da separare meglio la fase di lettura e verifica della singola ghigliottina da quella, più esterna, di lettura della sequenza di ghigliottine. A questo fine, è opportuno che il metodo privato ausiliario **readOne** restituisca **null** quando arriva a fine file, così da permettere un'articolazione standard del ciclo di lettura esterno in **readAll**.



Il metodo ***readAll*** riceve un ***Reader*** già aperto: restituisce la lista, eventualmente vuota ma mai nulla, delle ghigliottine lette dal file. In caso di problemi di I/O dev'essere lasciata uscire ***IOException***, mentre in caso di problemi nel formato delle righe si deve lanciare ***BadFormatException*** (fornita) con preciso e specifico messaggio d'errore.

FORMATO DEL FILE: il file è strutturato a blocchi, ognuno dei quali descrive una ghigliottina. Ogni blocco consiste innanzitutto di un certo numero di righe che descrivono terne: a seguire vi è la riga con la risposta esatta, indi per ultima una riga composta di lineette (almeno 6), che chiude il blocco.

Le righe che descrivono terne hanno la forma ***PAROLA1/PAROLA2=QUALEDELLEDUE***, dove ***PAROLA1*** e ***PAROLA2*** sono sequenze di caratteri qualsiasi diversi da '/' e '=', mentre ***QUALEDELLEDUE*** è "***FIRST***" o "***SECOND***". Intorno alle parole, o prima o dopo di esse, possono esservi spazi o tabulazioni in numero qualunque.

ESEMPI DI FILE LECITI (il rombo, solo per motivi di visibilità in questa sede, indica la tabulazione):

LEGGE/DECRETO=FIRST PATATA/CAROTA=FIRST SOPRA/SOTTO=SECOND FATICA/SACRIFICIO=SECOND FANTASMA/ASMA=FIRST Risposta esatta=SPIRITO ----- CANTATO/INCANTATO=SECOND ...	LEGGE/DECRETO = FIRST PATATA / CAROTA=FIRST SOPRA◆/SOTTO=SECOND FATICA/SACRIFICIO◆=◆SECOND◆ FANTASMA/ASMA=FIRST Risposta esatta=SPIRITO ----- CANTATO/INCANTATO=SECOND ...
--	--

Parte 2

(punti: 12)

Controller (namespace *ghigliottina.ui*)

(punti: 3)

- a) l'interfaccia ***Controller*** (fornita – UML mostrato insieme a quello della UI) dichiara i due metodi ***listaGhigliottine*** e ***sorreggiaGhigliottina***: il primo restituisce la lista di ***Ghigliottina*** (che si suppone ricevuta dal costruttore della classe che la implementa), il secondo sorteggia e restituisce una singola ***Ghigliottina*** a caso fra quelle disponibili
- b) la classe ***MyController*** (**da realizzare**) implementa tale interfaccia.

GUI (namespace *ghigliottina.ui*)

(punti: 9)

L'interfaccia grafica è costituita da due pannelli annidati: all'interno, un ***GhigliottinaPanel*** (fornito) incorpora e gestisce tutta la grafica del gioco; all'esterno, un ***OuterGhigliottinaPanel*** (**da completare**) gestisce i campi di testo in cui il concorrente scrive la risposta e quello, adiacente, in cui – alla pressione del pulsante SVELA – comparirà la risposta corretta. L'applicazione nel suo complesso è attivata da ***GhigliottinaApp*** (fornita), che provvede anche a fornire al controller una ghigliottina prestabilita in caso di malfunzionamento del reader, fungendo anche da mock.

La GUI dev'essere simile (non necessariamente identica) a quella più oltre illustrata (Fig.1): si tratta di un ***BorderPane***, in cui in alto sono presenti due campi di testo, sormontati da apposite label: quello destinato a mostrare la risposta esatta non è editabile. Alla loro destra il pulsante SVELA, quando premuto:

- scatena la verifica della risposta (che dev'essere insensibile alla grafia e quindi a maiuscole/minuscole)
- fa comparire i dialoghi corrispondenti (Figg. 6,7,8)
- infine resetta l'app allo stato iniziale, con una nuova ghigliottina (Fig. 9).

Sotto, invece, è presente un ***GhigliottinaPanel*** (fornito), il cui costruttore si aspetta di ricevere l'ammontare iniziale del montepremi: appositi accessori (`getMontepremi`/`getMontepremiAsString`) consentono di recuperare in qualsiasi momento, e in particolare quindi al termine della ghigliottina, l'ammontare residuo del montepremi, rispettivamente sotto forma di intero e di stringa già opportunamente formattata.

Quando la GUI compare, il gioco è già attivo e viene proposta al concorrente la prima coppia di parole (Fig. 1): il concorrente sceglie quella desiderata utilizzando i radiobutton, poi conferma premendo il pulsante "Ghigliottina". Successivamente il gioco evolve, eventualmente dimezzando via via il montepremi a ogni scelta errata (Figg. 1-3): una volta mostrate tutte le terne (solitamente cinque), il pulsante "Ghigliottina" viene disabilitato e lo sfondo diventa grigio (Fig. 4). A quel punto il concorrente scrive la propria soluzione nel campo di testo apposito (a destra in Fig. 5), indi preme il pulsante SVELA: il dialogo che compare mostra l'esito del gioco (Fig. 6 o, in altra situazione, Fig. 7). Se si preme il pulsante SVELA prima di aver scritto una soluzione (Fig. 8), compare un dialogo di errore.

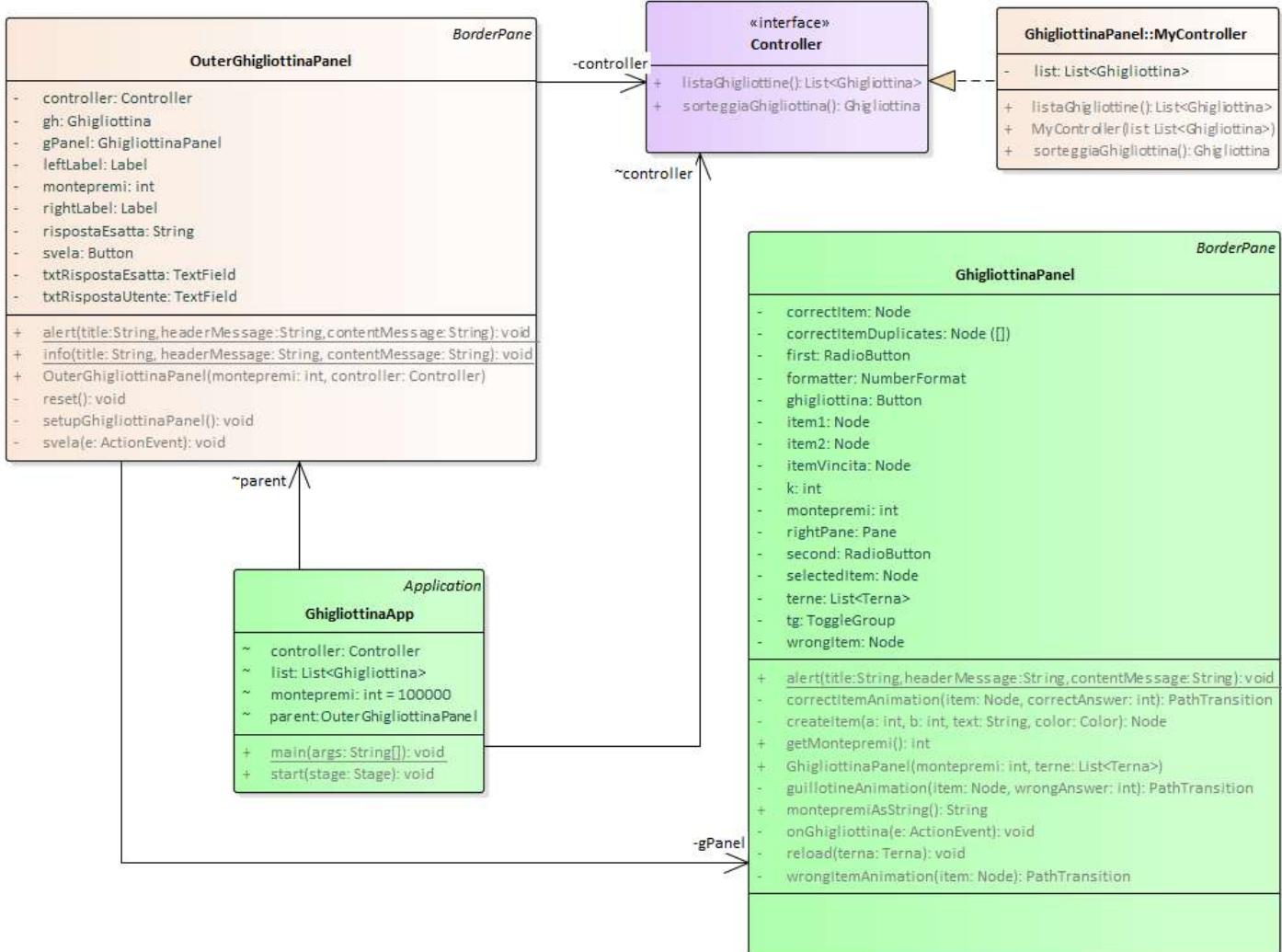
Per far comparire questi dialoghi sono disponibili nella classe ***OuterGhigliottinaPanel*** due metodi statici:

- ***alert***, che mostra consente di una finestra di dialogo utile a segnalare errori all'utente si riconosce per l'icona "X" (Figg. 8).
- ***info***, che mostra consente di una finestra di dialogo utile a mostrare informazioni all'utente (per le risposte): si riconosce per l'icona "i" (Figg. 6,7).

È richiesto di **completare opportunamente *OuterGhigliottinaPanel***, in particolare:

- predisponendo la struttura del pannello interno superiore, con i campi di testo e il pulsante SVELA;
- gestendo l'evento corrispondente alla pressione di tale pulsante, ovvero:
 - scatenando la verifica della risposta (insensibile alla grafia e quindi a maiuscole/minuscole)
 - facendo comparire i dialoghi corrispondenti (Figg. 6,7,8)
 - infine resettando l'app allo stato iniziale, con una nuova ghigliottina (Fig. 9).

*****SEGUE ALLA PAGINA SUCCESSIVA*****



Cose da ricordare

- salva costantemente il tuo lavoro: l'informatica a volte può essere “subdolamente ostile”..
- in particolare: se ora compila e stai per fare modifiche, salva la versione attuale (non si sa mai)

Checklist di consegna

- Hai fatto un **JAR eseguibile**, che contenga cioè l'indicazione del main?
- Hai controllato che **si compili e ci sia tutto**? [NB: non includere il PDF del testo]
- Hai **rinominato IL PROGETTO**, lo ZIP e il JAR esattamente come richiesto?
- Hai **chiamato** la cartella del progetto esattamente come richiesto?
- **Hai fatto un unico file ZIP (NON .7z, rar o altri formati) contenente l'intero progetto?**
In particolare, ti sei assicurato di aver incluso tutti i file .java (e non solo i .class)?
- **Hai consegnato DUE file distinti, ossia lo ZIP col progetto e il JAR eseguibile?**
- Su EOL, hai premuto il tasto “**CONFERMA**” per inviare il tuo elaborato?



Figura 1



Figura 2



Figura 3



Figura 4



Figura 5



Figura 6

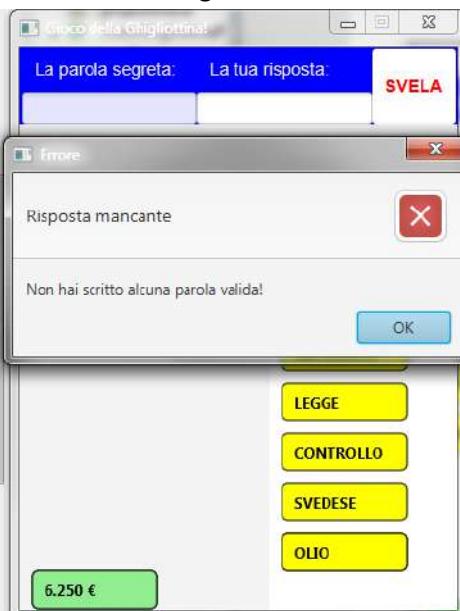
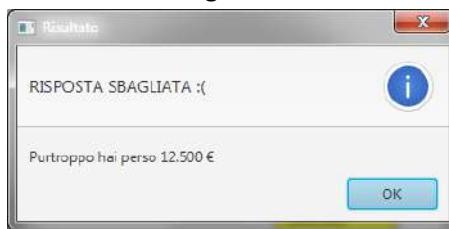


Figura 8



Figura 9

Figura 7

ESAME DI FONDAMENTI DI INFORMATICA T-2 del 13/1/2021

Proff. E. Denti – R. Calegari – A. Molesini

Tempo a disposizione: 3 ore

NOME PROGETTO ECLIPSE: CognomeNome-matricola (es. RossiMario-0000123456)

NOME CARTELLA PROGETTO: CognomeNome-matricola (es. RossiMario-0000123456)

NOME ZIP DA CONSEGNARE: CognomeNome-matricola.zip (es. RossiMario-0000123456.zip)

NOME JAR DA CONSEGNARE: CognomeNome-matricola.jar (es. RossiMario-0000123456.jar)

Si devono consegnare DUE FILE: *I'intero progetto Eclipse e il JAR eseguibile*

Si ricorda che compiti *non compilabili o palesemente lontani da 18/30* NON SARANNO CORRETTI e causeranno la verbalizzazione del giudizio “RESPINTO”

Si vuole sviluppare un'app per giocare a Master Mind (detto anche Codice Segreto).

DESCRIZIONE DEL DOMINIO DEL PROBLEMA

Master Mind è un gioco di abilità fra due giocatori, il *codificatore* e il *risolutore*. Preliminarmente, il codificatore deve inventare una *combinazione* di colori (il codice segreto) che il risolutore deve poi indovinare entro un prefissato *numero massimo di tentativi*.

A tal fine sono disponibili *piolini colorati* di diversi colori. I colori disponibili sono di norma 6, ma possono essere 8 o anche più nelle versioni per giocatori esperti; analogamente, la combinazione è di norma costituita da 4 piolini colorati, ma può essere di 5 nelle versioni per esperti. La figura a lato illustra la versione a 8 colori con combinazione da 4.

EVOLUZIONE DEL GIOCO

A ogni tentativo del risolutore, il codificatore risponde con una serie di *piolini risposta* bianchi o neri, collocati a lato del tentativo del risolutore, che assumono il seguente significato:

- Per ogni colore indovinato nella giusta posizione: un piolino nero
- Per ogni colore presente ma in posizione errata: un piolino bianco
- Per ogni colore non presente nella combinazione segreta: nessun piolino.

Colori:		
	Proposte	Risposte
1	● ○ ○ ○	○○
2	● ● ○ ○ ○ ○	○○
3	● ● ○ ○ ○ ○ ○ ○	○○
4	● ● ○ ○ ○ ○ ○ ○	●
5	● ○ ○ ○ ○ ○ ○ ○	○
	-----	●●●●

Pertanto la risposta può comprendere da 0 a 4 piolini, con un mix di bianchi e neri: 4 piolini neri equivalgono a combinazione indovinata, nel qual caso la vittoria è del risolutore; se invece questi non riesce a indovinare la combinazione segreta entro il prefissato numero massimo di tentativi, la vittoria va al codificatore.

L'ordine dei piolini di risposta non è significativo: conta unicamente la quantità di piolini neri e bianchi. Per questo, spesso si conviene di elencare prima i neri e poi i bianchi, così da favorire il confronto fra i tentativi e il ragionamento.

NB: il codificatore non ha limiti nell'uso dei colori, che possono essere anche ripetuti più volte.

CURIOSITÀ

Il gioco può essere fatto anche sostituendo ai colori dei numeri, o ai piolini risposta due simboli (“O”/”X”, o altro); alcune varianti prevedono piolini risposta bianchi e rossi (anziché neri); alcuni giochi al computer usano rettangoli colorati invece di cerchi; etc. Un simulatore online è disponibile sul sito del CNR al link <http://mastermind.itd.cnr.it/index.php>.

ESEMPI

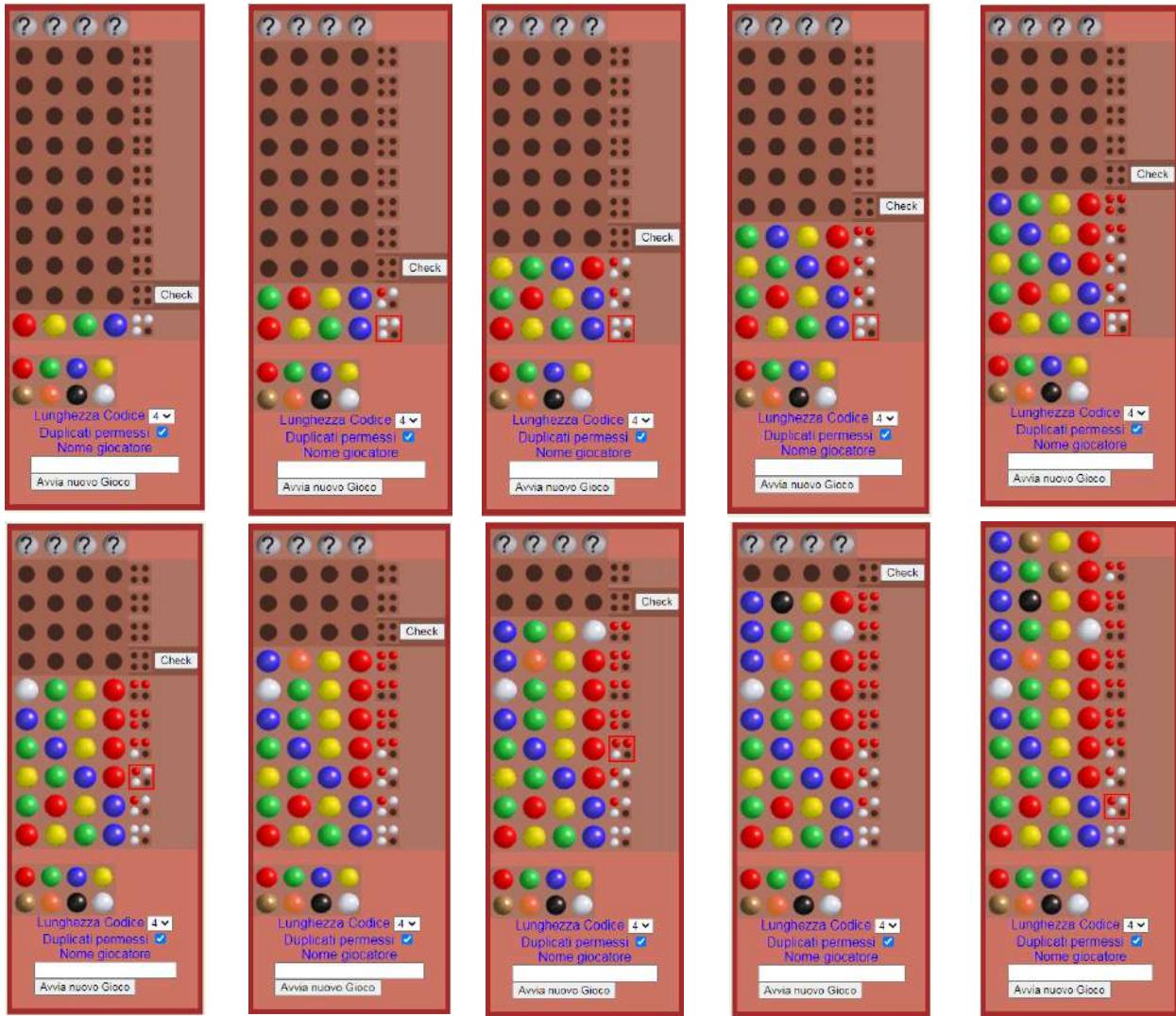
Si riportano di seguito alcune partite al simulatore, una persa dal risolutore, le altre vinte in diverse situazioni: i piolini risposta in questo caso sono bianchi e rossi, in quanto il nero è usato per indicare il vuoto.

Nell'applicazione da sviluppare, per semplicità, la descrizione dei colori sarà testuale (“ROSSO”, “VERDE”, etc.) e lo stesso varrà per i piolini-risposta (“BIANCO”, “NERO”, “VUOTO”): in particolare si userà il termine “VUOTO” per indicare una casella senza piolini (v. oltre). Solo nella grafica si utilizzeranno combo con sfondo colorato (fornito), così da dare maggior gradevolezza al gioco (vedere figure in coda al testo).

Cose da ricordare

- salva costantemente il tuo lavoro: l'informatica a volte può essere “subdolamente ostile”
- se ora compila e stai per fare modifiche, salva la versione attuale (non si sa mai...)

ESEMPIO PASSO-PASSO: partita persa dal risolutore



ALTRI ESEMPI: partite vinte dal risolutore



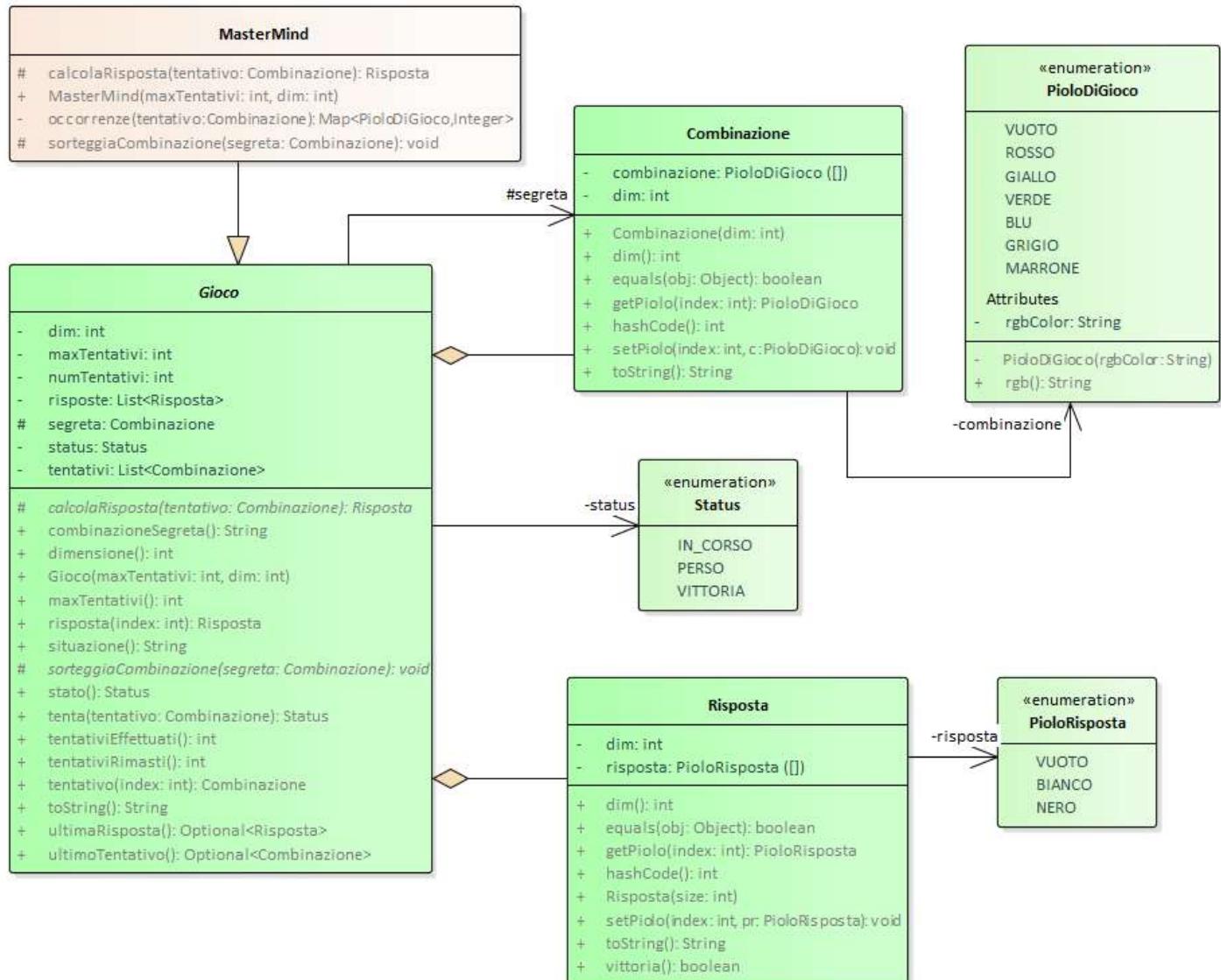
Parte 1

(punti: 18)

Dati (namespace mastermind.model)

(punti: 13)

Il modello dei dati deve essere organizzato secondo il diagramma UML di seguito riportato:



SEMANTICA:

- L'enumerativo **PioloDiGioco** (fornito) definisce i colori possibili: a ciascuno è associato il codice `rgb`, utilizzato nella grafica per colorare opportunamente lo sfondo delle caselle. A fini di test è previsto anche il colore "VUOTO".
- L'enumerativo **PioloRisposta** (fornito) definisce i tre casi possibili – nero, bianco, vuoto.
- L'enumerativo **Status** (fornito) definisce i tre stati possibili del gioco – in corso, partita persa, partita vinta.
- La classe **Combinazione** (fornita) modella una combinazione di colori: il costruttore ne riceve la dimensione (ossia il numero di pioli-colore di cui è composta) e la inizializza con posizioni tutte vuote. Gli accessori `setPiolo`/`getPiolo` permettono di impostare l'i-esimo piolo della combinazione (numerati da 0 a N-1, da sinistra a destra). Il metodo `toString` emette la sequenza di colori separati da virgole, mentre `equals` ed `hashcode` sono definite in modo tale da consentire un facile confronto fra combinazioni.
- La classe **Risposta** (fornita), analoga alla precedente, modella invece una risposta costituita da una sequenza di piolini neri, bianchi o vuoti. Come sopra, gli accessori `setPiolo`/`getPiolo` permettono di impostare l'i-esimo piolo-risposta della combinazione (numerati da 0 a N-1, da sinistra a destra). Il metodo `toString` emette la sequenza di pioli-risposta separati da virgole, mentre `equals` ed `hashcode` sono definite in modo tale da consentire un facile confronto fra combinazioni. Il metodo `vittoria` è vero se la risposta è costituita da tutti piolini neri.
- La classe astratta **Gioco** (fornita) implementa lo schema generale di gioco: il costruttore riceve il numero massimo di tentativi e la dimensione della combinazione. Sono forniti molti metodi per recuperare il numero massimo di

tentativi, il numero di tentativi effettuati e quelli rimasti, il tentativo i-esimo e la risposta i-esima, l'ultimo tentativo e l'ultima risposta, lo stato del gioco (uno **Status**) e la combinazione segreta. Metodi rilevanti:

- Il metodo **situazione** restituisce una stringa descrittiva dell'intera sequenza di tentativi e rispettive risposte, mentre **toString** completa tali informazioni con alcune stringhe descrittive e lo stato del gioco
- Il metodo **tenta** effettua una giocata con la binazione tentativo ricevuta come argomento: internamente calcola la risposta, quindi aggiorna e restituisce lo stato del gioco dopo tale giocata.

I due metodi astratti (protetti) **sorreggiaCombinazione** e **calcolaRisposta** astraggono la specifica logica di generazione della combinazione segreta e dell'algoritmo di calcolo della risposta.

g) La classe concreta **MasterMind (da realizzare)** deve specializzare **Gioco** nel caso di Mastermind, e pertanto:

- Il metodo **sorreggiaCombinazione** deve sorteggiare i colori richiesti (anche ripetuti)
- Il metodo **calcolaRisposta** deve calcolare la risposta secondo le regole di Master Mind descritte nel Dominio del Problema.

SUGGERIMENTO: per il calcolo dei piolini bianchi, che richiede attenzione in particolare in presenza di colori ripetuti, può convenire calcolare preliminarmente il totale dei piolini (bianchi+neri) della risposta, sottraendo poi quelli neri (facili da calcolare). A tal fine sono possibili diverse strategie, quali ad esempio:

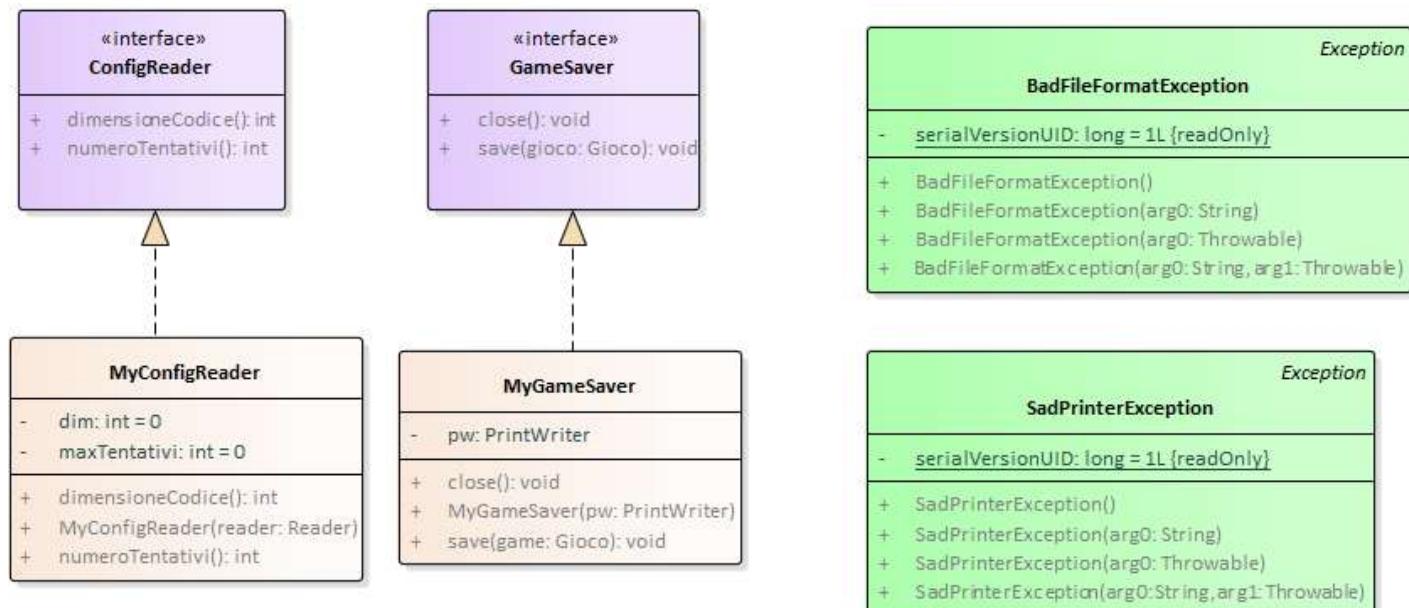
- a) Operando su una copia della combinazione segreta, considerare un piolo per volta, svuotando via via quelli già considerati, in modo da non contarli più volte
- b) Calcolare preventivamente il numero di occorrenze di ogni colore rispettivamente nella combinazione segreta e nel tentativo corrente, agendo poi per differenza.

Non vi sono invece, evidentemente, particolari difficoltà per il calcolo dei piolini neri.

Persistenza (namespace `mastermind.persistence`)

(punti: 5)

Questo package definisce due componenti: **GameSaver** per stampare su file (`gameover.txt`) lo stato di una partita, e **ConfigReader** per leggere da un file di testo (`config.txt`) la configurazione iniziale del gioco (ovvero, la dimensione della combinazione segreta e il numero massimo di tentativi ammessi).



SEMANTICA:

- a) l'interfaccia **ConfigReader** (fornita) dichiara i due metodi **dimensioneCodice** e **numeroTentativi**, dall'ovvio significato
- b) la classe **MyConfigReader (da realizzare)** implementa tale interfaccia: il costruttore riceve un **Reader** già aperto, da cui legge le due righe di configurazione, nel formato sotto specificato. L'ordine delle due righe non è prestabilito:

potrebbe esserci prima la riga relativa alla dimensione o prima quella relativa ai tentativi, non è dato sapere. È il costruttore a svolgere tutto il lavoro di lettura, memorizzando infine i due valori letti in due proprietà interne, che vengono poi restituite dai due metodi *dimensioneCodice* e *numeroTentativi*. In caso di problemi di I/O o nel formato delle righe, il costruttore deve lanciare *BadFormatException* (fornita) con apposito messaggio d'errore.

FORMATO DEL FILE: ogni riga contiene una delle due dichiarazioni “Tentativi” o “Lunghezza combinazione”, scritte con qualunque sequenza di maiuscole e/o minuscole, seguita dal carattere “=” e poi, dopo eventualmente spazi o tabulazioni intermedi, il valore intero. Da notare che fra le due parole “Lunghezza” e “combinazione” dev’esservi invece esattamente uno e un solo spazio, altrimenti la dichiarazione non sarà riconosciuta valida.

ESEMPI DI FILE LECITI:

Tentativi = 10 Lunghezza combinazione = 4	Tentativi = 10 Lunghezza combinazione = 4
--	--

- c) l’interfaccia **GameSaver** (fornita) dichiara i metodi *save*, che salva un **Gioco**, e *close*, che chiude il writer di uscita
 - d) la classe **MyGameSaver (da realizzare)** implementa tale interfaccia: il costruttore deve ricevere un *PrintWriter*, che poi utilizza nei metodi *save* (per salvare lo stato attuale del gioco) e *close* (per chiudere il writer stesso).
- NB: successive chiamate a *save* causano l’accumulo (append) delle diverse fasi del gioco nello stesso file.

Parte 2

(punti: 12)

Controller (namespace mastermind.ui)

Il controller (il cui UML è fornito sotto insieme alla UI) – articolato in interfaccia e implementazione – è fornito già pronto: il suo stato interno crea e gestisce il **Gioco** con tutto lo stato della partita, nonché il **GameSaver** da usare per i salvataggi su file. Conseguentemente, i suoi metodi fanno da ponte con entrambe tali entità, quasi sempre richiamando gli omonimi metodi di **Gioco**. Da notare:

- il costruttore separa nel metodo accessorio *init* la logica di inizializzazione della partita, così da poterla più facilmente riutilizzare nel metodo *restart*
- *restart* reinizializza lo stato del controller per una nuova partita (quindi nuovo **Gioco** e nuovo **GameSaver**)

L’interfaccia **Controller** offre altresì il metodo statico *alert* per far comparire una finestra di dialogo utile a segnalare errori all’utente (in questa applicazione, solo nel caso in cui la stampa su file fallisca per qualche motivo).

GUI (namespace mastermind.ui)

(punti: 12)

In caso di malfunzionamento del ConfigReader, usare come main MasterMindAppMock, che utilizza valori di configurazione di default anziché leggerli da file.

L’interfaccia grafica dev’essere simile (non necessariamente identica) a quella sotto illustrata (Fig.1): in alto sono presenti tante combobox quanta la dimensione della combinazione segreta specificata nel file di configurazione (nell’esempio, 4), tutte precaricate con i colori possibili ad esclusione del colore “VUOTO”, che avendo solo fini di test non deve essere incluso. In basso, la barra di stato indica il numero massimo di tentativi, quelli rimasti e lo stato del gioco.

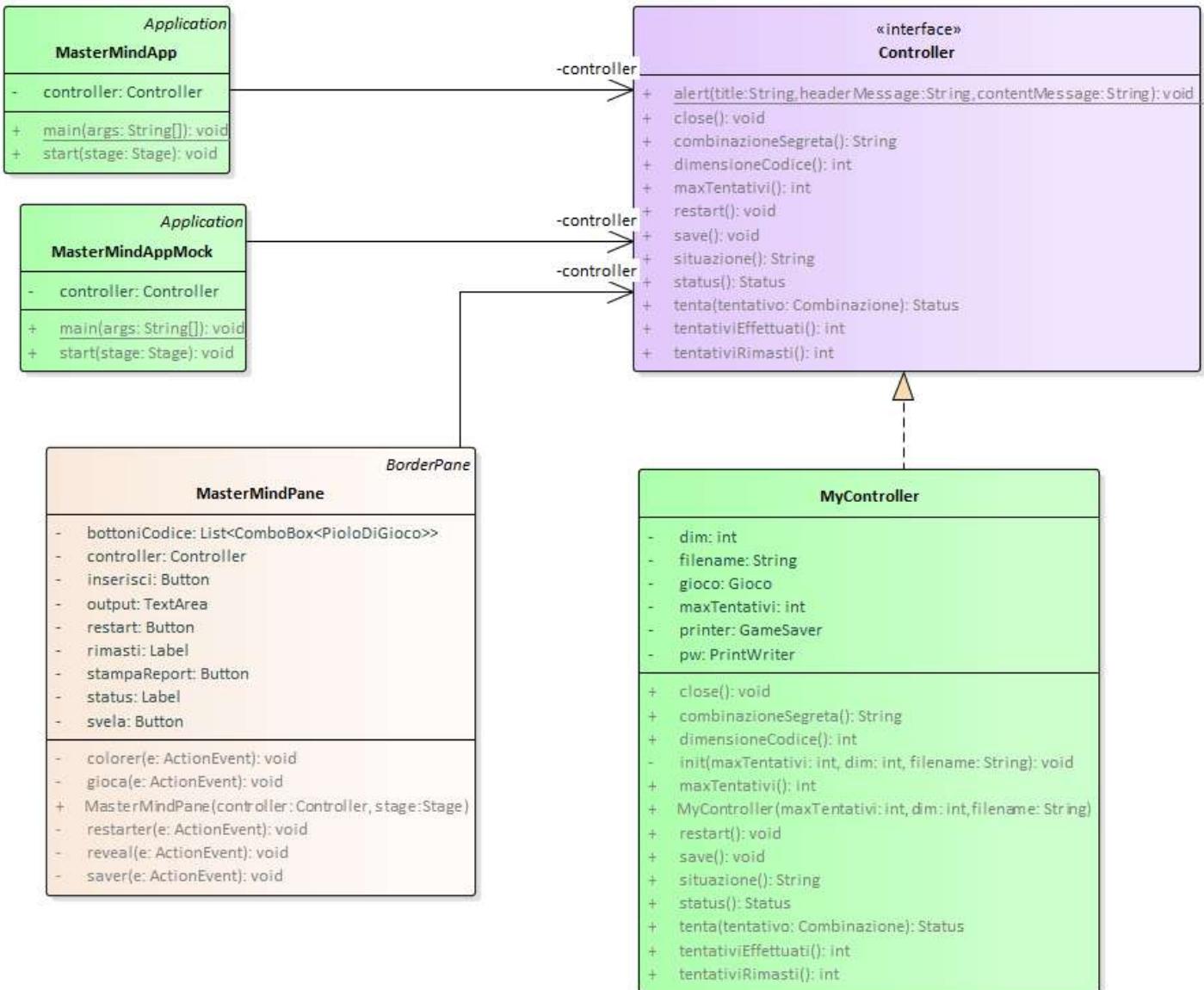
Per effettuare una giocata occorre in primis selezionare la nuova combinazione di tentativo tramite le combo (Fig.2), indi premere il pulsante “Inserisci” (Fig.3): ciò causa il calcolo dalla risposta e la visualizzazione della situazione aggiornata. Analogamente si procedere per le giocate successive (Figg.4,5,6): se, entro il numero massimo di tentativi, il solutore indovina la combinazione segreta, la partita è vinta e lo stato aggiornato opportunamente (Fig.6).

In qualunque momento:

- il tasto “Svela”, sulla sinistra, consente di svelare la combinazione segreta (Fig. 7). Senza però interrompere il gioco, che prosegue comunque regolarmente
- il tasto “Restart”, sulla sinistra, riporta tutto allo stato iniziale, per fare una nuova partita

- il tasto “Stampa report”, sulla sinistra, salva sul file di testo “gameover.txt” lo stato attuale del gioco.

Se si preme il tasto “Inserisci” senza aver prima selezionato tutti i colori, compare la finestra di Alert con opportuno messaggio di errore (Fig. 8).



a) La classe **MasterMindApp** (fornita) contiene il main di partenza dell'applicazione
(la sua variante **MasterMindAppMock** bypassa il reader configurando la GUI con valori di default)

b) La classe **MasterMindPane** (da completare) è una specializzazione di **BorderPane**:

- in alto, contiene una lista di Combobox opportunamente popolate e, sotto di esse, il pulsante “Inserisci”
- a sinistra, i tre pulsanti di controllo
- a destra, l'area di uscita, dove viene costantemente visualizzata la situazione del gioco
- in basso, la barra di stato con le informazioni sullo stato del gioco.

Nello scheletro fornito sono già presenti alcune parti, e precisamente:

- il metodo `colorer`, per la gestione del colore di sfondo delle combo
- il metodo `rearter`, per la gestione del pulsante **Restart**

mentre vanno completati il costruttore e i metodi `gioca`, `reveal` e `saver` che gestiscono rispettivamente i pulsanti **Inserisci**, **Svela** e **Stampa Report**.

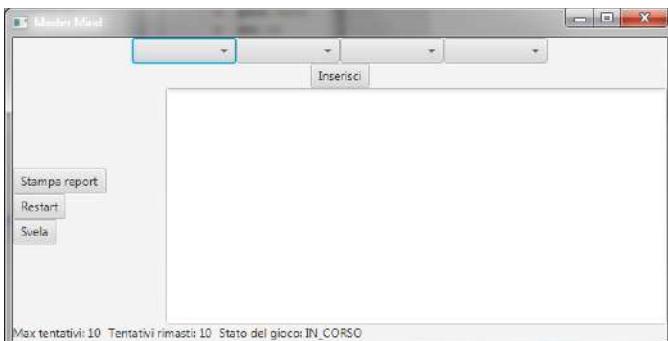


Figura 1



Figura 2

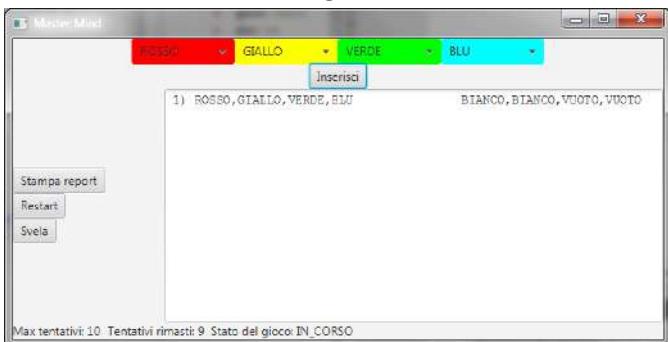


Figura 3

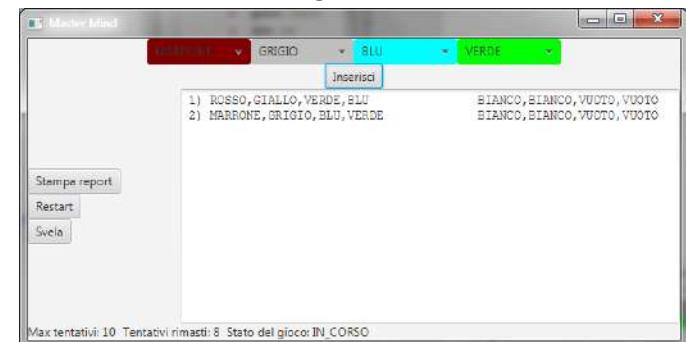


Figura 4



Figura 5



Figura 6



Figura 7

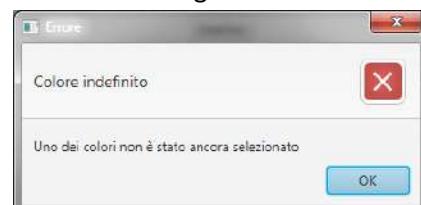


Figura 8

Checklist di consegna

- Hai fatto un **JAR eseguibile**, che contenga cioè l'indicazione del main?
- Hai controllato che **si compili e ci sia tutto?** [NB: non includere il PDF del testo]
- Hai **rinominato IL PROGETTO**, lo ZIP e il JAR esattamente come richiesto?
- Hai **chiamato** la cartella del progetto esattamente come richiesto?
- **Hai fatto un unico file ZIP (NON .7z, rar o altri formati)** contenente l'intero progetto?
In particolare, ti sei assicurato di aver incluso tutti i file .java (e non solo i .class)?
- **Hai consegnato DUE file distinti, ossia lo ZIP col progetto e il JAR eseguibile?**
- Su EOL, hai **premuto** il tasto “CONFERMA” per inviare il tuo elaborato?

ESAME DI FONDAMENTI DI INFORMATICA T-2 del 9/9/2020

Proff. E. Denti – R. Calegari – A. Molesini

Tempo a disposizione: 3 ore

NOME PROGETTO ECLIPSE: CognomeNome-matricola (es. RossiMario-0000123456)

NOME CARTELLA PROGETTO: CognomeNome-matricola (es. RossiMario-0000123456)

NOME ZIP DA CONSEGNARE: CognomeNome-matricola.zip (es. RossiMario-0000123456.zip)

NOME JAR DA CONSEGNARE: CognomeNome-matricola.jar (es. RossiMario-0000123456.jar)

Si devono consegnare DUE FILE: *I'intero progetto Eclipse e il JAR eseguibile*

Si ricorda che compiti non compilabili o palesemente lontani da 18/30 NON SARANNO CORRETTI e causeranno la verbalizzazione del giudizio “RESPINTO”

L’Università di Dentinia, **UniDent**, ha richiesto lo sviluppo di un’applicazione che permetta ai propri studenti di seguire la propria carriera, calcolando anche la media pesata in base ai crediti degli esami sostenuti.

DESCRIZIONE DEL DOMINIO DEL PROBLEMA

Il *piano didattico* del corso di studi è composto da una serie di *attività formative*, caratterizzate ciascuna da nome (che può contenere spazi), *settore scientifico-disciplinare* (una sigla standardizzata), *tipologia* (una lettera da A a F, a cui corrispondono rispettivamente le attività *di base*, *caratterizzanti*, *affini* o *integrative*, *a scelta libera*, per la prova finale e la lingua inglese, altre) e naturalmente dal *numero di crediti* (CFU) ad essa assegnato.

Un *esame* associa a una attività formativa una *data* di effettuazione e un *voto* (o giudizio) ottenuto.

In UniDent non esistono *idoneità*: pertanto, i voti possibili sono soltanto quelli da 18 a 30 e Lode, oltre ai giudizi

Assente, Ritirato, Respinto. Le regole universitarie prevedono inoltre che:

- non si possa ripetere un esame già sostenuto con esito positivo
- non si possa sostenere più volte lo stesso esame (per la stessa attività formativa) nello stesso giorno

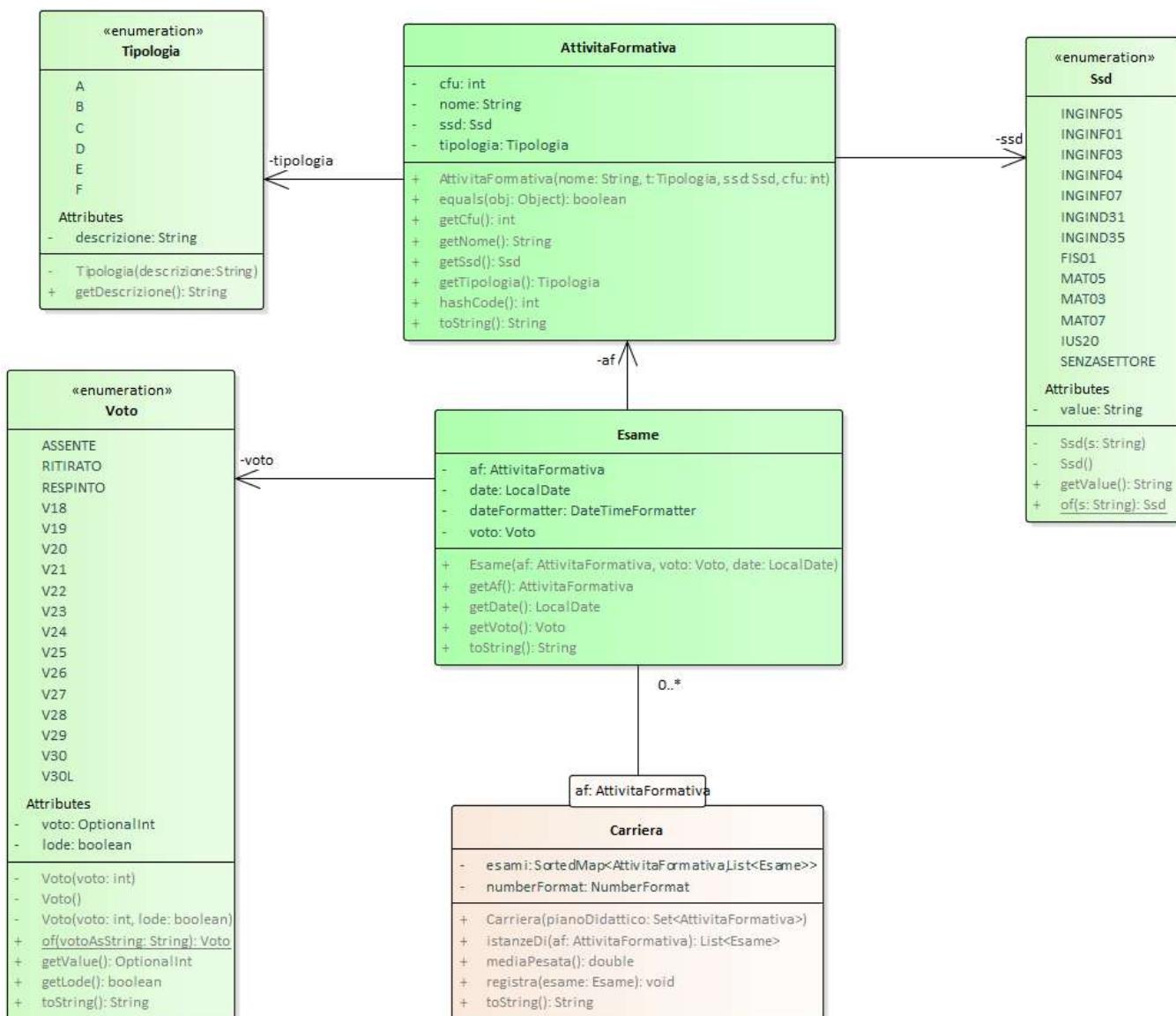
La *carriera* di uno studente comprende l’elenco di tutte le attività formative da sostenere, a ciascuna delle quali sono via via associati gli esami corrispondenti (anche più di uno per ogni attività formativa, nel caso sfortunato in cui l’esito non sia stato inizialmente positivo). Ovviamente, all’inizio la carriera contiene tutte le attività formative ma nessun esame associato.

La *media pesata* degli esami sostenuti è definita come il rapporto fra la somma dei prodotti *voto*cfu* e la somma dei cfu corrispondenti: ovviamente, i giudizi *Assente, Ritirato, Respinto* non fanno media. Nel caso siano presenti solo esami con giudizio, il risultato è per convenzione NaN. La media deve avere *esattamente due cifre decimali*.

ESEMPIO: se uno studente ha sostenuto tre esami, rispettivamente da 12, 9 e 6 cfu, con voti 28, 24 e 30, la media pesata si calcola con la formula $M = (12*28 + 9*24 + 6*30)/(12+9+6) = 27,11/30$.

Il piano didattico del corso di studio *Ingegneria Informatica* di **UniDent** è descritto nel file **PianoDidattico.txt**, il cui formato è riportato più oltre.

TEMPO STIMATO PER SVOLGERE L’INTERO COMPITO: 1h50 – 2h20



SEMANTICA:

- l'enumerativo **Tipologia** (fornito) elenca le possibili tipologie da A ad F, con le corrispondenti descrizioni.
- l'enumerativo **Ssd** (fornito) elenca i possibili settori scientifico-disciplinari. Oltre ai metodi tipici degli enumerativi, è presente il metodo factory `of` che, data la stringa corrispondente (es. "Ing-Inf/05"), ottiene il corrispondente valore dell'enumerativo (che per ovvie ragioni di sintassi Java non può contenere trattini o barrette). Dualmente, il metodo `getValue` restituisce il valore (stringa) corrispondente. Fra le costanti enumerate è inclusa lo speciale valore **SENZASETTORE** (a cui corrisponde la stringa "\t\t"), usato per rappresentare in modo uniforme le attività formative di tipologia E ed F, che non hanno settore.
- l'enumerativo **Voto** (fornito) elenca i possibili voti, inclusi i tre giudizi **Assente**, **Ritirato** e **Respinto**. Anche in questo caso, oltre ai metodi tipici degli enumerativi è presente il metodo factory `of` che, data la stringa corrispondente (es. "22", "30L", "RESPINTO", etc.), ottiene il corrispondente valore dell'enumerativo (che per ovvie ragioni di sintassi Java non può contenere solo cifre). Dualmente, il metodo `getValue` restituisce il valore (intero), se esiste, corrispondente al voto: per questo si utilizza un `OptionalInt`. Il metodo `getLode` restituisce `true` per i soli 30 a cui è associata la lode. Un'opportuna implementazione di `toString` consente di ottenere una rappresentazione stringa coerente al valore enumerativo.

- d) la classe **AttivitaFormativa** (fornita) rappresenta l'attività formativa con le proprietà discusse nel *Dominio del Problema*: sono presenti i classici metodi per recuperare gli elementi, produrre un'opportuna stringa descrittiva, nonché *equals* e *hashcode*.
- e) la classe **Esame** (fornita) rappresenta un esame con le proprietà discusse nel *Dominio del Problema*: anche in questo caso sono presenti i classici metodi per recuperare gli elementi e *toString*.
- f) la classe **Carriera (da realizzare)** rappresenta la carriera dello studente. Internamente mantiene una mappa ordinata per nome dell'attività formativa che associa a ogni attività formativa la lista (inizialmente vuota, poi ordinata in senso crescente per data) degli esami sostenuti per tale attività formativa. Metodi:
- il costruttore riceve il piano didattico, inteso come insieme (set) non ordinato delle attività formative: provvede a inizializzare tutte le strutture dati e le entità accessorie necessarie
 - il metodo *registra(Esame)* consente di registrare l'esame ricevuto come argomento, a condizione che quest'ultimo non abbia data precedente a quelli già registrati per la stessa attività formativa e rispetti le regole UniDent (ossia, non sia già stato superato con esito positivo e non sia già stato sostenuto, con qualunque esito, nello stesso giorno). Se l'esame si riferisce a un'attività formativa non presente in carriera, o non rispetta tali regole, dev'essere lanciata apposita **IllegalArgumentException** con messaggio d'errore specifico e adeguatamente descrittivo.
 - Il metodo *istanzeDi(AttivitaFormativa)* restituisce la lista, eventualmente nulla, degli esami corrispondenti a una data attività formativa
 - Il metodo *mediaPesata* calcola la media pesata degli esami finora sostenuti
 - Il metodo *toString* deve produrre una stringa organizzata su più righe, **secondo il formato visibile nelle figure in fondo**. In particolare, dopo una prima riga di intestazione ("Esami sostenuti:"), dev'essere presente una riga per ogni esame, elencando gli esami sostenuti in ordine alfabetico e, in subordine, per data di effettuazione, secondo lo schema sotto illustrato. **NB: è cruciale che le attività formative per cui non sono stati ancora sostenuti esami siano omesse**. Terminato l'elenco degli esami, dopo una riga vuota, dev'essere riportata la media pesata, secondo lo schema sotto illustrato:

Esami sostenuti: Fondamenti di Informatica T-2 CFU: 6 Data: 10/07/20 Voto: RITIRATO Fondamenti di Informatica T-2 CFU: 6 Data: 10/08/20 Voto: 30
Media pesata: 30/30

REQUISITI:

- Le parole chiave ("CFU:", "Data:", "Voto:") devono essere scritte esattamente come indicato, coi ":" finali
- dopo ogni parola chiave deve sempre essere presente almeno uno spazio
- i dati dovrebbero essere opportunamente incolonnati, come negli esempi sopra.

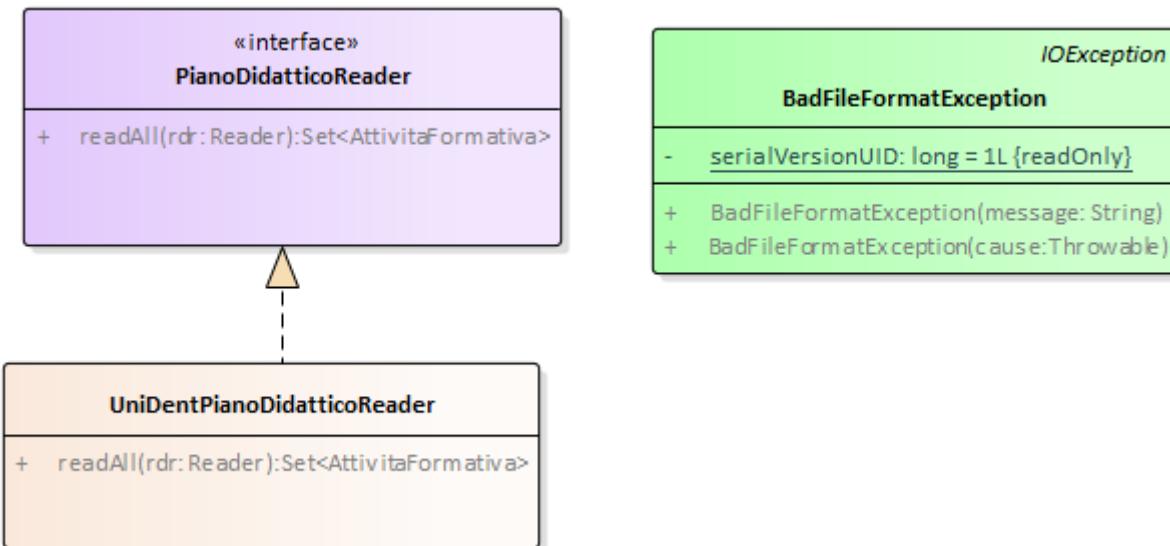
Persistenza (*unident.persistence*)

[TEMPO STIMATO: 40-50 minuti] (punti 9)

Il file di testo specifica una attività formativa per riga. Ogni riga contiene nell'ordine, separati da tabulazioni:

- il codice numerico univoco dell'attività formativa
- il nome dell'attività (può contenere spazi, apostrofi e ogni altro carattere utile)
- il periodo (semestre) di effettuazione (una sigla di uno o più caratteri senza spazi intermedi)
- la tipologia (una lettera fra A ed F)
- solo per le tipologie fra A e D:** il settore scientifico-disciplinare (una sigla standardizzata)
- il numero intero di crediti corrispondenti

27991	ANALISI MATEMATICA T-1	1	A	Mat/05	9
...					
17268	PROVA FINALE	0	E	3	
28072	LABORATORIO DI AMMINISTRAZIONE DI SISTEMI	T	2	F	6
28074	TIROCINIO T		2	F	6
...					
28014	FONDAMENTI DI TELECOMUNICAZIONI T	2	B	Ing-Inf/03	9
28020	SISTEMI OPERATIVI T	2	B	Ing-Inf/05	9
...					



SEMANTICA:

- a) L'eccezione **BadFormatException** (fornita) specializza **IOException** per esprimere l'idea di file formattato in modo scorretto: come tale è a controllo obbligatorio.
- b) L'interfaccia **PianoDidatticoReader** (fornita) dichiara il metodo **readAll**, che legge da un Reader (ricevuto come argomento) i dati di tutte le attività formative elencate nel piano didattico, restituendole sotto forma di **Set** (ovviamente, il piano didattico non può prevedere due attività identiche).
- c) La classe **UniDentPianoDidatticoReader** (**da realizzare**) implementa **PianoDidatticoReader**: non prevede costruttori, si limita a implementare il metodo **readAll** come sopra specificato. In caso di problemi di I/O deve essere propagata l'opportuna **IOException**, mentre in caso di Reader nullo o altri problemi di formato dei file deve essere lanciata una opportuna **BadFormatException**, il cui messaggio dettagli l'accaduto.

In particolare, il reader deve verificare: 1) che ogni riga contenga il giusto numero di elementi, tenendo conto in particolare dell'assenza del settore scientifico-disciplinare nelle attività di tipologia E ed F; 2) che la tipologia sia valida, ossia compresa fra A ed F; 3) che il settore scientifico-disciplinare sia valido, ossia sia correttamente accettato dal metodo factory di **Ssd**; 4) che il numero di crediti sia intero; 5) che non siano presenti attività formative duplicate.

Parte 2

[TEMPO STIMATO: 20-30 minuti]

(punti: 7)

Controller (`unident.controller`)

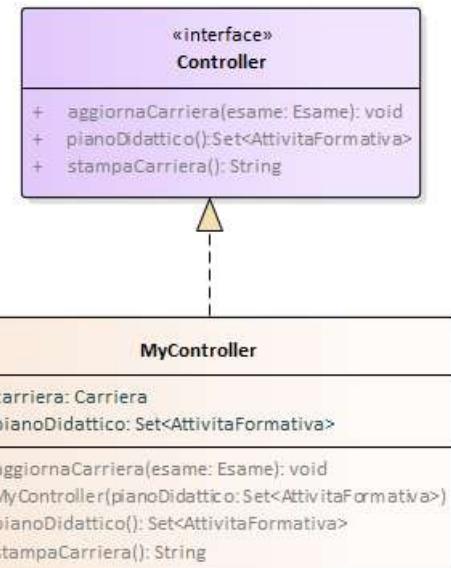
(punti 0)

Il Controller (fornito) è organizzato secondo il diagramma UML nella figura alla pagina seguente.

SEMANTICA:

- a) L'interfaccia **Controller** (fornita) dichiara i metodi *pianoDidattico*, *aggiornaCarriera* e *stampaCarriera*.
- b) La classe **MyController** (fornita) implementa tale interfaccia, fornendo:

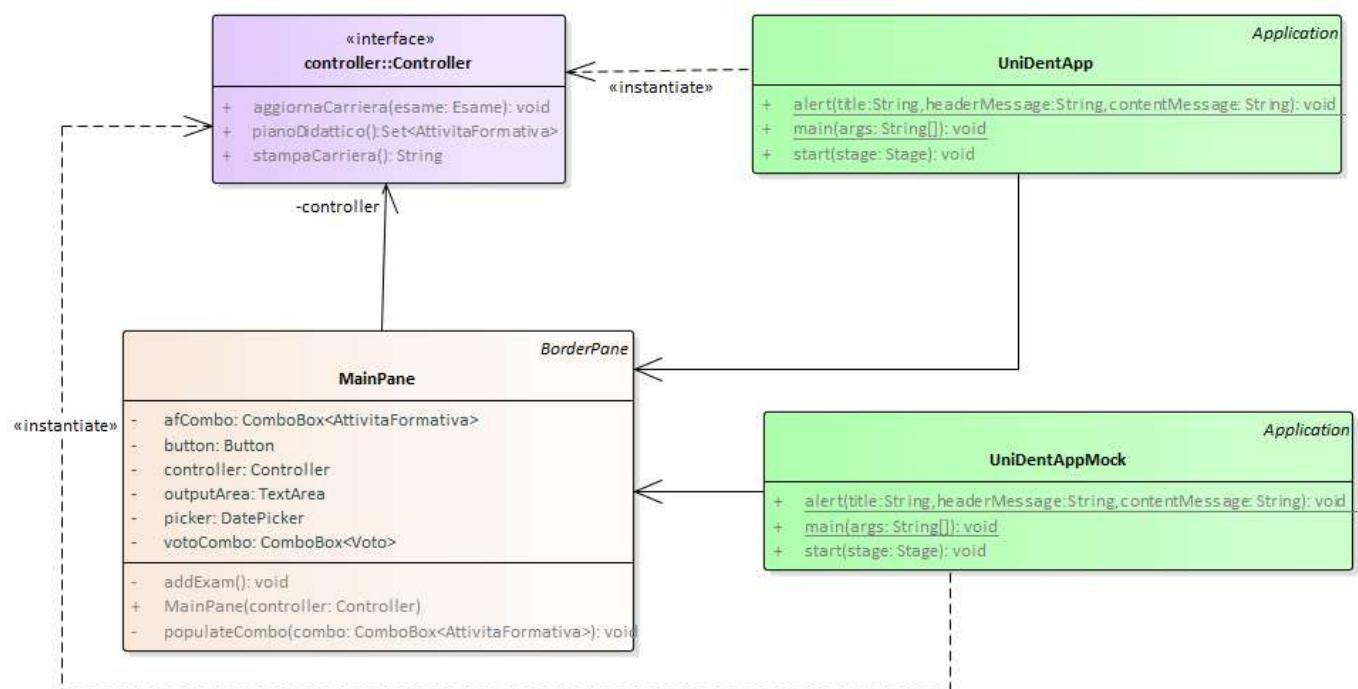
- il costruttore che, dall'insieme delle attività formative (piano didattico), costruisce e inizializza la **Carriera** dello studente;
- implementa i tre metodi delegando il lavoro, per quanto necessario, all'istanza di **Carriera** interna.



Interfaccia utente (unident.ui)

[TEMPO STIMATO: 20-30 minuti] (punti 7)

L'interfaccia utente è illustrata nelle figure seguenti e segue il modello sotto illustrato:



La classe **UniDentApp** (fornita) costituisce l'applicazione JavaFX che si occupa di aprire i file, creare il controller e incorporare il **MainPane**. Per consentire di collaudare la GUI anche in assenza / in caso di malfunzionamento della parte di persistenza, è possibile avviare l'applicazione mediante la classe **UniDentAppMock**.

Entrambe le classi contengono anche il **metodo statico ausiliario *alert***, utile per mostrare avvisi all'utente.

Il MainPane è fornito parzialmente realizzato: è presente la parte strutturale, mentre manca la parte di popolamento combo e gestione degli eventi.

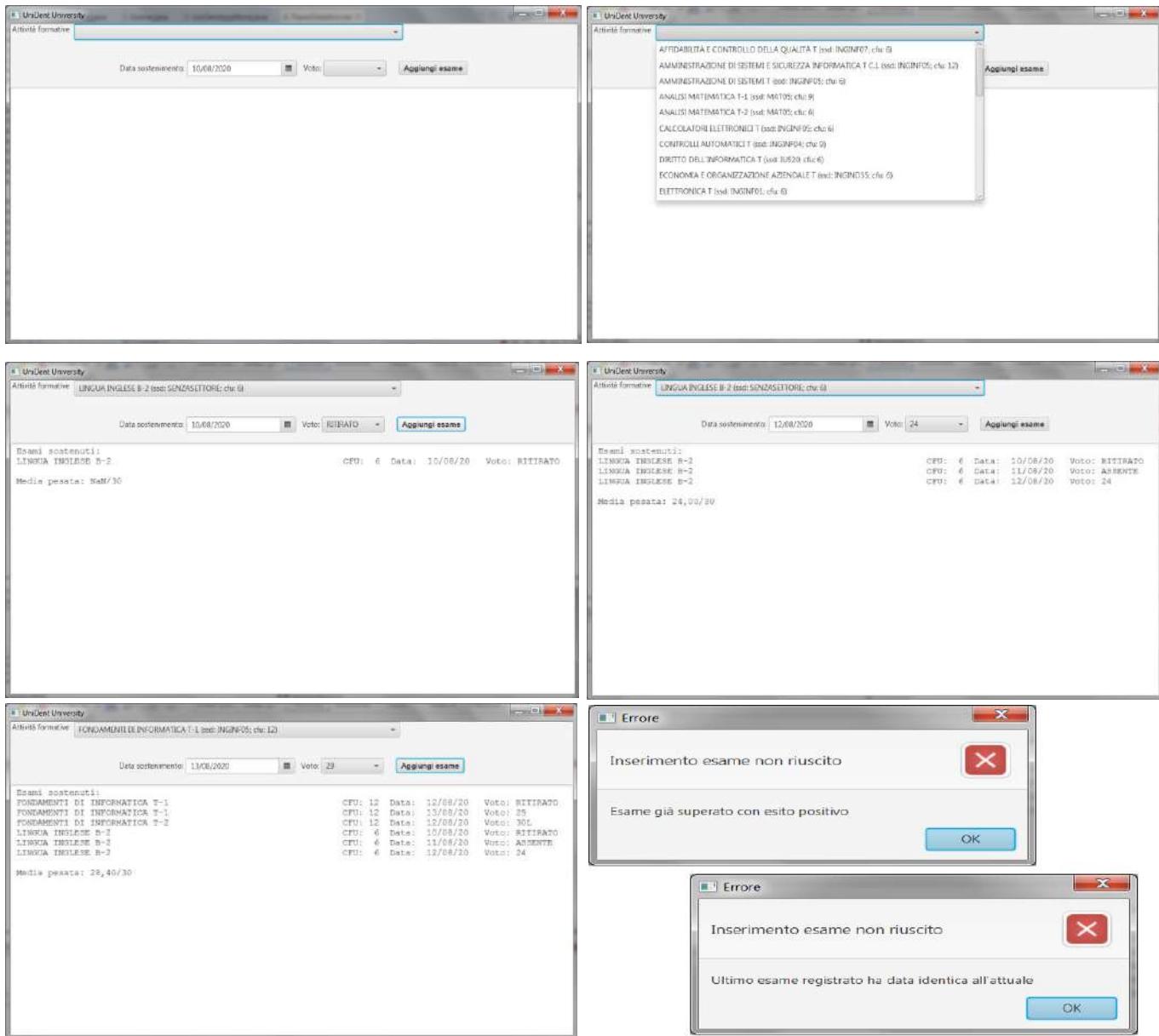
La classe **MainPane** (da completare) estende **BorderPane** e prevede:

- 1) in alto, la **ComboBox** da popolare con l'*elenco alfabetico ordinato* delle attività formative: inizialmente, nessun elemento dev'essere preselezionato (Fig. 1)
- 2) sotto, un **Datepicker** per scegliere la data dell'esame (inizializzato alla data corrente, Fig. 1), una **ComboBox** coi possibili voti (senza alcun elemento preselezionato, Fig. 1) e infine un **Button** "AGGIUNGI ESAME" per inserire l'esame in carriera

- 3) in basso, una **TextArea** che mostra gli esami finora sostenuti, in ordine alfabetico e in subordine per data crescente (Figg. 3, 4 e 5) seguiti dalla media pesata, su due cifre decimali; quando la media non è calcolabile, dev'essere mostrato NaN (Fig. 3). Al fine di ottenere una presentazione gradevolmente incolonnata, è richiesto l'uso di un font non proporzionale, come il Courier New (vedere figure).

La **parte da completare** comprende l'uso e/o l'implementazione dei seguenti metodi:

- 1) **populateCombo**, che popola la combo con *l'elenco alfabetico ordinato di tutte le attività formative* (Fig. 2)
- 2) **addExam**, da chiamare in risposta all'evento di pressione del **Button**, cerca di inserire l'esame in carriera; se ciò non risulta possibile (perché il controller emette eccezione), dev'essere emesso apposito messaggio di errore tramite **alert** (Fig. 6).



Cose da ricordare

- salva costantemente il tuo lavoro: l'informatica a volte può essere “subdolamente ostile”..
- in particolare: se ora compila e stai per fare modifiche, salva la versione attuale (non si sa mai)

Checklist di consegna

- Hai fatto un **JAR eseguibile**, che contenga cioè l'indicazione del main?
- Hai controllato che **si compilii e ci sia tutto**? [NB: non includere il PDF del testo]

- Hai **rinominato** IL PROGETTO, lo ZIP e il JAR esattamente come richiesto?
- Hai **chiamato** la cartella del progetto esattamente come richiesto?
- Hai fatto un **unico file ZIP (NON .7z, rar o altri formati)** contenente l'intero progetto?
In particolare, ti sei assicurato di aver incluso tutti i file .java (e non solo i .class)?
- **Hai consegnato DUE file distinti, ossia lo ZIP col progetto e il JAR esequibile?**
- Su EOL, hai **premuto** il tasto “CONFERMA” per inviare il tuo elaborato?

ESAME DI FONDAMENTI DI INFORMATICA T-2 del 23/7/2020

Proff. E. Denti – R. Calegari – A. Molesini

Tempo a disposizione: 3 ore

NOME PROGETTO ECLIPSE: CognomeNome-matricola (es. RossiMario-0000123456)

NOME CARTELLA PROGETTO: CognomeNome-matricola (es. RossiMario-0000123456)

NOME ZIP DA CONSEGNARE: CognomeNome-matricola.zip (es. RossiMario-0000123456.zip)

NOME JAR DA CONSEGNARE: CognomeNome-matricola.jar (es. RossiMario-0000123456.jar)

Si devono consegnare DUE FILE: *I'intero progetto Eclipse e il JAR eseguibile*

Si ricorda che compiti non compilabili o palesemente lontani da 18/30 NON SARANNO CORRETTI e causeranno la verbalizzazione del giudizio “RESPINTO”



Le **Rapidissime Ferrovie di Dentinia** (RFD), che gestiscono una antica rete ferroviaria a vapore fra alcune città, hanno chiesto lo sviluppo di una nuova funzionalità nella pre-esistente applicazione per la ricerca dei percorsi fra stazioni della loro rete: oltre ai percorsi possibili con relativo chilometraggio, desiderano che sia calcolato anche il *tempo di percorrenza*

DESCRIZIONE DEL DOMINIO DEL PROBLEMA

Ogni *linea ferroviaria* è descritta da una sequenza di *stazioni*, caratterizzate ciascuna dal nome del luogo (che può contenere spazi) e dalla *progressiva chilometrica* (ossia, la distanza dal capolinea iniziale). In alcune stazioni, dette *punti di interscambio (hub)*, si intersecano due o più linee: ciò consente di offrire ai viaggiatori anche soluzioni di viaggio con *al più un cambio intermedio*. Non si considerano soluzioni con due o più cambi, perché scomode.

Si dice segmento una tratta orientata fra due qualsiasi stazioni *della stessa linea*. Un segmento è semplice se non esistono al suo interno stazioni intermedie, ossia non può essere ulteriormente spezzato in sotto-segmenti.

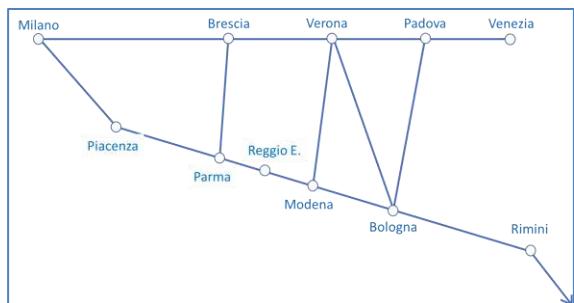
ESEMPIO: il segmento Parma-Milano, che è cosa distinta dal segmento Milano-Parma (che descriverebbe un percorso nella direzione opposta), *non è semplice*, essendo presenti al suo interno molte stazioni intermedie.

Un *percorso* fra due stazioni si dice *diretto* se non prevede cambi (ed è quindi costituito da un unico segmento), *indiretto* altrimenti: in questo secondo caso il percorso è costituito da *due o più segmenti consecutivi* (ossia tali che la stazione terminale di ciascuno coincide con quella di inizio del successivo).

ESEMPIO: fra Parma e Milano, nella rete in figura, sono possibili due percorsi: uno diretto (Parma-Milano sulla linea Bologna-Milano) e uno indiretto (Parma-Brescia sulla linea omonima + Brescia-Milano sulla linea Venezia-Milano).

Ovviamente, la lunghezza di un percorso è pari alla somma delle lunghezze dei segmenti che lo costituiscono, a loro volta pari alla differenza fra le progressive chilometriche delle rispettive stazioni di estremità.

ESEMPIO: il percorso fra Lodi (km 183,80 della linea Bologna-Milano) e Rimini (km 111,04 della linea Bologna-Lecce) è di 294,84 km e comprende due segmenti (Lodi-Bologna e Bologna-Rimini).

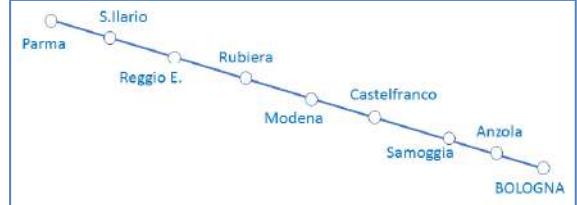


Per calcolare i tempi di percorrenza occorre conoscere la velocità ammessa sui singoli tratti di linea. Per semplicità, si suppone che essa sia costante all'interno di un singolo segmento semplice: si conviene perciò che ogni stazione sia associata alla velocità da mantenere nel tratto di linea fino alla stazione successiva.

ESEMPIO: sulla linea Bologna-Milano, il primo segmento semplice (Bologna Centrale-Anzola dell'Emilia) potrebbe avere una velocità ammessa di 120 km/h, mentre il successivo potrebbe averne una diversa (es. 140 km/h) e così via, di stazione in stazione, fino a Milano Centrale (termine linea).

Il tempo di percorrenza di un dato percorso si ottiene quindi sommmando i tempi di percorrenza dei segmenti semplici che lo compongono, così da considerare tutti gli eventuali cambiamenti di velocità lungo la linea.

ESEMPIO: nel caso a lato, il tempo di percorrenza della tratta Bologna Centrale-Modena si ottiene sommando i tempi parziali dei segmenti semplici Bologna Centrale-Anzola, Anzola-Samoggia, Samoggia-Castelfranco e Castelfranco-Modena, a cui possono essere associate velocità diverse: NON prendendo per buona la velocità iniziale di Bologna fino a Modena!



CONVENZIONE: poiché una linea è descritta nel verso dal capolinea iniziale (progressiva km 0,0) al capolinea finale, le stazioni costituiscono una *sequenza ordinata in ordine di progressiva chilometrica crescente*. Di conseguenza, anche le velocità ammesse si intendono “da quella stazione alla successiva” nello stesso verso.

ESEMPIO: la linea Bologna-Milano è descritta partendo da Bologna Centrale (progressiva km 0,0) e terminando a Milano Centrale (progressiva km 216,18): perciò, la velocità associata alla stazione di Bologna Centrale si intende valida fino ad Anzola, quella associata ad Anzola si assume valida fino a Samoggia, e così via.

Di ciò bisogna tenere conto quando si calcolano i tempi di percorrenza: la velocità da assumere nel tratto da A a B, infatti, *potrebbe non essere quella associata alla stazione A, ma quella associata alla stazione B*, se la linea è descritta nel verso opposto, “da B verso A” (ossia la progressiva chilometrica di A è maggiore di quella di B).

ESEMPIO: la velocità ammessa nella tratta Modena (progressiva km 36,93) - Castelfranco (progressiva km 25,01) della linea Bologna-Milano **non è** quella associata alla stazione di Modena, ma quella associata alla stazione di Castelfranco, appunto perché la linea Bologna-Milano è descritta partendo da Bologna, non da Milano. La velocità associata a Modena è quella valida fino alla “sua” stazione “successiva”, ossia fino a Rubiera (progressiva 49,59); e così via proseguendo (quella di Rubiera copre il segmento fino a Reggio Emilia, etc.).

La rete delle **Rapidissime Ferrovie di Dentinia**, costituita da *sette linee*, è quella illustrata sopra: ogni linea è descritta in un singolo file di testo, di estensione “.line”, il cui formato è riportato più oltre.

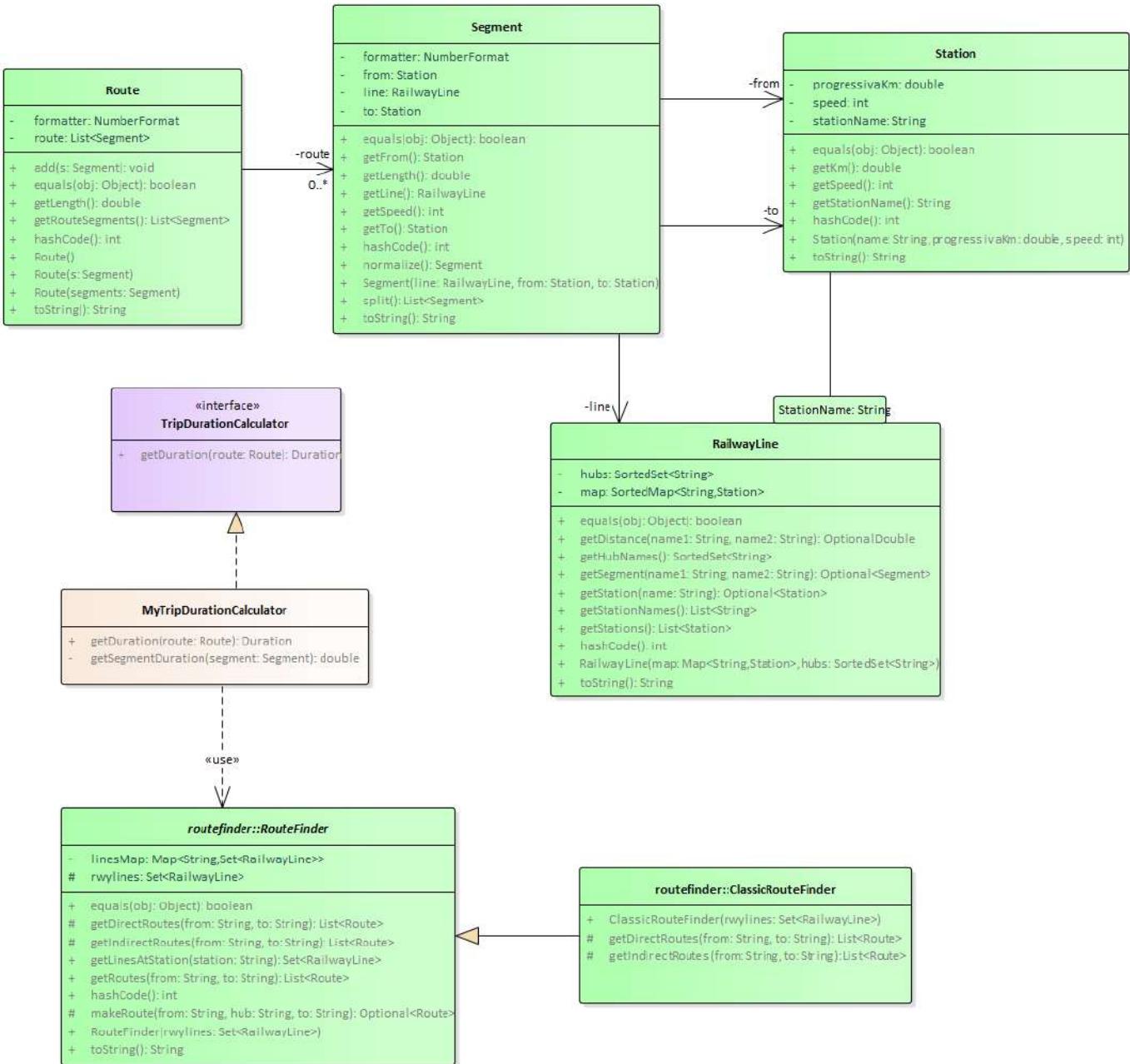
TEMPO STIMATO PER SVOLGERE L'INTERO COMPITO: 1h50 – 2h20

Parte 1

(punti: 20)

Modello dei dati (package rfd.model)

[TEMPO STIMATO: 20-30 minuti] (punti: 9)



SEMANTICA:

- la classe **Station** (fornita) rappresenta una stazione, caratterizzata da nome, progressiva chilometrica (numero reale in km) e velocità ammessa (numero intero in km/h) “fino alla stazione successiva” della linea di cui fa parte.
- la classe **Segment** (fornita) rappresenta un segmento di una data *linea* (**RailwayLine**) compreso fra due **Station** distinte (anche non adiacenti): sono presenti i classici metodi per recuperare gli elementi, produrre un’opportuna stringa descrittiva, nonché **equals** e **hashcode**. Da segnalare i seguenti metodi di particolare interesse:
 - normalize**: produce il corrispondente segmento “normalizzato”, orientato nel verso della linea
ESEMPIO: normalizzando il segmento Modena-Bologna viene restituito il segmento Bologna-Modena, che è orientato secondo il verso crescente della linea Bologna-Milano (ovviamente, se il segmento iniziale è già normalizzato, viene restituito intoccato)

- *split*: suddivide un segmento nella lista dei suoi segmenti semplici, elencati *sempre e comunque* in ordine normalizzato (anche se il segmento iniziale non lo era).
ESEMPIO: spaccando il segmento Modena-Bologna verrà restituita la lista di segmenti semplici [Bologna-Anzola, Anzola-Samoggia, Samoggia-Castelfranco, Castelfranco-Modena], rispettando cioè *sempre e comunque* l'ordinamento della linea Bologna-Milano.
- c) la classe **RailwayLine** (fornita) rappresenta una linea ferroviaria intesa come insieme di **Station**, che il costruttore si aspetta di ricevere sotto forma di mappa indicizzata per nome della stazione. Il secondo argomento del costruttore è l'insieme ordinato dei *nomi* delle stazioni della linea che possono fungere da *punti di interscambio*. La classe offre svariati metodi per:
- recuperare la lista dei nomi delle stazioni (metodo *getStationNames*) / delle stazioni come oggetti (metodo *getStations*), nonché l'insieme ordinato dei nomi degli hub (metodo *getHubNames*)
 - recuperare una **Station**, se esiste, dato il suo nome (metodo *getStation*): restituisce un optional
 - calcolare la distanza fra due **Station** distinte, se esistono (metodo *getDistance*)
 - recuperare il **Segment**, se esiste, corrispondente alla tratta compresa fra due **Station** distinte (metodo *getSegment*)
 - emettere una stringa descrittiva (metodo *toString*)
- d) la classe **Route** (fornita) rappresenta un *percorso* per un viaggiatore, inteso come *sequenza di Segment*. I costruttori consentono di costruire la **Route** in tre situazioni tipiche (inizialmente vuota, inizialmente con un solo segmento, o a partire da un numero variabile di segmenti): è comunque possibile aggiungere via via altri segmenti *in coda* alla sequenza, tramite il metodo *add*. Opportuni accessori consentono di recuperare i vari elementi: anche in questo caso sono presenti *equals*, *hashCode* e *toString*.
- e) La classe astratta **RouteFinder** (fornita sotto forma di libreria JAR, senza sorgente) rappresenta l'entità in grado di cercare percorsi fra due città date (metodo *getRoutes*): a tal fine riceve l'insieme di **RailwayLine** che rappresenta la rete ferroviaria. Il metodo *getLinesAtStation* restituisce l'insieme – eventualmente vuoto - delle **RailwayLine** che servono una data stazione. I due metodi protetti *getDirectRoutes* e *getIndirectRoutes* (stessi argomenti di *getRoutes*) sono destinati a essere implementati da sottoclassi concrete: in questa classe la loro implementazione si limita a lanciare **UnsupportedOperationException** a fini di test. Infine, il metodo protetto *makeRoute* costruisce, se esiste, il percorso indiretto fra due città date passando per l'hub intermedio specificato.
- f) La classe concreta **ClassicRouteFinder** (fornita anch'essa sotto forma di libreria JAR, senza sorgente) estende opportunamente **RouteFinder**.
- g) L'interfaccia **TripDurationCalculator** (fornita) dichiara il metodo *getDuration* che calcola la durata di un percorso (**Route**).
- h) la classe **MyTripDurationCalculator** (**da realizzare**) concretizza tale interfaccia implementando **(punti: 9)**
getDuration nel modo spiegato nel *Dominio del Problema*, ovvero: **[TEMPO STIMATO: 20-30 minuti]**
- recupera i segmenti del percorso
 - li normalizza
 - li spartisce nei segmenti semplici che li compongono
 - utilizza questi ultimi per il calcolo del tempo di percorrenza (tempo = spazio / velocità), assumendo come velocità quella ammessa nel segmento semplice considerato.

Persistenza (rfd.persistence)

[TEMPO STIMATO: 50-60 minuti] (punti 11)

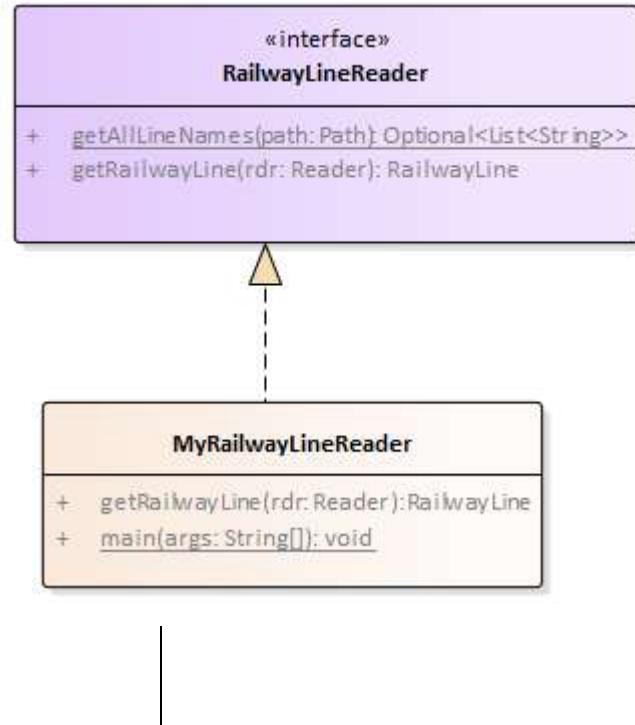
Sono presenti vari file di testo con estensione “.line”, uno per ogni linea ferroviaria, tutti formattati secondo lo stesso schema. Le righe sono tutte di identica lunghezza (è una specifica) e contengono nell'ordine:

- nei primi 8 caratteri, la *progressiva chilometrica* (un numero reale con due cifre decimali, formattato secondo le convenzioni italiane), con eventuali spazi davanti e/o dietro;

- in fondo, preceduta da uno spazio, la velocità ammessa (un numero intero)
- in mezzo, il nome della stazione (che può contenere spazi o qualunque altro carattere) seguito, per le stazioni che fungono da interscambio con altre linee, dalla parola "HUB", scritta con qualunque mix di maiuscole e/o minuscole.

ATTENZIONE: non è garantito che prima della parola "HUB" vi siano spazi o altri separatori: è anche possibile che siano tutti consecutivi (ad esempio, "Torre CencelloHUB").

0,00	Bologna Centrale	HUB	125
2,63	Bologna San Vitale		170
6,55	San Lazzaro di Savena		170
13,01	Ozzano dell'Emilia		170
...			
96,22	Savignano sul Rubicone		140
101,27	Santarcangelo di Romagna		140
...			



SEMANTICA:

- a) L'interfaccia **RailwayLineReader** (fornita) dichiara il metodo `getRailwayLine`, che legge da un Reader (ricevuto come argomento) i dati di una singola linea ferroviaria, restituendo la corrispondente **RailwayLine**. NB: l'interfaccia contiene anche il metodo statico `getAllLineNames` che restituisce la lista dei nomi di file di tipo ".line" (ossia, quelli che descrivono le linee ferroviarie) contenuti nella cartella passata come argomento. Tale metodo è invocato automaticamente dal main dell'applicazione (vedere Parte 2).
 - b) La classe **MyRailwayLineReader** (**da realizzare**) implementa **RailwayLineReader**: non prevede costruttori, si limita a implementare il metodo `getRailwayLine` come sopra specificato. In caso di problemi di I/O deve essere propagata l'opportuna `IOException`, mentre in caso di Reader nullo o altri problemi di formato dei file deve essere lanciata una opportuna `IllegalArgumentException`, il cui messaggio dettagli l'accaduto. In particolare, il reader deve verificare: 1) che la progressiva chilometrica abbia la forma di numero reale separato da virgola; 3) che la velocità sia un numero intero; 4) che il nome della stazione non sia vuoto né consista della sola parola "HUB".
- SUGGERIMENTO:** potrebbe essere comodo, in questo caso, sfruttare i metodi della classe `String...`

Parte 2

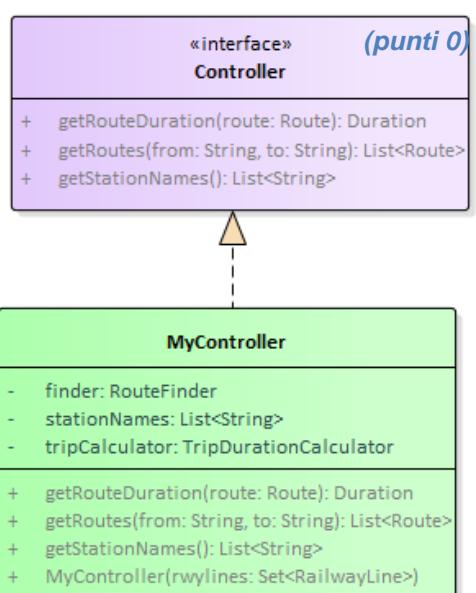
[**TEMPO STIMATO: 40-50 minuti**] (**punti: 10**)

Controller (rfd.controller)

Il Controller (fornito) è organizzato secondo il diagramma UML in figura.

SEMANTICA:

- a) L'interfaccia **Controller** (fornita) dichiara i metodi `getStationNames`, `getRoutes` e `getRouteDuration`.
- b) La classe **MyController** (fornita) implementa tale interfaccia, fornendo:
 - il costruttore che, dall'insieme delle linee societarie, estrae l'elenco dei nomi delle stazioni, quindi crea internamente il **RouteFinder** e il **TripDurationCalculator** necessari;



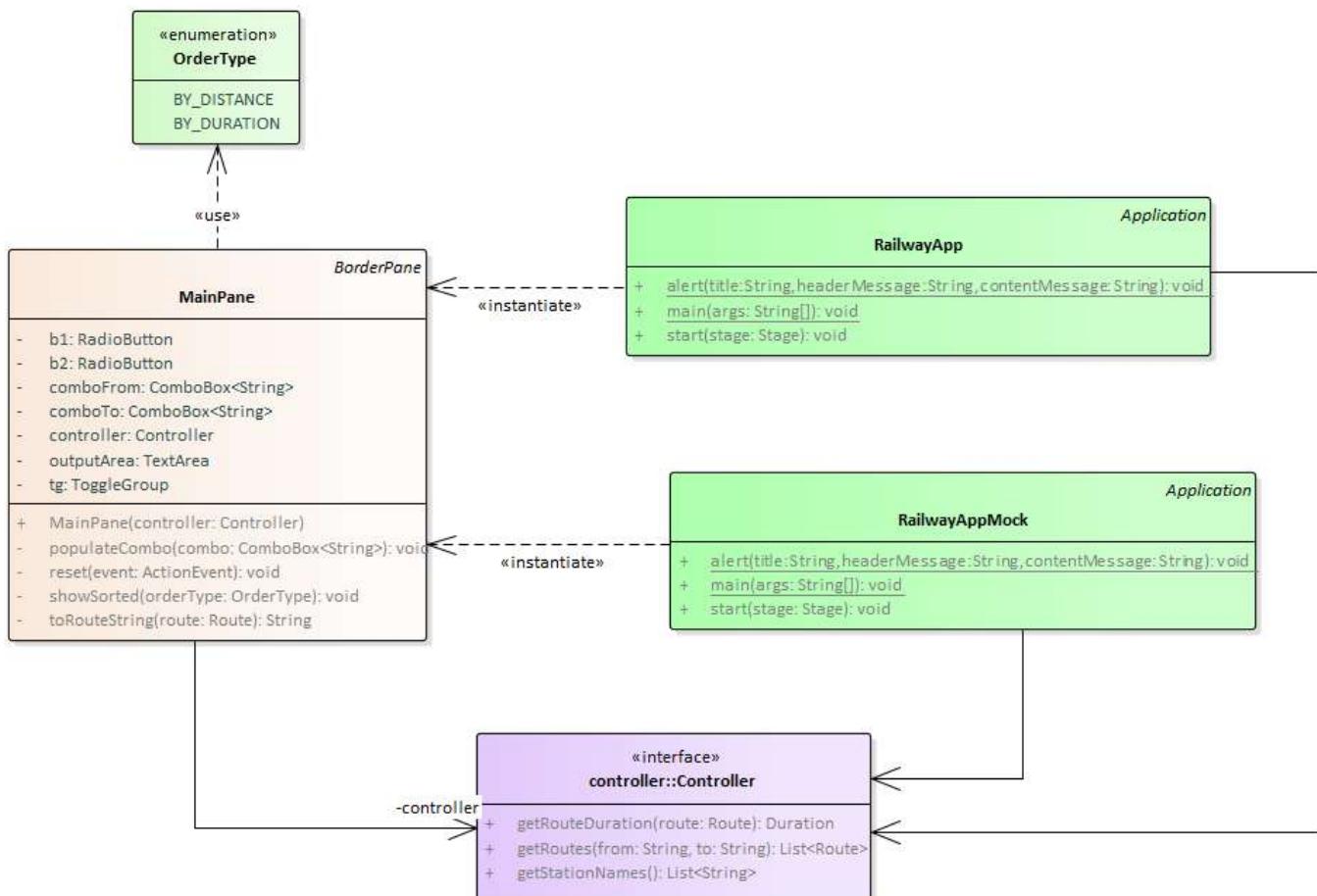
- implementa i tre metodi delegando il lavoro ai rispettivi *finder* e *trip duration calculator*.

NB: la lista dei nomi di stazione restituita da `getStationNames` è già ordinata alfabeticamente.

Interfaccia utente (rfd.ui)

[TEMPO STIMATO: 40-50 minuti] (punti 10)

L'interfaccia utente è illustrata nelle figure seguenti e segue il modello sotto illustrato:



La classe **RailwayApp** (fornita) costituisce l'applicazione JavaFX che si occupa di aprire i file, creare il controller e incorporare il **MainPane**. Per consentire di collaudare la GUI anche in assenza / in caso di malfunzionamento della parte di persistenza, è possibile avviare l'applicazione mediante la classe **RailwayAppMock**.

Entrambe le classi contengono anche il **metodo statico ausiliario alert**, utile per mostrare avvisi all'utente.

Il MainPane è fornito parzialmente realizzato: è presente la parte strutturale, mentre manca la parte di popolamento combo e gestione degli eventi.

La classe **MainPane (da completare)** estende **BorderPane** e prevede:

- 1) in alto, due **ComboBox** da popolare con *l'elenco alfabetico ordinato di tutte le stazioni di tutte le linee*
- 2) sotto, due **RadioButton** per scegliere il criterio di ordinamento dei percorsi, per distanza o per durata
- 3) in basso, una **TextArea** che mostra i percorsi (**Route**) trovati, nell'ordine richiesto.

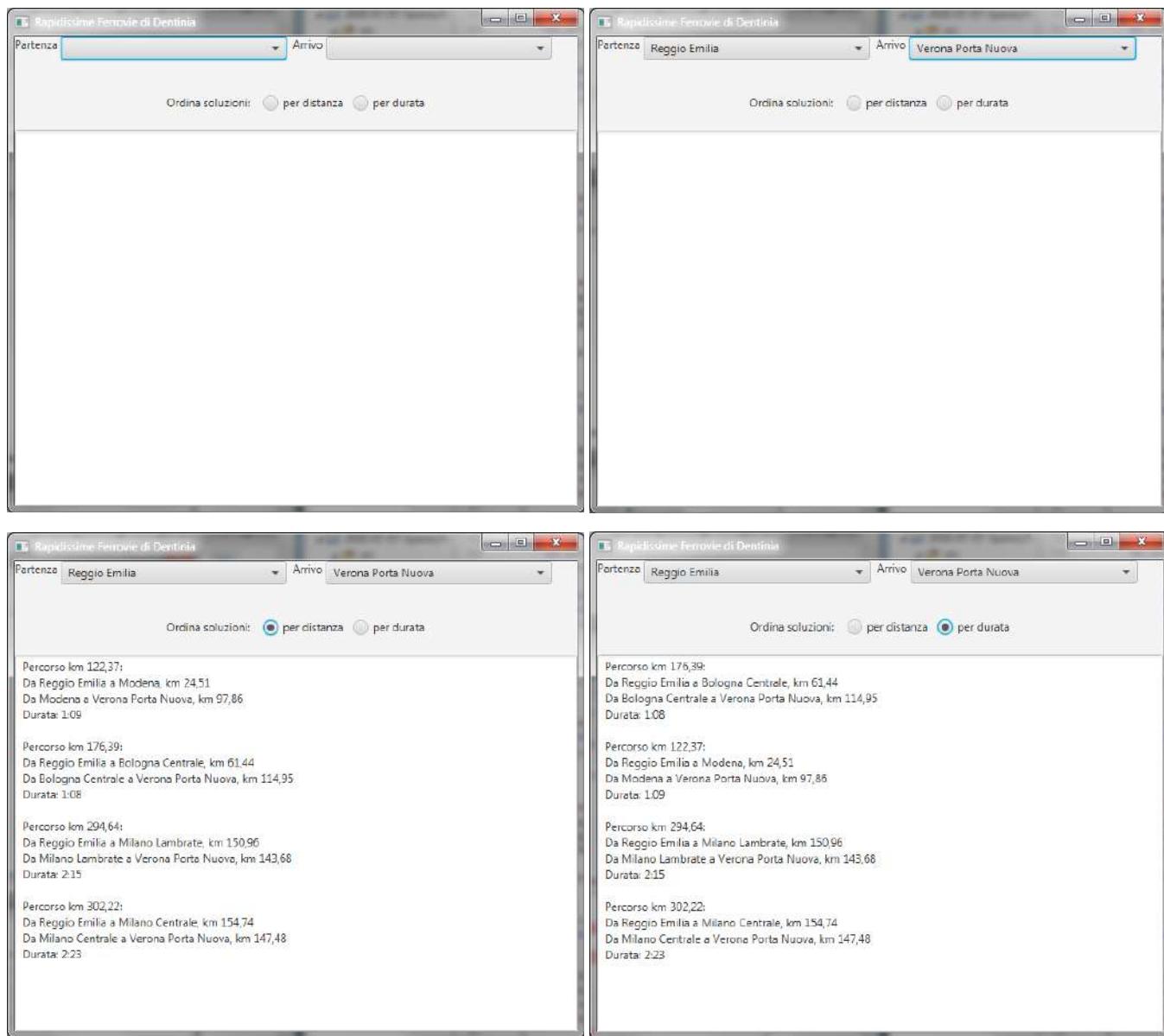
La **parte da completare** comprende l'uso e/o l'implementazione dei seguenti metodi:

- 1) **populateCombo**, che popola la combo con *l'elenco alfabetico ordinato di tutte le stazioni*
- 2) **reset**, da chiamare in risposta all'evento di selezione delle combo, che riporta la GUI allo stato iniziale, svuotando la **TextArea** e riportando i due **RadioButton** allo stato iniziale di nessun pulsante selezionato
- 3) **showSorted**, da chiamare in risposta all'evento di pressione dei due **RadioButton**, ha come argomento di ingresso un valore dell'enumerativo **OrderType** che permette di discriminare l'ordinamento desiderato (per durata o per distanza chilometrica): il metodo reagisce all'evento ottenendo e mostrando l'elenco

dei percorsi ordinandolo col giusto comparatore, secondo il criterio di confronto richiesto.

NB: si avvale del metodo ausiliario `toRouteString` per mostrare il percorso con la durata ben formattata.

- 4) `toRouteString`, data una **Route**, produce una stringa nel previsto formato di uscita, concatenando alla `toString` di **Route**, su riga separata, l'indicazione della durata del percorso nel formato ore:minuti, coi minuti su esattamente due cifre.



Cose da ricordare

- salva costantemente il tuo lavoro: l'informatica a volte può essere “subdolamente ostile”..
- in particolare: se ora compila e stai per fare modifiche, salva la versione attuale (non si sa mai)

Checklist di consegna

- Hai fatto un **JAR eseguibile**, che contenga cioè l'indicazione del main?
- Hai controllato che **si compil** e **ci sia tutto**? [NB: non includere il PDF del testo]
- Hai **rinominato IL PROGETTO**, lo ZIP e il JAR esattamente come richiesto?
- Hai **chiamato** la cartella del progetto esattamente come richiesto?
- **Hai fatto un unico file ZIP (NON .7z, rar o altri formati)** contenente l'intero progetto?
In particolare, ti sei assicurato di aver incluso tutti i file .java (e non solo i .class)?
- **Hai consegnato DUE file distinti, ossia lo ZIP col progetto e il JAR eseguibile?**
- Su EOL, hai premuto il tasto “CONFERMA” per inviare il tuo elaborato?

ESAME DI FONDAMENTI DI INFORMATICA T-2 del 7/7/2020

Proff. E. Denti – R. Calegari – A. Molesini

Tempo a disposizione: 3 ore

NOME PROGETTO ECLIPSE: CognomeNome-matricola (es. RossiMario-0000123456)

NOME CARTELLA PROGETTO: CognomeNome-matricola (es. RossiMario-0000123456)

NOME ZIP DA CONSEGNARE: CognomeNome-matricola.zip (es. RossiMario-0000123456.zip)

NOME JAR DA CONSEGNARE: CognomeNome-matricola.jar (es. RossiMario-0000123456.jar)

Si devono consegnare DUE FILE: *l'intero progetto Eclipse e il JAR eseguibile*

Si ricorda che compiti non compilabili o palesemente lontani da 18/30 NON SARANNO CORRETTI e causeranno la verbalizzazione del giudizio “RESPINTO”

È stata richiesta una app per agevolare la soluzione di **Sudoku**. L'obiettivo non è giocare “contro” il computer o “farselo risolvere” dal computer, ma semplicemente predisporre un supporto informatico che consenta al solutore di inserire i vari numeri senza dover ricorrere a carta e matita.

DESCRIZIONE DEL DOMINIO DEL PROBLEMA

Sudoku è un gioco di logica e abilità costituito da una *griglia* di 9×9 celle, ciascuna delle quali può contenere un numero da 1 a 9 o essere vuota; la griglia è suddivisa in:

- 9 righe orizzontali
- 9 colonne verticali
- 9 "sotto-griglie" di 3×3 celle contigue.

Queste sotto-griglie sono delimitate da bordi in neretto e chiamate *regioni*. Le griglie proposte al giocatore hanno da 20 a 35 celle già pre-inizializzate con un numero.

Lo scopo del gioco è riempire le caselle bianche con i “giusti” valori da 1 a 9 in modo tale che in ogni riga, in ogni colonna e in ogni regione siano presenti tutte le cifre da 1 a 9, ma senza ripetizioni.

ESEMPIO (foto): nello schema iniziale in alto, si consideri la terza sotto-griglia (quella in alto a destra): risulta ovvio che il numero 5 vada collocato a sinistra del 6 già presente, in quanto è l'unica scelta possibile data la collocazione degli altri 5 nelle prime due righe in alto e nell'ultima colonna a destra.

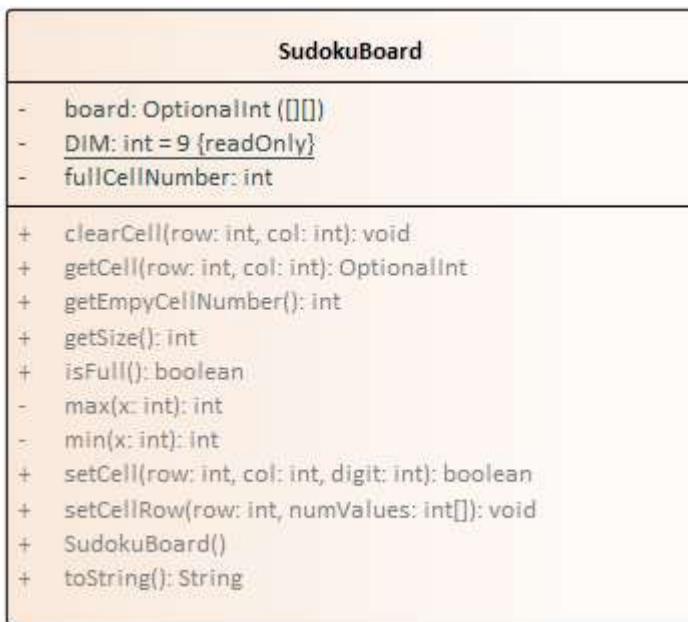
Per curiosità, l'immagine a lato mostra il corrispondente schema risolto.

5	3			7				
6			1	9	5			
	9	8					6	
8			6					3
4		8		3				1
7			2				6	
	6				2	8		
		4	1	9			5	
			8			7	9	

5	3	4	6	7	8	9	1	2
6	7	2	1	9	5	3	4	8
1	9	8	3	4	2	5	6	7
8	5	9	7	6	1	4	2	3
4	2	6	8	5	3	7	9	1
7	1	3	9	2	4	8	5	6
9	6	1	5	3	7	2	8	4
2	8	7	4	1	9	6	3	5
3	4	5	2	8	6	1	7	9

Lo schema di partenza da risolvere è descritto in un apposito file di testo, il cui formato è riportato più oltre.

TEMPO STIMATO PER SVOLGERE L'INTERO COMPITO: 1h50 – 2h20



SEMANTICA:

- a) la classe **SudokuBoard** (**da realizzare**) rappresenta lo schema di gioco, una *scacchiera quadrata NxN* di **OptionalInt**. La classe deve mettere a disposizione i seguenti metodi:
- la dimensione della griglia N , impostata a 9, deve essere memorizzata in un'apposita costante
 - **costruttore** crea la scacchiera e la inizializza nel modo opportuno (corrispondente alla scacchiera vuota)
 - **setCellRow** consente di caricare nello schema un'intera riga di valori. Per convenzione, nell'array che viene passato in ingresso al metodo le celle vuote sono indicate con il valore 0. Il metodo deve controllare accuratamente i parametri ricevuti (sia le dimensioni che i valori), lanciando **IllegalArgumentException** in caso di non conformità, con adeguata messaggistica.
 - **getCell** restituisce un **OptionalInt** che rappresenta il contenuto della cella relativa agli indici ricevuti come argomento. Come sopra, il metodo deve controllare accuratamente i parametri ricevuti, lanciando **IllegalArgumentException** in caso di non conformità.
 - **getSize** restituisce la dimensione N della scacchiera (numero di celle per ogni riga o colonna)
 - **clearCell** “pulisce” il contenuto della cella relativa agli indici passati come argomento. Al solito, il metodo deve controllare i parametri ricevuti, lanciando **IllegalArgumentException** se necessario.
 - **setCell** imposta la cella relativa alle coordinate ricevute in ingresso al valore ricevuto anch'esso in ingresso come terzo argomento, verificando preventivamente che tale inserimento rispetti le regole del SUDOKU: in caso positivo restituisce *true*, mentre se così non è non inserisce nulla e restituisce *false*. Anche qui, il metodo deve controllare i parametri ricevuti, lanciando **IllegalArgumentException** se necessario.
 - **getEmptyCellNumber** restituisce il numero di caselle ancora vuote.
 - **toString** restituisce una stringa con la struttura della scacchiera (in righe) intesa come sequenza dei numeri che etichettano le varie caselle, *utilizzando il carattere “ ” per indicare le caselle vuote*.

Persistenza (*sudoku.persistence*)

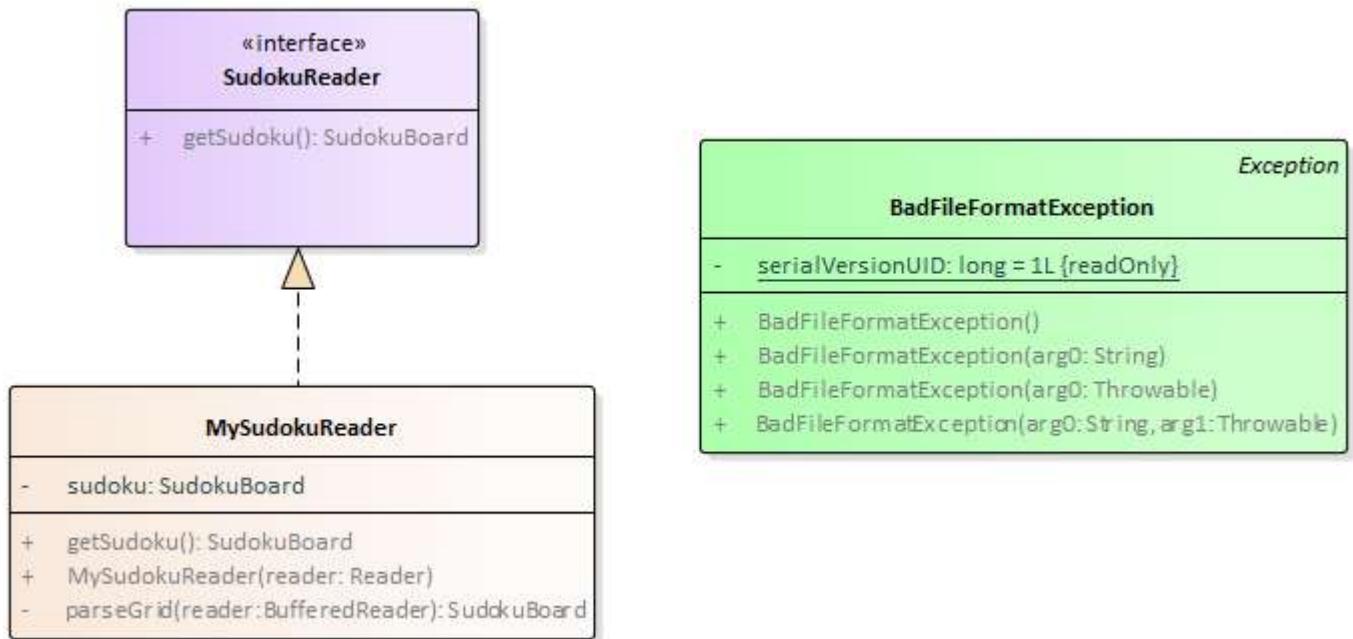
[TEMPO STIMATO: 30-40 minuti] (punti 10)

Lo schema di gioco è descritto nel file di testo **sudokuConfig.txt** formattato come segue:

- ogni riga nel file di testo descrive una riga della scacchiera del sudoku, che a sua volta conterrà i dati delle rispettive colonne (nel caso della scacchiera classica 9x9, ci saranno quindi 9 righe ognuna delle quali descriverà 9 colonne): è esplicitamente richiesto di non cablare soluzione sulla dimensione 9 del sudoku tradizionale
- in ogni riga i valori numerici sono separati fra loro da tabulazioni, mentre le celle vuote sono rappresentate dal carattere "#"

Esempio:

```
7 # # 5 8 # # # 3
# # 4 # # # # 2 #
...
```



SEMANTICA:

- L'interfaccia **SudokuReader** (fornita) dichiara il metodo `getSudoku` che restituisce la scacchiera letta
- La classe **MySudokuReader** (**da realizzare**) implementa **SudokuReader**: il costruttore riceve un **Reader** già aperto ed effettua la lettura, costruendo e memorizzando internamente il **SudokuBoard** corrispondente. L'accessor specificato da **SudokuReader** restituisce tale **SudokuBoard**. In caso di problemi di I/O deve essere propagata l'opportuna **IOException**, mentre in caso di problemi di formato dei file deve essere lanciata una **BadFormatException** (fornita) il cui messaggio dettagli l'accaduto.

In particolare, il reader deve verificare: 1) che ogni riga contenga esattamente 9 item e che ci siano esattamente 9 righe; 2) che i numeri presenti siano tutti compresi tra 1 e 9.

Parte 2

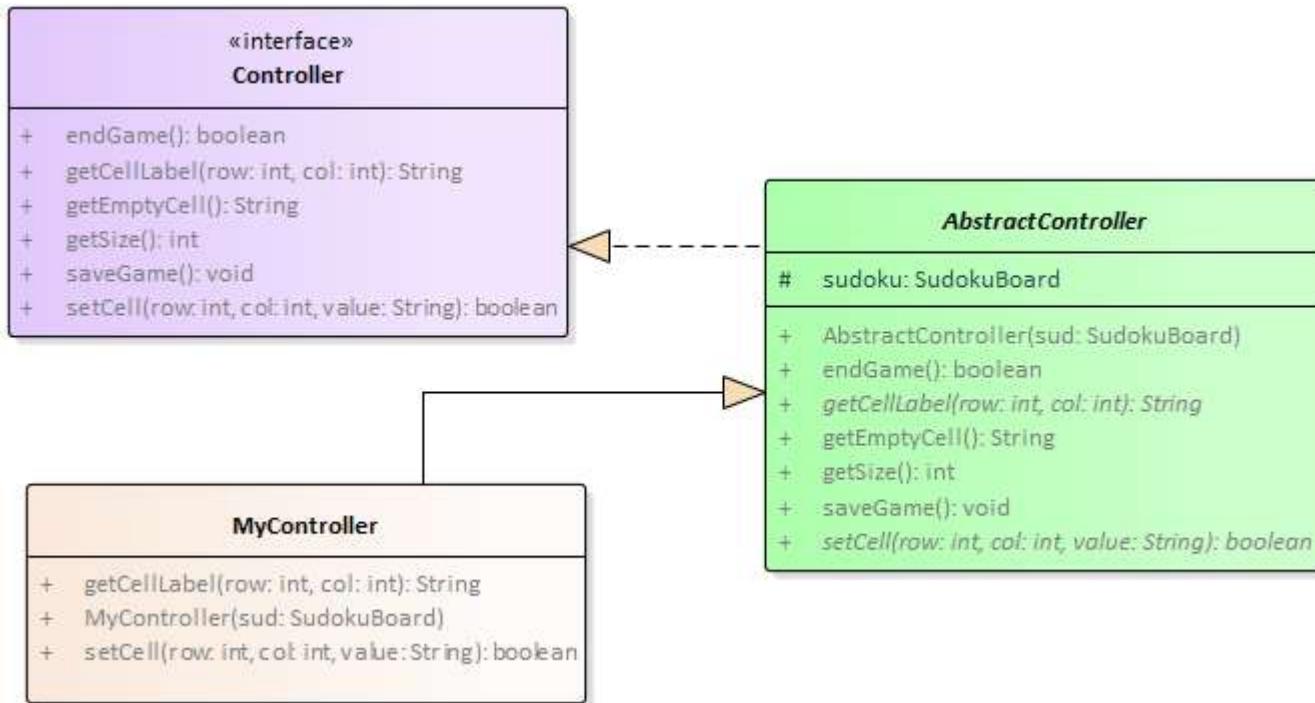
(punti: 8)

Controller (package sudoku.controllerl)

[TEMPO STIMATO: 10-15 minuti]

(punti: 4)

Il Controller è organizzato secondo il diagramma UML in figura.



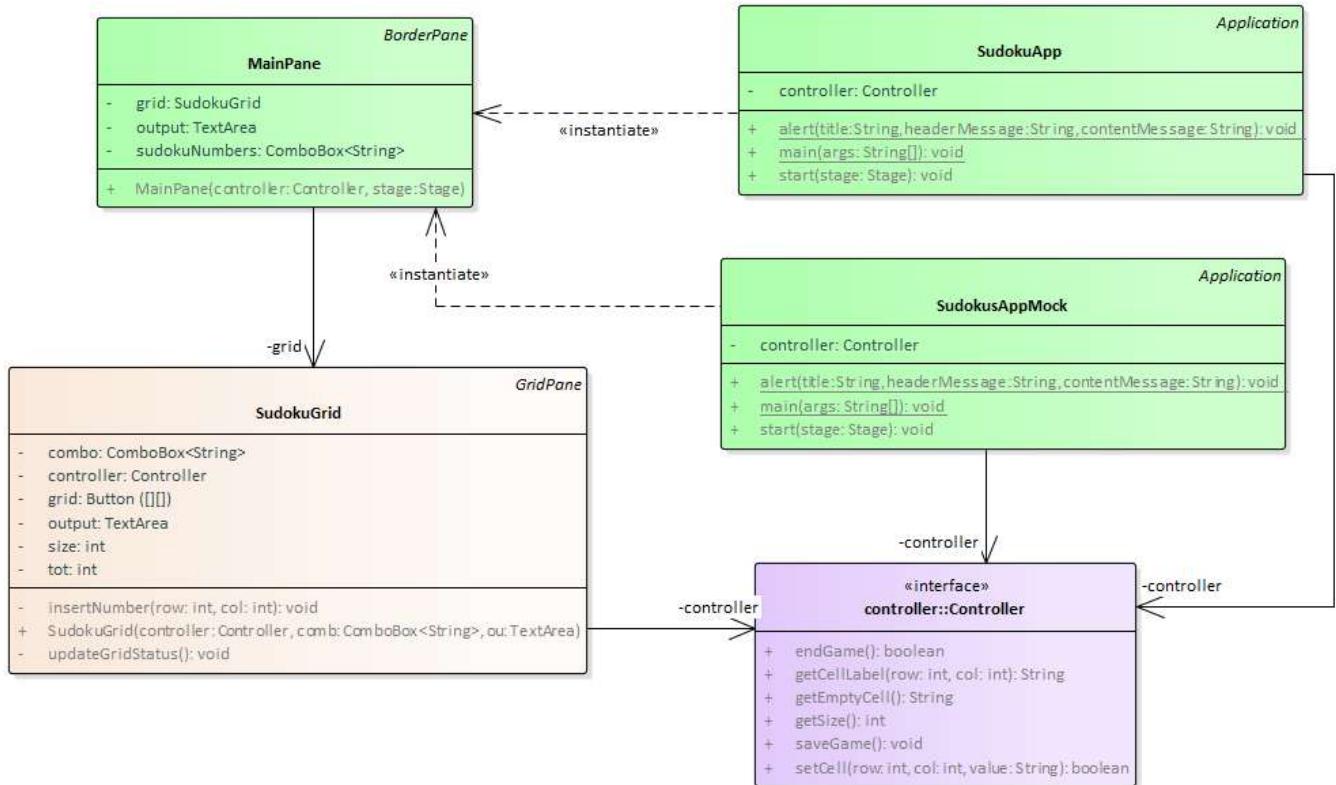
SEMANTICA:

- L'interfaccia **Controller** (fornita) dichiara i metodi
 - **getSize** restituisce la dimensione N della scacchiera intesa come numero di celle per riga (o colonna)
 - **getCellLabel** restituisce il contenuto della cella specificata in forma di stringa
 - **setCell** imposta la cella relativa alle coordinate ricevute in ingresso al valore ricevuto anch'esso in ingresso come terzo parametro, richiamando gli opportuni metodi del model per effettuare l'operazione richiesta. Restituisce true se l'operazione è andata a buon fine, false altrimenti
 - **endGame** restituisce true se la scacchiera è tutta piena
 - **saveGame** salva su File di testo la soluzione ottenuta
 - **getEmptyCell** restituisce il numero di celle ancora da riempire
- La classe **AbstractController** (fornita) implementa l'interfaccia **Controller** e lascia astratti i due metodi **setCell** e **getCellLabel**
- La classe **MyController** (**da realizzare**) estende la classe **AbstractController**, fornendo l'implementazione ai due metodi astratti **setCell** e **getCellLabel** secondo quanto specificato nell'interfaccia **Controller**

Interfaccia utente (package sudoku.ui)

[TEMPO STIMATO: 10-15 minuti] (punti: 4)

L'interfaccia utente è illustrata nelle figure seguenti e segue il modello sotto illustrato:



La classe **SudokuApp** (fornita) costituisce l'applicazione JavaFX che si occupa di aprire i file, creare il controller e incorporare il **MainPane**. Per consentire di collaudare la GUI anche in assenza / in caso di malfunzionamento della parte di persistenza, è possibile avviare l'applicazione mediante la classe **SudokusAppMock**.

Entrambe le classi contengono anche il **metodo statico ausiliario alert**, utile per mostrare avvisi all'utente.

La classe **MainPane** (fornita) estende **BorderPane** e prevede:

- 1) nel lato sinistro, una **Label**, una **ComboBox** che contiene i numeri da inserire nella scacchiera del Sudoku, una **TextArea** che indica quante caselle sono ancora da riempire nella scacchiera
- 2) nella parte centrale, una **SudokuGrid**.

La classe **SudokuGrid** è fornita parzialmente realizzata: è presente la parte strutturale, mentre manca la parte di gestione degli eventi. Questa classe estende un **GridPane** e rappresenta una griglia di pulsanti.

La **parte da completare** comprende l'uso e/o l'implementazione dei seguenti metodi:

- 1) **updateGridStatus**, usato sia all'inizio per configurare lo stato iniziale (e questo lo abbiamo già fatto noi) sia in **insertNumber** (questo lo dovete fare voi) inserisce la corretta label su ogni bottone e aggiorna la TextArea in base al numero di celle che sono ancora da riempire (Fig.1)
- 2) Il metodo **insertNumber** reagisce alla pressione dei bottoni che rappresentano la scacchiera e deve eseguire le seguenti azioni:
 - recupera il valore della **ComboBox**
 - prova ad impostare tale valore nella cella
 - se non è possibile, segnala errore (Fig.2)
 - verifica se il gioco è terminato e nel caso provvede a salvare su file la scacchiera completa, mostrando contestualmente a video un messaggio di complimenti (Fig.3)

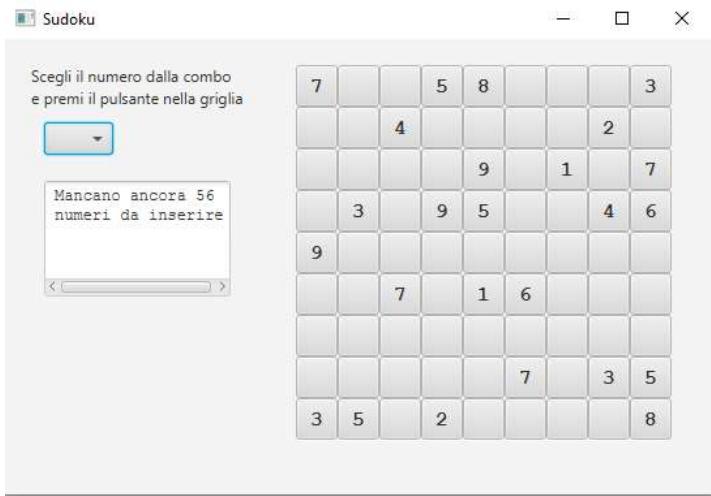


Fig.1

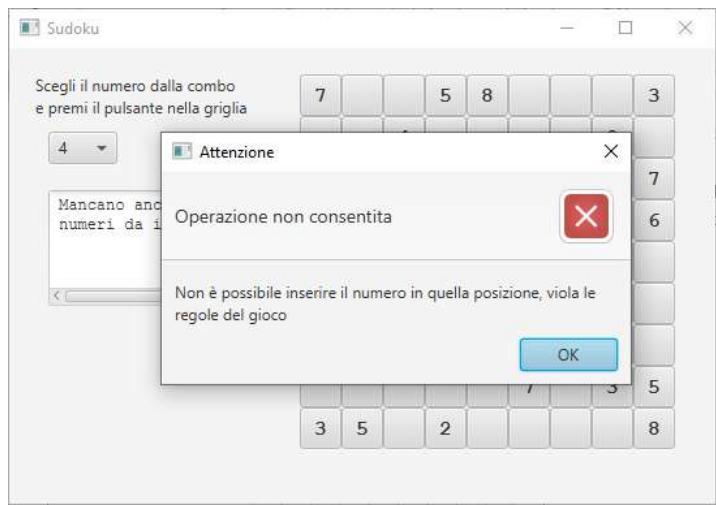


Fig.2

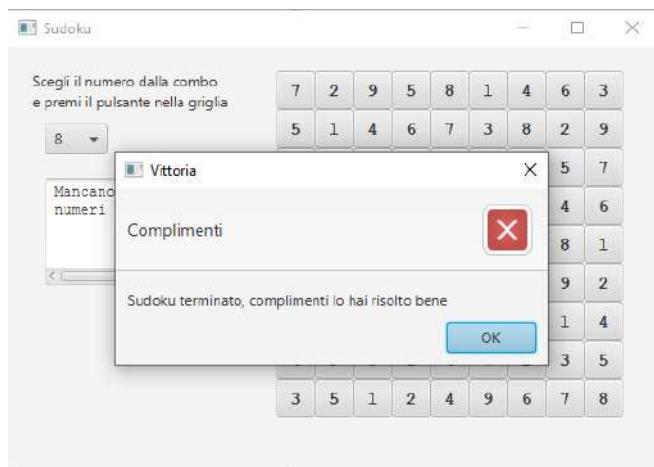


Fig.3

Cose da ricordare

- salva costantemente il tuo lavoro: l'informatica a volte può essere “subdolamente ostile”..
- in particolare: se ora compila e stai per fare modifiche, salva la versione attuale (non si sa mai)

Checklist di consegna

- Hai fatto un **JAR eseguibile**, che contenga cioè l'indicazione del main?
- Hai controllato che **si compili e ci sia tutto**? [NB: non includere il PDF del testo]
- Hai **rinominato IL PROGETTO**, lo ZIP e il JAR esattamente come richiesto?
- Hai **chiamato** la cartella del progetto esattamente come richiesto?
- **Hai fatto un unico file ZIP (NON .7z, rar o altri formati)** contenente l'intero progetto?
In particolare, ti sei assicurato di aver incluso tutti i file .java (e non solo i .class)?
- **Hai consegnato DUE file distinti, ossia lo ZIP col progetto e il JAR eseguibile?**
- Su EOL, hai premuto il tasto “CONFERMA” per inviare il tuo elaborato?

ESAME DI FONDAMENTI DI INFORMATICA T-2 del 06/02/2020

Proff. E. Denti – R. Calegari – A. Molesini

Tempo a disposizione: 4 ore MAX

NOME PROGETTO ECLIPSE: CognomeNome-matricola (es. RossiMario-0000123456)

NOME CARTELLA PROGETTO: CognomeNome-matricola (es. RossiMario-0000123456)

NOME ZIP DA CONSEGNARE: CognomeNome-matricola.zip (es. RossiMario-0000123456.zip)

NB: l'archivio ZIP da consegnare deve contenere l'intero progetto Eclipse

Si ricorda che compiti *non compilabili o palesemente lontani da 18/30* NON SARANNO CORRETTI e causeranno la verbalizzazione del giudizio “RESPINTO”

Per le elezioni del Governatore di Dentinia la legge elettorale prevede l'uso di un sistema proporzionale, ma lascia la possibilità di introdurre uno *sbarramento* che limiti la frammentazione, escludendo le liste che non ottengono una certa *percentuale minima* di voti. Si vuole sviluppare un'applicazione che simuli l'attribuzione dei seggi al variare, appunto, del livello di sbarramento fra un minimo di 0 (assenza di qualunque sbarramento) e un massimo del 10%.

DESCRIZIONE DEL DOMINIO DEL PROBLEMA

Nei sistemi proporzionali i seggi sono attribuiti in proporzione ai voti ottenuti da ciascuno. In particolare, nel sistema *D'Hondt* (il più diffuso) l'attribuzione dei seggi avviene *dividendo i voti ottenuti da ogni lista per i numeri naturali 1,2,3,4...* e inserendo poi i risultati di tutte le liste in un'unica graduatoria decrescente: i seggi si attribuiscono quindi seguendo tale graduatoria.

ESEMPIO

Si supponga di dover assegnare 20 seggi ai partiti A, B, C, D, E che abbiano ottenuto rispettivamente 9.100.000, 7.200.000, 4.880.000, 4.100.000, 1.320.000 voti. Si dividono quindi tali voti per i numeri naturali da 1 a 20 (caso peggiore in cui un singolo partito prenda tutti i seggi), ottenendo i seguenti valori:

A) 9.100.000, 4.550.000, 3.033.333, 2.275.000, 1.820.000, 1.516.666, 1.300.000, 1.137.500, 1.011.111, 910.000, ...

B) 7.200.000, 3.600.000, 2.400.000, 1.800.000, 1.440.000, 1.200.000, 1.028.571, 900.000, 800.000, 720.000, ...

C) 4.880.000, 2.440.000, 1.626.666, 1.220.000, 976.000, 813.333, 697.142, 610.000, 542.222, 488.000, ...

D) 4.100.000, 2.050.000, 1.366.666, 1.025.000, 820.000, 683.333, 585.714, 512.500, 455.555, 410.000, ...

E) 1.320.000, 1.160.000, 773.333, 580.000, 464.000, 386.666, 331.428, 290.000, 257.777, 232.000, ...

Per assegnare i 20 seggi in palio occorre ora prendere i 20 valori migliori, ossia quelli evidenziati in giallo: pertanto, la lista A ottiene 7 seggi, la lista B 5 seggi, la lista C 4 seggi, la lista D 3 seggi e la lista E 1 seggio soltanto.

Per attenuare l'effetto di parcellizzazione che i sistemi proporzionali tendono a determinare, spesso si introduce uno **sbarramento**, che esclude dal riparto i partiti che non raggiungono una data percentuale di voti.

Nel caso ad esempio di uno **sbarramento al 5%**, le liste che non raggiungono la soglia di 1.330.000 voti (sopra, la lista E) non partecipano al riparto: pertanto, il suo seggio viene assegnato al successivo in graduatoria (la lista B, in azzurro).

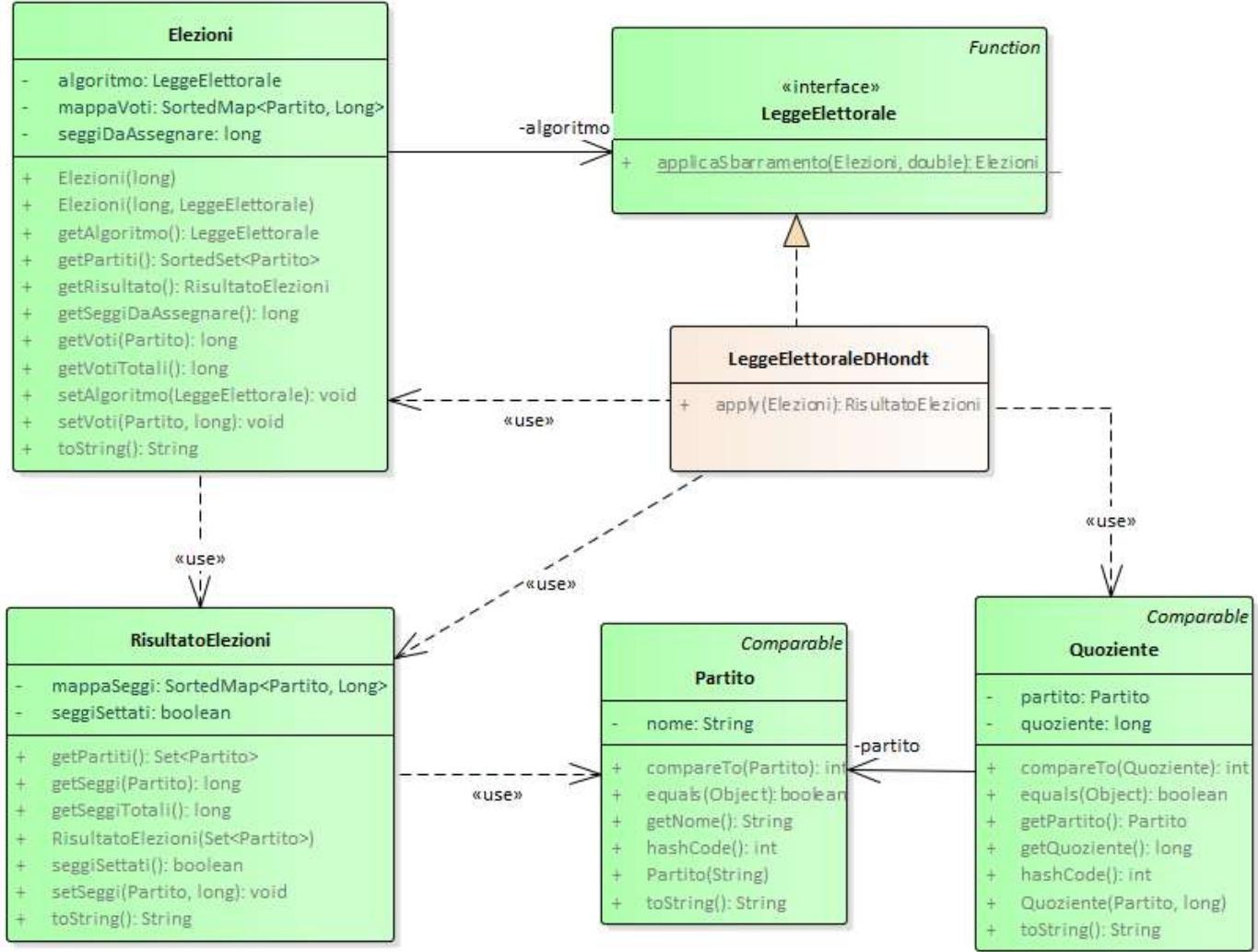
Il file di testo [Voti.txt](#) contiene i risultati delle elezioni (nel formato dettagliato più oltre): la prima riga indica il numero di seggi da assegnare, la seconda i voti ottenuti da ciascun partito, separati da virgole.

Parte 1

(punti: 20)

Dati (package dentinia.governor.model)

(punti: 9)



SEMANTICA:

- la classe **Partito** (fornita) rappresenta un partito, caratterizzato dal suo nome;
- la classe **Elezioni** (fornita) mantiene i dati relativi ai voti ottenuti da ogni partito: il costruttore riceve il numero di seggi da assegnare ed eventualmente l'algoritmo da usare per il calcolo del risultato (se non specificato, rimane null). Gli accessori `setVoti/getVoti` operano sui voti di un dato partito in modo controllato, mentre gli accessori `setAlgoritmo/getAlgoritmo` impostano/restituiscono l'algoritmo di calcolo da usare; ovviamente, i metodi `getPartiti`, `getVotiTotali` e `getSeggiDaAssegnare` restituiscono le proprietà omonime. Il metodo `toString` restituisce una stringa multi-linea con partiti, voti e seggi perfettamente formattata. Il metodo `getRisultato` effettua il calcolo dei seggi con l'algoritmo precedentemente impostato: il risultato è un oggetto di tipo **RisultatoElezioni**, che associa a ogni partito i seggi corrispondenti. Se l'algoritmo è null, restituisce un'istanza di **RisultatoElezioni** con tutti i seggi azzerati.
- la classe **RisultatoElezioni** (fornita) mantiene i dati relativi ai seggi ottenuti da ciascun partito: il costruttore riceve un set di **Partito** e impone i seggi tutti a zero. Gli accessori `setSeggi/getSeggi`, `getSeggiTotali` permettono di recuperare le corrispondenti proprietà. Diversamente dal caso precedente, il metodo `toString` restituisce una mera rappresentazione interna utile solo a fini di debugging. Infine, il metodo `seggiSettati` restituisce un boolean che indica se i seggi sono stati impostati dopo la costruzione iniziale.

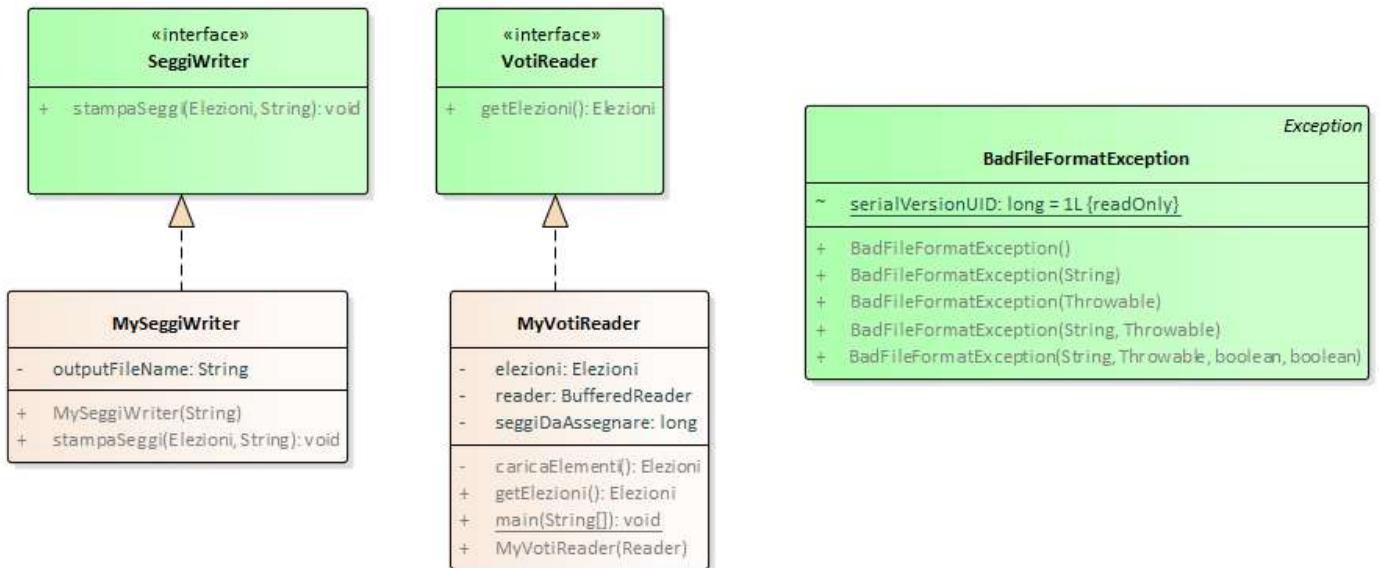
- d) l'interfaccia **LeggeElettorale** (fornita) è un semplice alias per il tipo funzione da **Elezioni** a **RisultatoElezioni**. Contiene però il metodo statico **applicaSbarramento**, che applica alle **Elezioni** date lo sbarramento specificato, restituendo una nuova istanza di Elezioni in cui i voti dei partiti sotto soglia sono azzerati.
- e) la classe **Quoziente** (fornita) rappresenta una coppia <**Partito**, valore>, utile per rappresentare i valori ottenuti nell'algoritmo D'Hondt: per tale motivo, è comparabile in senso decrescente sul valore (intero). Ovviamente, il costruttore riceve i due elementi della coppia, e due accessori consentono di recuperarli.
- f) la classe **LeggeElettoraleDHondt** (da realizzare) implementa **LeggeElettorale** usando il metodo D'Hondt. Si prevede che utilizzi opportune istanze di **Quoziente** per rappresentare le divisioni successive dei voti di ogni partito per 1,2,3,4...

Persistenza (dentinia.governor.persistence)

(punti 11)

Il file di testo **Voti.txt** contiene i risultati delle elezioni: la prima riga indica il numero di seggi totali da assegnare, nella forma "SEGGI" seguita da un intero, mentre la successiva riporta i voti ottenuti da ciascun partito, separati da virgole: ogni coppia partito/voti prevede prima il nome del partito (che può contenere spazi), poi i voti, formattati secondo l'uso italiano con il “.” come separatore delle migliaia.

```
SEGGI 20
Lista A 9.100.000, Lista E 1.320.000, Lista C 4.880.000, Lista D 4.100.000, Lista B 1.500.000
```



SEMANTICA:

- L'interfaccia **VotiReader** (fornita) dichiara il metodo **getElezioni**, che restituisce una istanza di **Elezioni** opportunamente popolata;
- L'interfaccia **SeggiWriter** (fornita) dichiara il metodo **stampaSeggi** che stampa **Elezioni** utilizzando la **toString**, preceduta dalla data e ora corrente e dal messaggio ricevuto come argomento; la data e ora devono includere il nome del giorno (formato FULL, ma attenzione.... ☺)
- La classe **MyVotiReader** (da realizzare) implementa **VotiReader**: il costruttore riceve in ingresso il Reader da cui leggere i dati, effettua la lettura e popola opportunamente l'entità Elezioni mantenuta al suo interno, che viene poi semplicemente restituita a ogni invocazione di **getElezioni**. In caso di problemi di I/O viene propagata l'opportuna **IOException**, nel caso di Reader nullo deve invece essere lanciata una **IllegalArgumentException**, infine eventuali problemi nel formato del file devono essere incapsulati in un'opportuna **BadFormatException**.

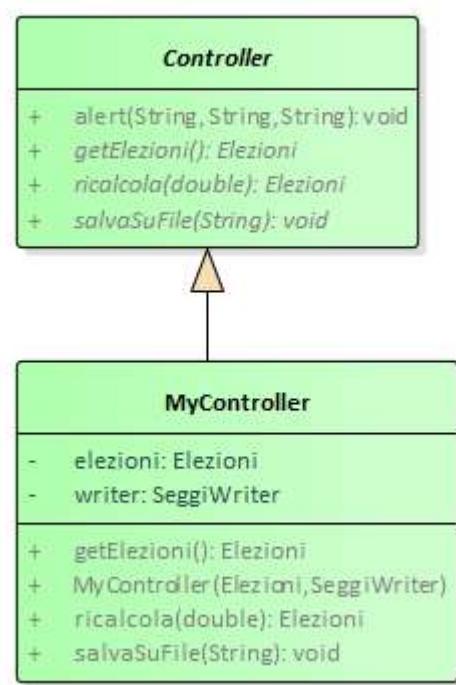
- d) la classe **MySeggiWriter** (da realizzare) implementa **SeggiWriter**: il costruttore riceve in ingresso il nome del file su cui scrivere (il main dell'applicazione usa “**Report.txt**”). Il metodo **stampaSeggi** deve produrre una tabella analoga a quella mostrata in calce, in cui la prima riga contiene data e ora (nel formato indicato), la seconda una frase con l'indicazione del livello di sbarramento applicato, mentre dalla terza riga in poi sono riportati, uno per riga, i partiti coi rispettivi voti e seggi, separati da tabulazioni e formattati secondo l'uso italiano per le migliaia (vedi esempio). In caso di problemi di I/O viene propagata l'opportuna **IOException**.

giovedì 16 gennaio 2020, 20:44:20
Metodo D'Hondt con sbarramento del 0.0%
Lista A Voti: 9.100.000 Seggi: 9
Lista B Voti: 1.500.000 Seggi: 1
Lista C Voti: 4.880.000 Seggi: 5
Lista D Voti: 4.100.000 Seggi: 4
Lista E Voti: 1.320.000 Seggi: 1
TOTALE Voti: 20.900.000 Seggi: 20

Parte 2

(punti: 10)

Il Controller (**completamente fornito**) è organizzato secondo il diagramma UML in figura.



SEMANTICA:

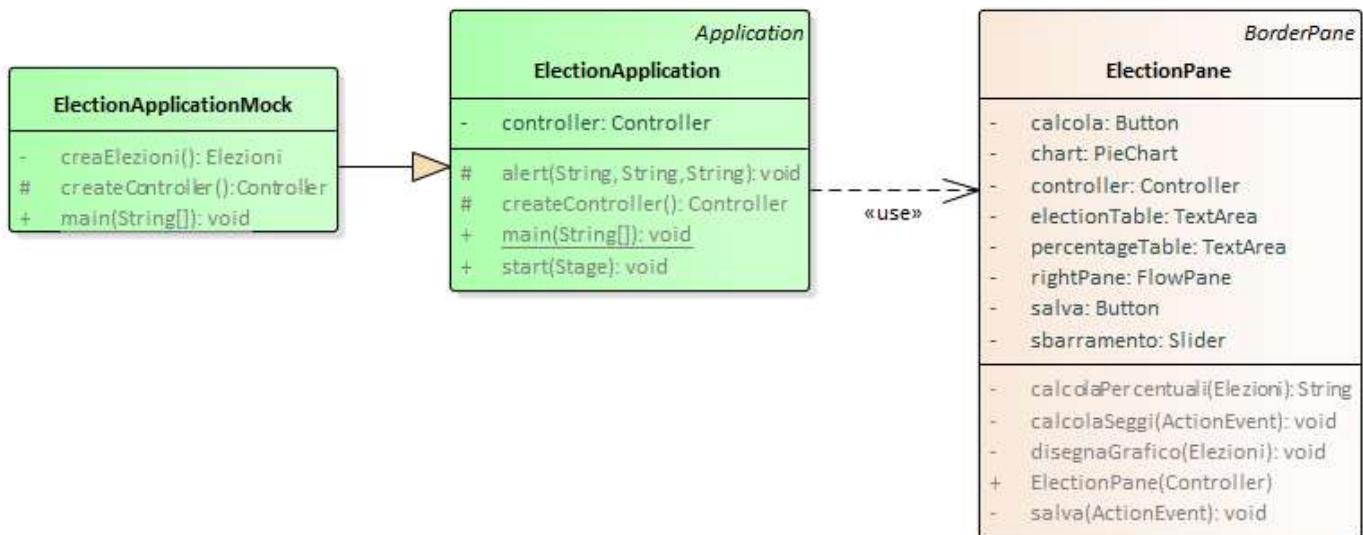
- a) La classe astratta **Controller** (fornita) dichiara l'interfaccia del controller (metodi **ricalcola**, **getVoti** e **salvaSuFile**) e implementa il metodo statico ausiliario **alert**, utile per mostrare avvisi all'utente.
- b) La classe **MyController** (pure fornita) completa l'implementazione realizzando:
- il costruttore a due argomenti (**Elezioni** e **SeggiWriter**) che memorizza gli argomenti in opportuni campi e imposta la legge elettorale D'Hondt come algoritmo da usare; tale istanza di **Elezioni** è quella che viene restituita dal metodo **getElezioni**;
 - il metodo **ricalcola** riceve in ingresso lo sbarramento richiesto, lo applica (tramite **LeggeElettorale**) e restituisce il risultato sotto forma di nuova istanza di **Elezioni**;
 - il metodo **salvaSuFile** opera solo se il controller dispone internamente di un oggetto **Elezioni** valido: in tal caso, semplicemente delega all'opportuno **SeggiWriter** la stampa effettiva, passandogli anche il messaggio personalizzato ricevuto come argomento.

Interfaccia utente ([dentinia.governor.ui.javafx](#))

(punti 10)

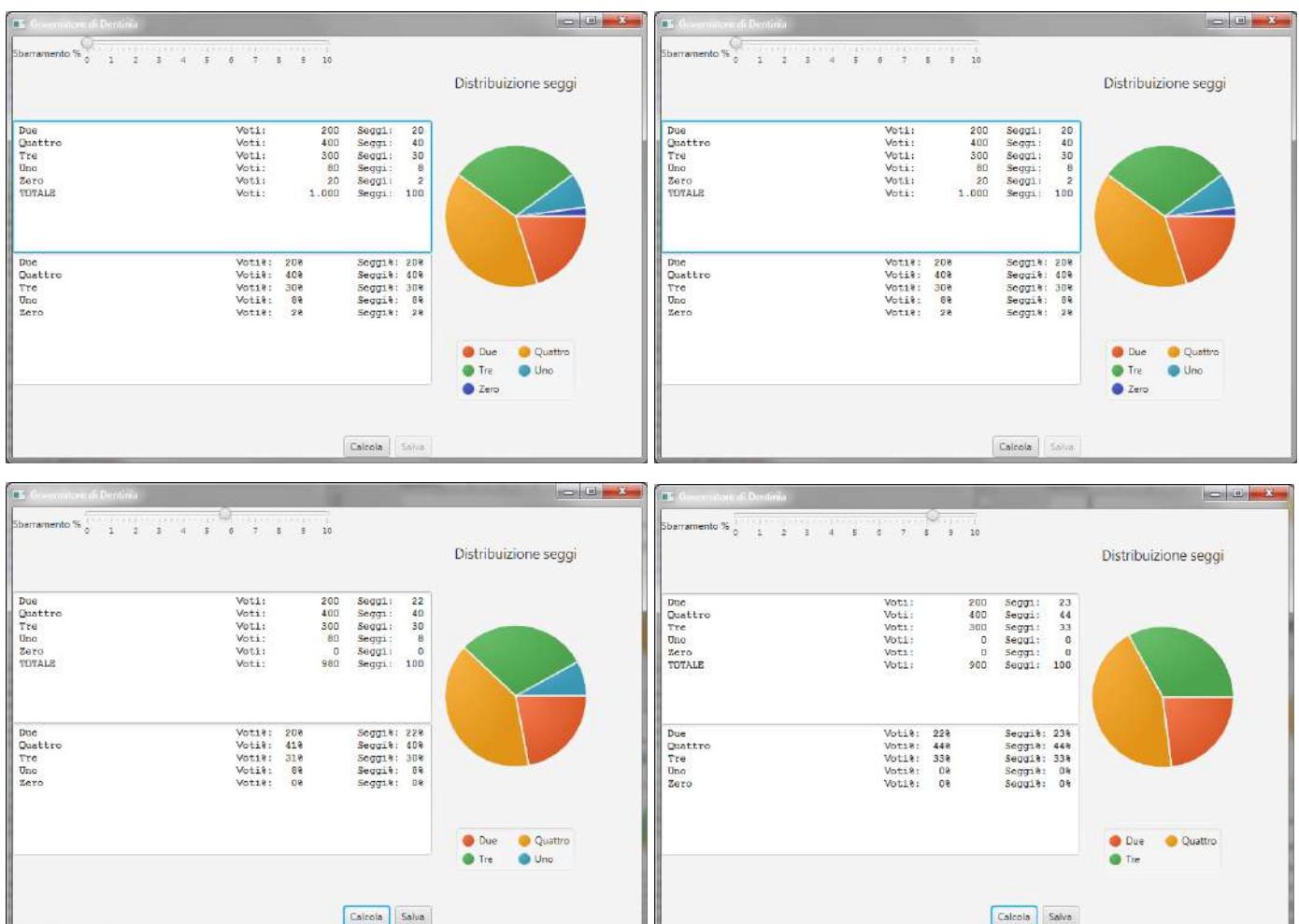
L'interfaccia utente deve essere simile (non necessariamente identica) all'esempio mostrato nelle figure seguenti.

L'architettura segue il modello sotto illustrato:



La classe **ElectionApplication** (fornita) costituisce l'applicazione JavaFX che si occupa di aprire il file, il controller e incorporare l'**ElectionPane** (da realizzare). Per consentire di collaudare la GUI anche in assenza della parte di persistenza, è possibile avviare l'applicazione mediante la classe **ElectionApplicationMock**.

L'interfaccia utente deve essere simile (non necessariamente identica) all'esempio mostrato nella figura seguente.



La classe **ElectionPane** (da realizzare), che estende **BorderPane**, prevede:

- 1) in alto, uno **Slider** configurato da 0 a 10, passo 1, inizialmente a 0, per impostare lo sbarramento %
- 2) a sinistra, due **TextArea** una sotto l'altra: la prima mostra la situazione delle elezioni in termini di voti e seggi assegnati, mentre la seconda esprime le stesse informazioni in percentuale. All'inizio mostrano la situazione corrispondente all'assenza di sbarramento.
- 3) A destra, un **PieChart** (senza etichette) mostra anch'esso la situazione corrente, in forma grafica.
- 4) In basso, due pulsanti *Calcola* e *Salva*, di cui il secondo inizialmente disabilitato, controllano il funzionamento dell'applicazione. *Calcola* scatena il calcolo dei seggi, tenendo conto dello sbarramento prescelto e aggiornando tabelle e grafico a torta: dopo ogni ricalcolo ri-abilita il pulsante *Salva* per consentire il salvataggio su file dei nuovi risultati. Il pulsante *Salva* effettua il salvataggio dei risultati sul file, auto-disabilitandosi subito dopo.

NB: mentre i dati della prima tabella sono facilmente ottenibili dal model, i secondi vanno calcolati!

Cose da ricordare

- salva costantemente il tuo lavoro, facendo ZIP parziali e consegne parziali (vale l'ultima)
- in particolare, se ora compila e stai per fare modifiche, salva la versione attuale (non si sa mai..)

Checklist di consegna

- Hai fatto un unico file ZIP (**non .7z!!!**) contenente l'intero progetto?
In particolare, ti sei assicurato di aver incluso tutti i file .java (e non solo i .class)?
- Hai controllato che si compili e ci sia tutto ? [NB: non serve includere il PDF del testo]
- Hai rinominato IL PROGETTO esattamente come richiesto?
- Hai chiamato IL FILE ZIP esattamente come richiesto?
- Hai chiamato la cartella del progetto esattamente come richiesto?
- Dopo aver caricato il file su Esamix, hai premuto il tasto “CONFERMA”, ottenendo il messaggio “Hai concluso l'esame”?

ESAME DI FONDAMENTI DI INFORMATICA T-2 del 06/02/2020

Proff. E. Denti – R. Calegari – A. Molesini

Tempo a disposizione: 4 ore MAX

NOME PROGETTO ECLIPSE: CognomeNome-matricola (es. RossiMario-0000123456)

NOME CARTELLA PROGETTO: CognomeNome-matricola (es. RossiMario-0000123456)

NOME ZIP DA CONSEGNARE: CognomeNome-matricola.zip (es. RossiMario-0000123456.zip)

NB: l'archivio ZIP da consegnare deve contenere l'intero progetto Eclipse

Si ricorda che compiti *non compilabili o palesemente lontani da 18/30* NON SARANNO CORRETTI e causeranno la verbalizzazione del giudizio “RESPINTO”

Per le elezioni del Governatore di Dentinia la legge elettorale prevede l'uso di un sistema proporzionale, ma lascia la possibilità di introdurre uno *sbarramento* che limiti la frammentazione, escludendo le liste che non ottengono una certa *percentuale minima* di voti. Si vuole sviluppare un'applicazione che simuli l'attribuzione dei seggi al variare, appunto, del livello di sbarramento fra un minimo di 0 (assenza di qualunque sbarramento) e un massimo del 10%.

DESCRIZIONE DEL DOMINIO DEL PROBLEMA

Nei sistemi proporzionali i seggi sono attribuiti in proporzione ai voti ottenuti da ciascuno. In particolare, nel sistema *D'Hondt* (il più diffuso) l'attribuzione dei seggi avviene *dividendo i voti ottenuti da ogni lista per i numeri naturali 1,2,3,4...* e inserendo poi i risultati di tutte le liste in un'unica graduatoria decrescente: i seggi si attribuiscono quindi seguendo tale graduatoria.

ESEMPIO

Si supponga di dover assegnare 20 seggi ai partiti A, B, C, D, E che abbiano ottenuto rispettivamente 9.100.000, 7.200.000, 4.880.000, 4.100.000, 1.320.000 voti. Si dividono quindi tali voti per i numeri naturali da 1 a 20 (caso peggiore in cui un singolo partito prenda tutti i seggi), ottenendo i seguenti valori:

A) 9.100.000, 4.550.000, 3.033.333, 2.275.000, 1.820.000, 1.516.666, 1.300.000, 1.137.500, 1.011.111, 910.000, ...

B) 7.200.000, 3.600.000, 2.400.000, 1.800.000, 1.440.000, 1.200.000, 1.028.571, 900.000, 800.000, 720.000, ...

C) 4.880.000, 2.440.000, 1.626.666, 1.220.000, 976.000, 813.333, 697.142, 610.000, 542.222, 488.000, ...

D) 4.100.000, 2.050.000, 1.366.666, 1.025.000, 820.000, 683.333, 585.714, 512.500, 455.555, 410.000, ...

E) 1.320.000, 1.160.000, 773.333, 580.000, 464.000, 386.666, 331.428, 290.000, 257.777, 232.000, ...

Per assegnare i 20 seggi in palio occorre ora prendere i 20 valori migliori, ossia quelli evidenziati in giallo: pertanto, la lista A ottiene 7 seggi, la lista B 5 seggi, la lista C 4 seggi, la lista D 3 seggi e la lista E 1 seggio soltanto.

Per attenuare l'effetto di parcellizzazione che i sistemi proporzionali tendono a determinare, spesso si introduce uno **sbarramento**, che esclude dal riparto i partiti che non raggiungono una data percentuale di voti.

Nel caso ad esempio di uno **sbarramento al 5%**, le liste che non raggiungono la soglia di 1.330.000 voti (sopra, la lista E) non partecipano al riparto: pertanto, il suo seggio viene assegnato al successivo in graduatoria (la lista B, in azzurro).

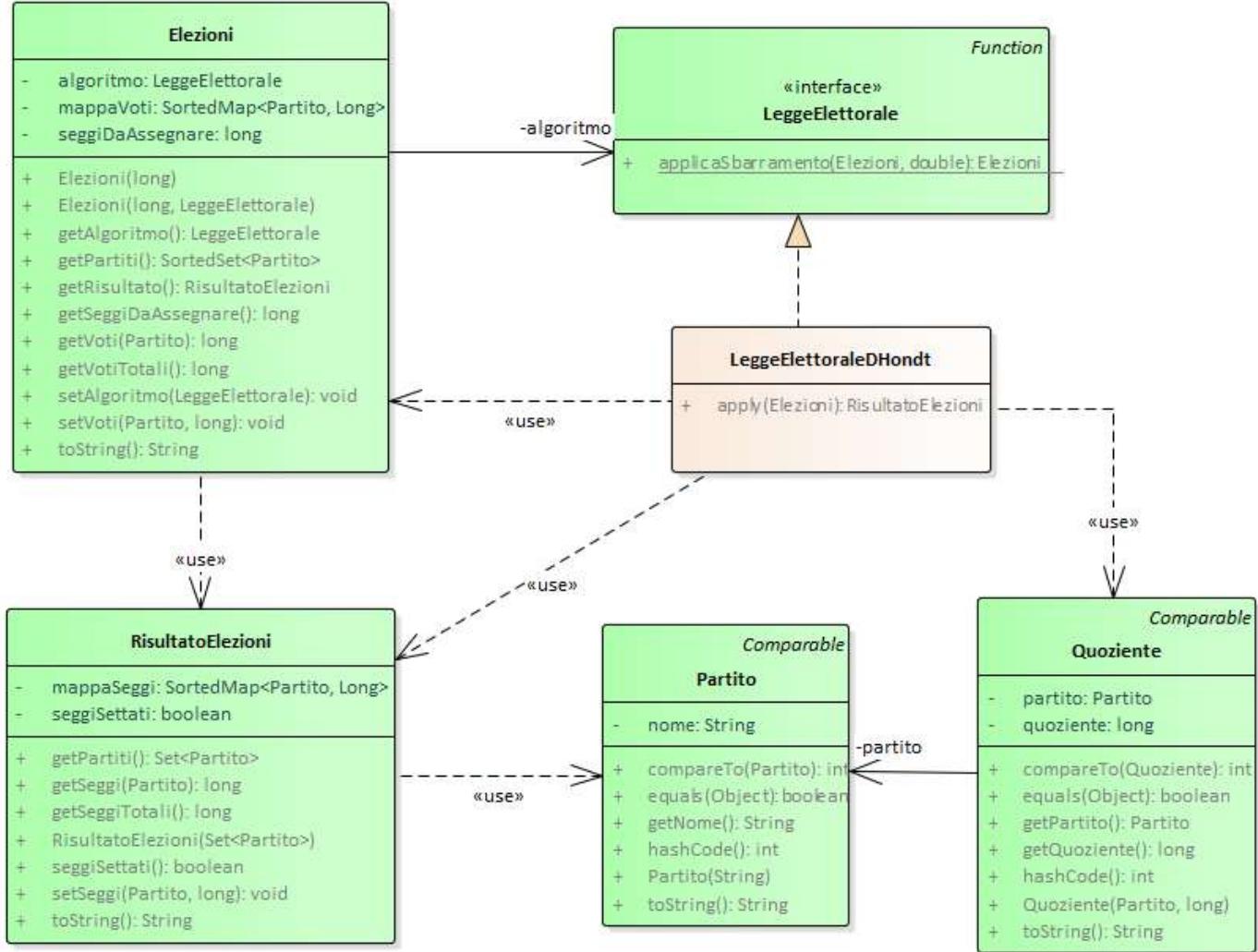
Il file di testo Voti.txt contiene i risultati delle elezioni (nel formato dettagliato più oltre): la prima riga indica il numero di seggi da assegnare, la seconda i voti ottenuti da ciascun partito, separati da virgole.

Parte 1

(punti: 20)

Dati (package dentinia.governor.model)

(punti: 9)



SEMANTICA:

- la classe **Partito** (fornita) rappresenta un partito, caratterizzato dal suo nome;
- la classe **Elezioni** (fornita) mantiene i dati relativi ai voti ottenuti da ogni partito: il costruttore riceve il numero di seggi da assegnare ed eventualmente l'algoritmo da usare per il calcolo del risultato (se non specificato, rimane null). Gli accessori `setVoti/getVoti` operano sui voti di un dato partito in modo controllato, mentre gli accessori `setAlgoritmo/getAlgoritmo` impostano/restituiscono l'algoritmo di calcolo da usare; ovviamente, i metodi `getPartiti`, `getVotiTotali` e `getSeggiDaAssegnare` restituiscono le proprietà omonime. Il metodo `toString` restituisce una stringa multi-linea con partiti, voti e seggi perfettamente formattata. Il metodo `getRisultato` effettua il calcolo dei seggi con l'algoritmo precedentemente impostato: il risultato è un oggetto di tipo **RisultatoElezioni**, che associa a ogni partito i seggi corrispondenti. Se l'algoritmo è null, restituisce un'istanza di **RisultatoElezioni** con tutti i seggi azzerati.
- la classe **RisultatoElezioni** (fornita) mantiene i dati relativi ai seggi ottenuti da ciascun partito: il costruttore riceve un set di **Partito** e impone i seggi tutti a zero. Gli accessori `setSeggi/getSeggi`, `getSeggiTotali` permettono di recuperare le corrispondenti proprietà. Diversamente dal caso precedente, il metodo `toString` restituisce una mera rappresentazione interna utile solo a fini di debugging. Infine, il metodo `seggiSettati` restituisce un boolean che indica se i seggi sono stati impostati dopo la costruzione iniziale.

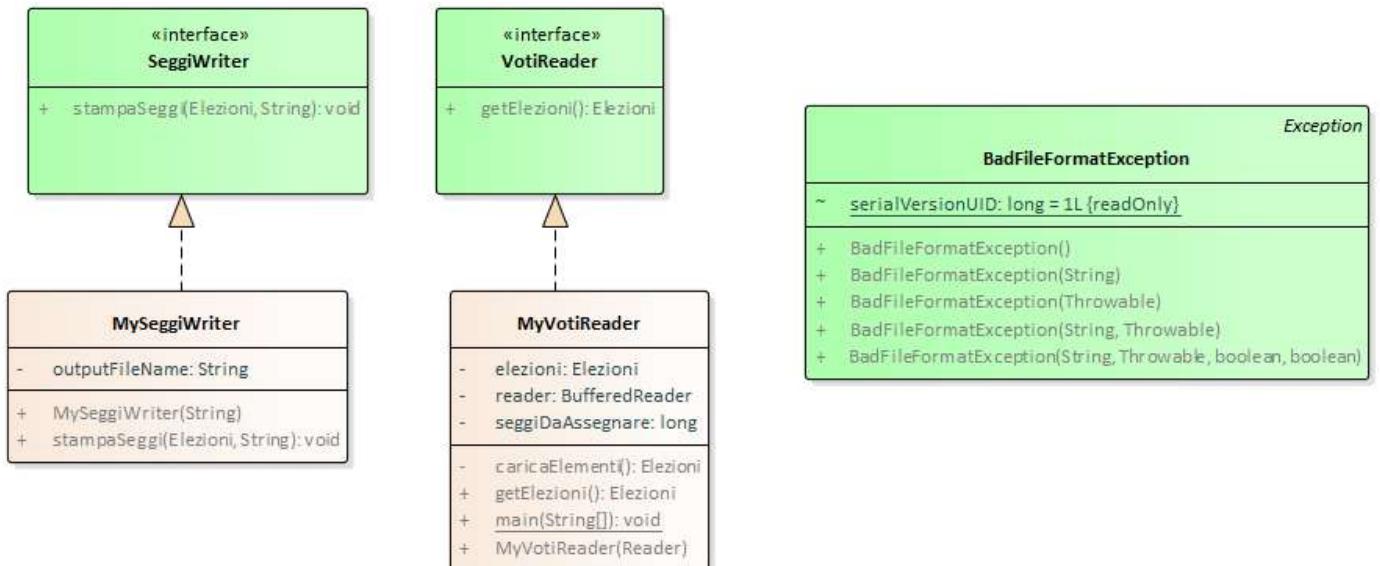
- d) l'interfaccia **LeggeElettorale** (fornita) è un semplice alias per il tipo funzione da **Elezioni** a **RisultatoElezioni**. Contiene però il metodo statico **applicaSbarramento**, che applica alle **Elezioni** date lo sbarramento specificato, restituendo una nuova istanza di Elezioni in cui i voti dei partiti sotto soglia sono azzerati.
- e) la classe **Quoziente** (fornita) rappresenta una coppia <**Partito**, valore>, utile per rappresentare i valori ottenuti nell'algoritmo D'Hondt: per tale motivo, è comparabile in senso decrescente sul valore (intero). Ovviamente, il costruttore riceve i due elementi della coppia, e due accessori consentono di recuperarli.
- f) la classe **LeggeElettoraleDHondt** (da realizzare) implementa **LeggeElettorale** usando il metodo D'Hondt. Si prevede che utilizzi opportune istanze di **Quoziente** per rappresentare le divisioni successive dei voti di ogni partito per 1,2,3,4...

Persistenza (dentinia.governor.persistence)

(punti 11)

Il file di testo **Voti.txt** contiene i risultati delle elezioni: la prima riga indica il numero di seggi totali da assegnare, nella forma "SEGGI" seguita da un intero, mentre la successiva riporta i voti ottenuti da ciascun partito, separati da virgole: ogni coppia partito/voti prevede prima il nome del partito (che può contenere spazi), poi i voti, formattati secondo l'uso italiano con il “.” come separatore delle migliaia.

```
SEGGI 20
Lista A 9.100.000, Lista E 1.320.000, Lista C 4.880.000, Lista D 4.100.000, Lista B 1.500.000
```



SEMANTICA:

- L'interfaccia **VotiReader** (fornita) dichiara il metodo **getElezioni**, che restituisce una istanza di **Elezioni** opportunamente popolata;
- L'interfaccia **SeggiWriter** (fornita) dichiara il metodo **stampaSeggi** che stampa **Elezioni** utilizzando la **toString**, preceduta dalla data e ora corrente e dal messaggio ricevuto come argomento; la data e ora devono includere il nome del giorno (formato FULL, ma attenzione.... ☺)
- La classe **MyVotiReader** (da realizzare) implementa **VotiReader**: il costruttore riceve in ingresso il Reader da cui leggere i dati, effettua la lettura e popola opportunamente l'entità Elezioni mantenuta al suo interno, che viene poi semplicemente restituita a ogni invocazione di **getElezioni**. In caso di problemi di I/O viene propagata l'opportuna **IOException**, nel caso di Reader nullo deve invece essere lanciata una **IllegalArgumentException**, infine eventuali problemi nel formato del file devono essere incapsulati in un'opportuna **BadFormatException**.

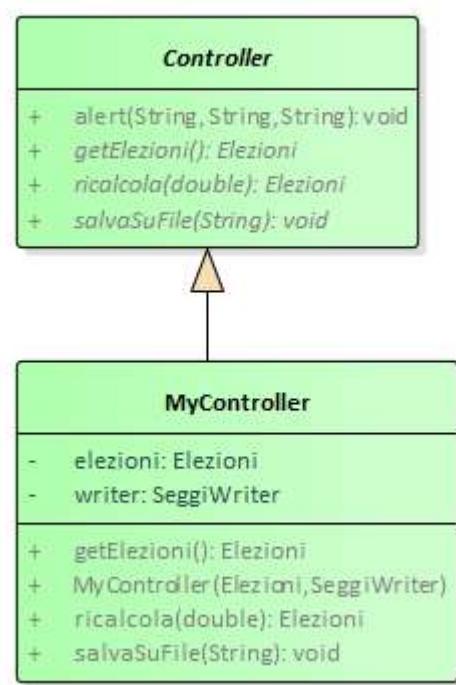
- d) la classe **MySeggiWriter** (da realizzare) implementa **SeggiWriter**: il costruttore riceve in ingresso il nome del file su cui scrivere (il main dell'applicazione usa “**Report.txt**”). Il metodo **stampaSeggi** deve produrre una tabella analoga a quella mostrata in calce, in cui la prima riga contiene data e ora (nel formato indicato), la seconda una frase con l'indicazione del livello di sbarramento applicato, mentre dalla terza riga in poi sono riportati, uno per riga, i partiti coi rispettivi voti e seggi, separati da tabulazioni e formattati secondo l'uso italiano per le migliaia (vedi esempio). In caso di problemi di I/O viene propagata l'opportuna **IOException**.

giovedì 16 gennaio 2020, 20:44:20
Metodo D'Hondt con sbarramento del 0.0%
Lista A Voti: 9.100.000 Seggi: 9
Lista B Voti: 1.500.000 Seggi: 1
Lista C Voti: 4.880.000 Seggi: 5
Lista D Voti: 4.100.000 Seggi: 4
Lista E Voti: 1.320.000 Seggi: 1
TOTALE Voti: 20.900.000 Seggi: 20

Parte 2

(punti: 10)

Il Controller (**completamente fornito**) è organizzato secondo il diagramma UML in figura.



SEMANTICA:

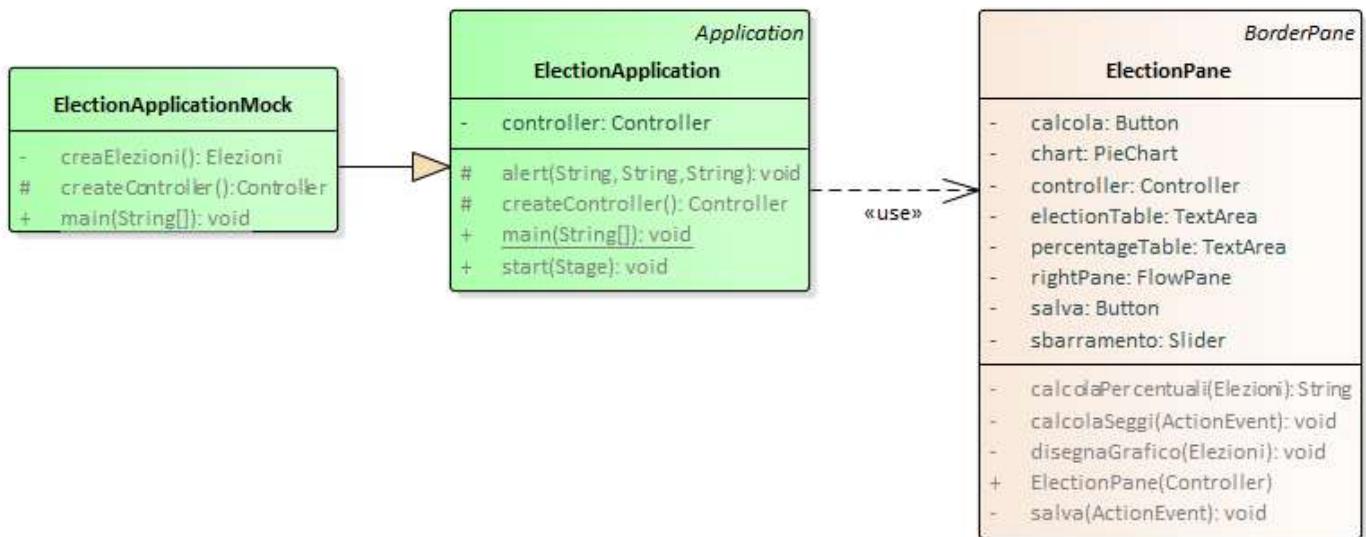
- a) La classe astratta **Controller** (fornita) dichiara l'interfaccia del controller (metodi **ricalcola**, **getVoti** e **salvaSuFile**) e implementa il metodo statico ausiliario **alert**, utile per mostrare avvisi all'utente.
- b) La classe **MyController** (pure fornita) completa l'implementazione realizzando:
- il costruttore a due argomenti (**Elezioni** e **SeggiWriter**) che memorizza gli argomenti in opportuni campi e imposta la legge elettorale D'Hondt come algoritmo da usare; tale istanza di **Elezioni** è quella che viene restituita dal metodo **getElezioni**;
 - il metodo **ricalcola** riceve in ingresso lo sbarramento richiesto, lo applica (tramite **LeggeElettorale**) e restituisce il risultato sotto forma di nuova istanza di **Elezioni**;
 - il metodo **salvaSuFile** opera solo se il controller dispone internamente di un oggetto **Elezioni** valido: in tal caso, semplicemente delega all'opportuno **SeggiWriter** la stampa effettiva, passandogli anche il messaggio personalizzato ricevuto come argomento.

Interfaccia utente ([dentinia.governor.ui.javafx](#))

(punti 10)

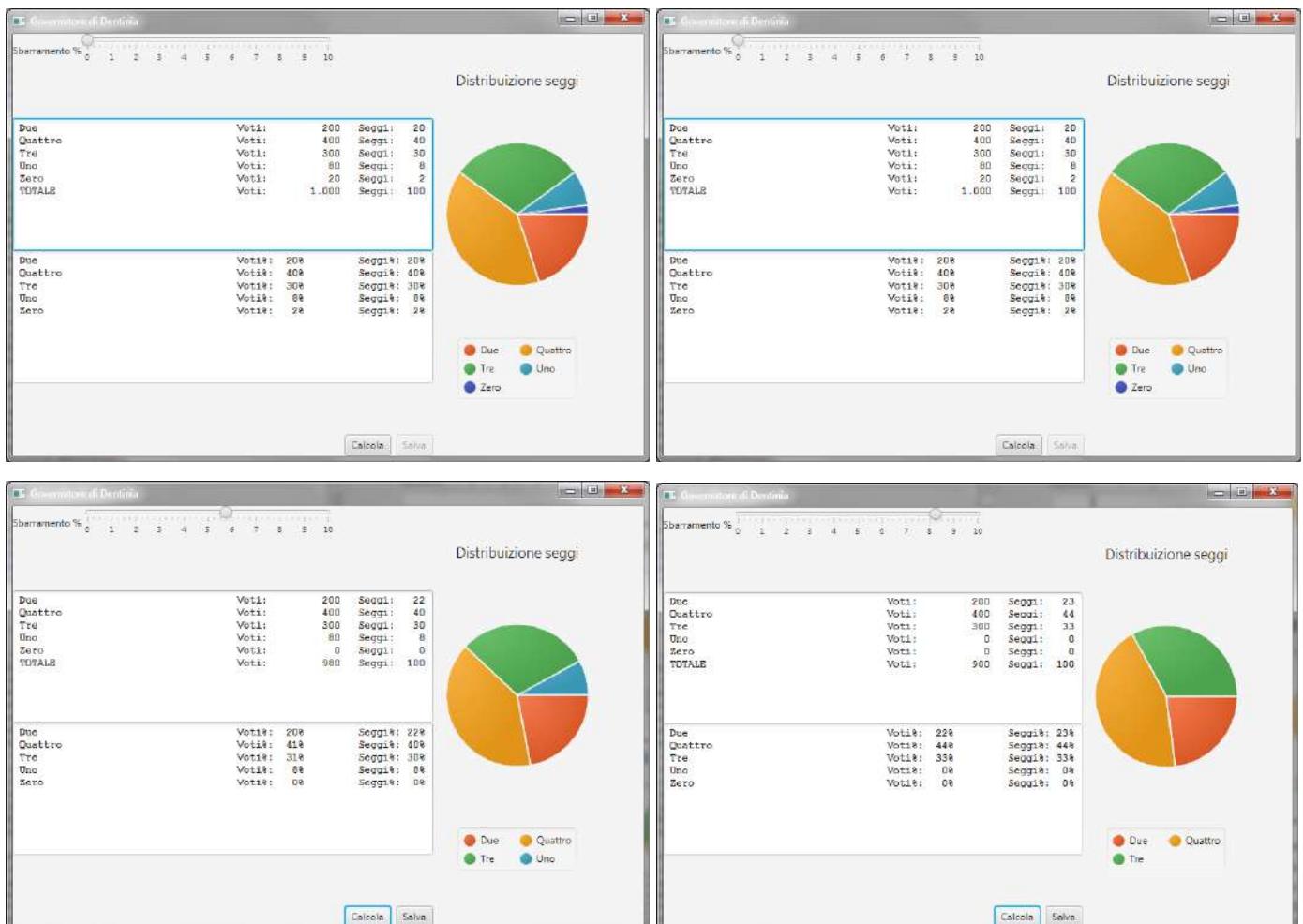
L'interfaccia utente deve essere simile (non necessariamente identica) all'esempio mostrato nelle figure seguenti.

L'architettura segue il modello sotto illustrato:



La classe **ElectionApplication** (fornita) costituisce l'applicazione JavaFX che si occupa di aprire il file, il controller e incorporare l'**ElectionPane** (da realizzare). Per consentire di collaudare la GUI anche in assenza della parte di persistenza, è possibile avviare l'applicazione mediante la classe **ElectionApplicationMock**.

L'interfaccia utente deve essere simile (non necessariamente identica) all'esempio mostrato nella figura seguente.



La classe **ElectionPane** (da realizzare), che estende **BorderPane**, prevede:

- 1) in alto, uno **Slider** configurato da 0 a 10, passo 1, inizialmente a 0, per impostare lo sbarramento %
- 2) a sinistra, due **TextArea** una sotto l'altra: la prima mostra la situazione delle elezioni in termini di voti e seggi assegnati, mentre la seconda esprime le stesse informazioni in percentuale. All'inizio mostrano la situazione corrispondente all'assenza di sbarramento.
- 3) A destra, un **PieChart** (senza etichette) mostra anch'esso la situazione corrente, in forma grafica.
- 4) In basso, due pulsanti Calcola e Salva, di cui il secondo inizialmente disabilitato, controllano il funzionamento dell'applicazione. *Calcola* scatena il calcolo dei seggi, tenendo conto dello sbarramento prescelto e aggiornando tabelle e grafico a torta: dopo ogni ricalcolo ri-abilita il pulsante *Salva* per consentire il salvataggio su file dei nuovi risultati. Il pulsante *Salva* effettua il salvataggio dei risultati sul file, auto-disabilitandosi subito dopo.

NB: mentre i dati della prima tabella sono facilmente ottenibili dal model, i secondi vanno calcolati!

Cose da ricordare

- salva costantemente il tuo lavoro, facendo ZIP parziali e consegne parziali (vale l'ultima)
- in particolare, se ora compila e stai per fare modifiche, salva la versione attuale (non si sa mai..)

Checklist di consegna

- Hai fatto un unico file ZIP (**non .7z!!!**) contenente l'intero progetto?
In particolare, ti sei assicurato di aver incluso tutti i file .java (e non solo i .class)?
- Hai controllato che si compili e ci sia tutto ? [NB: non serve includere il PDF del testo]
- Hai rinominato IL PROGETTO esattamente come richiesto?
- Hai chiamato IL FILE ZIP esattamente come richiesto?
- Hai chiamato la cartella del progetto esattamente come richiesto?
- Dopo aver caricato il file su Esamix, hai premuto il tasto “CONFERMA”, ottenendo il messaggio “Hai concluso l'esame”?

ESAME DI FONDAMENTI DI INFORMATICA T-2 del 9/1/2020

Proff. E. Denti, R. Calegari, A. Molesini – Tempo: 4 ore

NOME PROGETTO ECLIPSE: CognomeNome-matricola (es. RossiMario-0000123456)

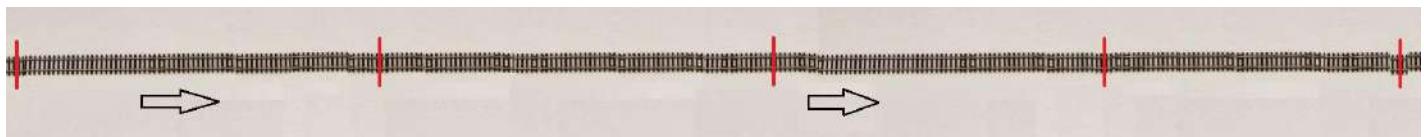
NOME CARTELLA PROGETTO: CognomeNome-matricola (es. RossiMario-0000123456)

NOME ZIP DA CONSEGNARE: CognomeNome-matricola.zip (es. RossiMario-0000123456.zip)

NB: l'archivio ZIP da consegnare deve contenere l'intero progetto Eclipse

Si ricorda che compiti non compilabili o palesemente lontani da 18/30 NON SARANNO CORRETTI e causeranno la verbalizzazione del giudizio "RESPINTO"

Si vuole sviluppare il software di controllo per consentire a due o più treni di circolare su un plastico ferroviario a ovale, in modo automatizzato, *senza mai tamponarsi*. Per praticità, il tracciato è disegnato comunque in orizzontale: i treni che “escono” dal bordo destro si considerano “rientrati” dal bordo sinistro. Coerentemente, il bordo sinistro assume la progressiva km 0.0 (inclusa), il bordo destro quella pari alla lunghezza massima del tracciato (esclusa).



DESCRIZIONE DEL DOMINIO DEL PROBLEMA

Il tracciato, a ovale, si suppone percorso sempre in un unico verso. Per evitare che due treni possano tamponarsi, il percorso (come al vero) è suddiviso in **sezioni**: ogni sezione può essere occupata in un dato istante da un solo treno.

A tal fine è protetta all'ingresso da un segnale (semaforo), che si dispone al rosso se la sezione successiva è occupata.

Ogni treno deve poter essere contenuto interamente all'interno di una qualsiasi sezione: perciò, la lunghezza massima di un treno è inferiore alla lunghezza della sezione più corta (nell'esempio sopra, s4).

Inoltre, per evitare la situazione di potenziale blocco generalizzato, con tutti i treni fermi a un semaforo rosso, il numero massimo di treni che possono circolare è pari a N-2, essendo N il numero delle sezioni: ciò garantisce che almeno un treno abbia sempre davanti una sezione libera in cui avanzare.

IPOTESI SEMPLIFICATIVA

Nel caso in questione, per semplicità, si suppone che ogni treno – quando non è fermo – si muova sempre a una velocità costante, caratteristica di quel treno.

ALGORITMO

Il software non ha in sé alcuna nozione di tempo: per simulare lo scorrere del tempo, si suppone che l'utente prema un **apposito pulsante CLOCK**. Il software reagirà ricalcolando le posizioni potenziali dei treni in base alle rispettive velocità: che poi essi possano avanzare o no, dipende se la sezione successiva è libera o occupata. In particolare, detta:

- v_i la velocità caratteristica del treno T_i
- pos_i la posizione in un dato istante del treno T_i

quando l'utente preme il pulsante CLOCK, la posizione viene ricalcolata secondo la formula $pos_i(t+\Delta t) = pos_i(t) + v_i * \Delta t$. Se tale posizione è ancora all'interno della sezione corrente, il treno avanza alla nuova posizione; se, invece, è oltre il confine della sezione successiva, il treno avanza solo se essa è libera, altrimenti rimane fermo alla posizione corrente.

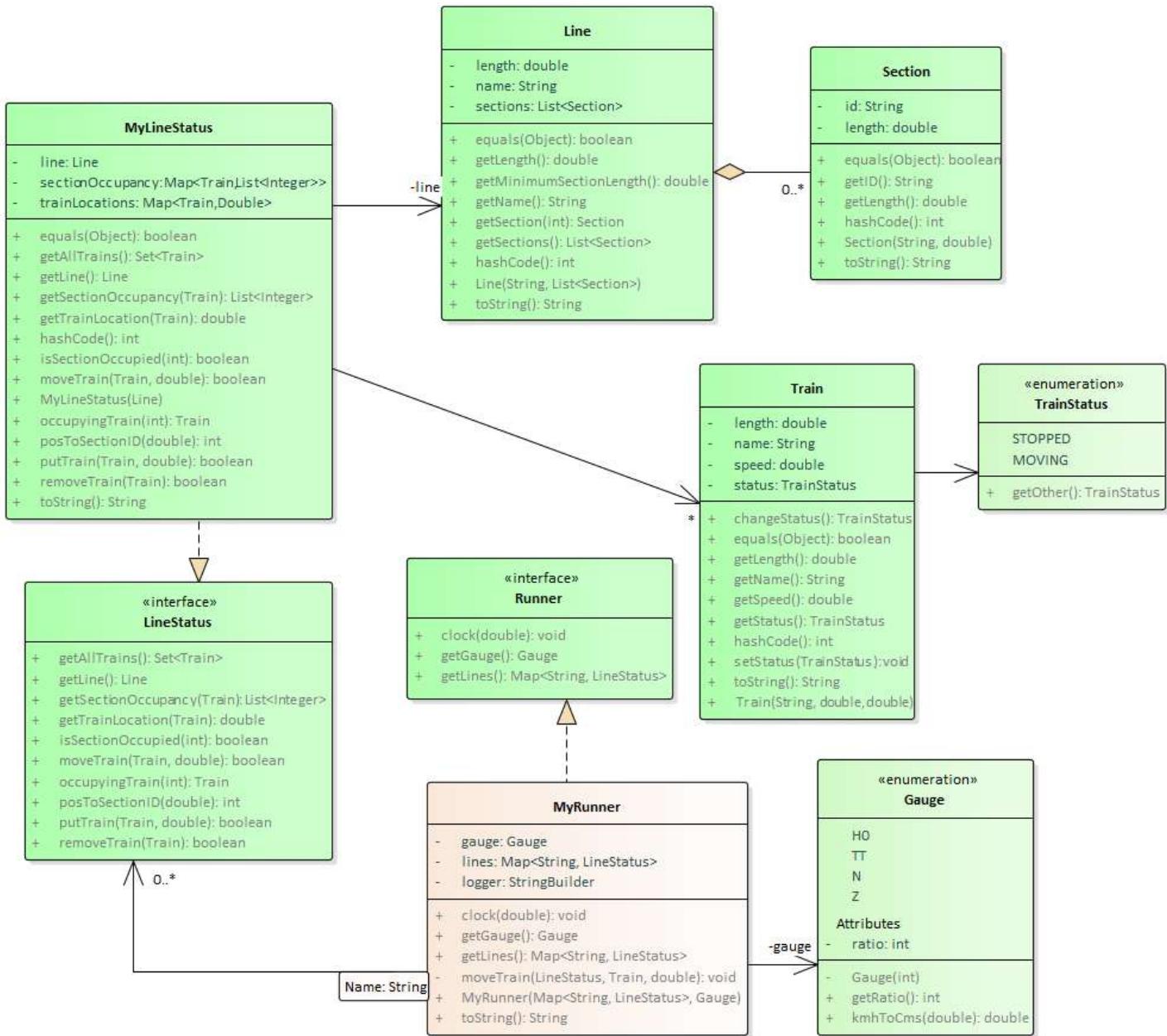
Parte 1

(punti: 22)

Dati (namespace minirail.model)

(punti: 12)

Il modello dei dati deve essere organizzato secondo il diagramma UML di seguito riportato:



SEMANTICA:

- L'enumerativo **Gauge** (fornito) elenca le scale del modellismo ferroviario, unitamente ai rispettivi rapporti di riduzione: ad esempio, la scala H0 è 87 volte più piccola del reale, la scala N 160 volte, etc. Il metodo di utilità `kmhToCms` converte una velocità reale, espressa in km/h, nella corrispondente velocità del trenino modello, espressa in cm/s *in quella specifica scala* (ad esempio, 160 km/h diventano 27.8 cm/s in scala N, o 51.1 in H0).
- La classe **Section** (fornita) rappresenta una sezione del tracciato, caratterizzata da un identificativo univoco e dalla sua lunghezza (l'unità di misura è irrilevante e non è quindi modellata). Si noti che si tratta di un concetto puramente geometrico, non connesso all'idea di tracciato o alla presenza di treni (e all'essere libera/occupata).
- La classe **Line** (fornita) rappresenta una linea intesa come sequenza di sezioni: anche questo è un concetto puramente geometrico. Essa è caratterizzata da un nome e dalla sua lunghezza (la somma delle lunghezze delle sezioni componenti). Alcuni metodi di utilità consentono di recuperare la lunghezza della sezione più corta, o la sezione i-esima della sequenza.

- d) L'interfaccia ***LineStatus*** (fornita) cattura la nozione di linea percorsa da treni: essa incapsula una linea, recuperabile dall'accessor ***getTrain***, e offre metodi per:
- posizionarvi sopra (***putTrain***) o rimuovervi i treni (***removeTrain***)
 - muovere un treno in una nuova posizione (***moveTrain***) [purché il treno si stia muovendo]
 - recuperare tutti i treni presenti sulla linea (***getAllTrains***)
 - recuperare la posizione attuale di un treno (***getTrainLocation***) o le sezioni da esso occupate (***getSectionOccupancy***)
 - sapere se una data sezione sia occupata (***isSectionOccupied***) e da quale treno (***occupyingTrain***)
 - convertire una posizione assoluta nel numero di sezione corrispondente (***posToSectionID***)
- e) La classe ***MyLineStatus*** (fornita) implementa tale interfaccia
- f) La classe ***Train*** (fornita) rappresenta un treno, caratterizzato da nome, lunghezza, velocità e stato (fermo o in movimento, espresso dall'enumerativo ***TrainStatus***). Volutamente, queste descrizioni sono a-dimensionali, potendosi utilizzare la classe sia per rappresentare treni reali, sia modellini.
- g) L'enumerativo ***TrainStatus*** esprime i due stati in cui il treno può trovarsi, fermo o in movimento; un metodo di utilità consente di ottenere facilmente l'opposto dello stato corrente.
- h) L'interfaccia ***Runner*** (fornita) cattura la nozione di simulatore del movimento dei treni nel plastico: incapsula una o più linee, espresse da altrettanti ***LineStatus***, e l'informazione sulla scala (un oggetto ***Gauge***). Offre metodi per:
- recuperare le linee, sotto forma di mappa (nome, LineStatus) (***getLines***)
 - recuperare la scala (***getGauge***)
 - far avanzare tutti i treni in base al tempo trascorso (metodo ***clock***), tenendo conto della loro velocità in scala e dell'occupazione delle varie sezioni
- i) La classe ***MyRunner*** (da realizzare) implementa ***Runner*** come sopra descritto. In particolare, il metodo ***clock***, per ogni linea sottoposta alla sua gestione, deve:
- recuperare il corrispondente ***LineStatus***, che ne incorpora tutto lo stato
 - recuperare tutti i treni presenti su tale linea e farli muovere uno ad uno (in un ordine imprecisato)

Si suggerisce di incapsulare in un metodo ausiliario ***moveTrain*** la logica di movimento di un singolo treno su una data linea. In base a quanto descritto nel Dominio del problema, tale metodo dovrà:

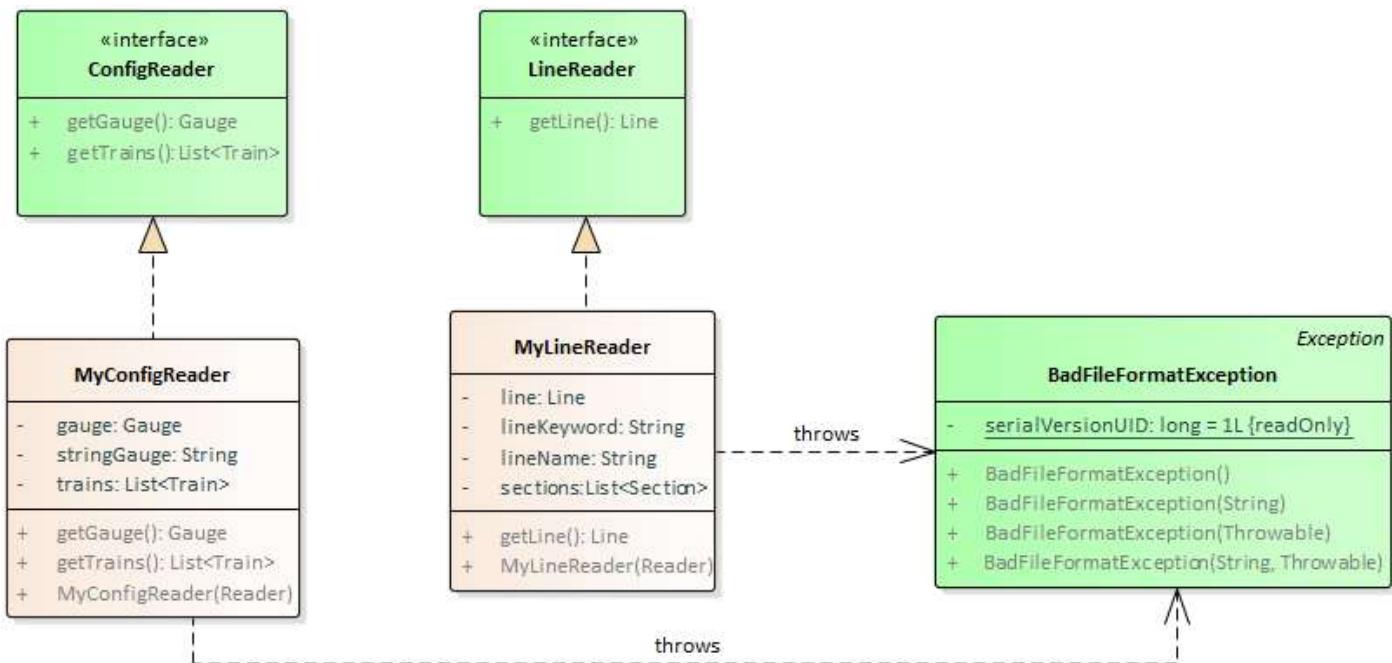
- recuperare la posizione attuale del treno
- calcolare lo spazio percorso nel tempo dato in quella scala
- calcolare la nuova posizione potenziale del treno, che esso andrà a occupare SE potrà effettivamente avanzare: a tale scopo occorre tenere conto che, trattandosi di un ovale, le posizioni vanno intese "modulo L", essendo L la lunghezza del tracciato (quindi, ad esempio, un treno che si trovi alla posizione 310 e debba teoricamente avanzare alla posizione 330 in un tracciato lungo 320 dovrà in realtà avere come nuova posizione 10, ossia 330 modulo 320)
- delegare l'effettiva azione di movimento al metodo ***moveTrain*** di ***LineStatus***
- tracciare quanto fatto in uno ***StringBuilder*** a uso interno, utile per poter ridefinire *toString* in modo che restituiscia un resoconto completo (log) di tutte le azioni fatte e tentate.

A tal fine, ***MyRunner*** deve quindi mantenere internamente uno ***StringBuilder***, su cui ***moveTrain*** possa scrivere ciò che fa. (un esempio di output è visibile nelle immagini dell'applicazione alla fine)

Persistenza (namespace minirail.persistence)

(punti: 10)

Questo package definisce due componenti: il **GameSaver** per stampare su file (*gameover.txt*) lo stato di una partita, e il **ConfigReader** per leggere da file di testo (*config.txt*) la configurazione iniziale (dimensione scacchiera e numero mine).



SEMANTICA:

- l'interfaccia **ConfigReader** (fornita) dichiara i due metodi ***getGauge*** e ***getTrains***, dall'ovvio significato
- la classe **MyConfigReader** (**da realizzare**) implementa tale interfaccia: il costruttore riceve un **Reader** già aperto, da cui legge le righe di configurazione nel formato sotto specificato. *La prima riga definisce la scala, le successive i treni.* È il costruttore a svolgere tutto il lavoro di lettura, che deve prevedere un'accurata validazione dell'input (eventuali campi mancanti o nulli, separatori errati, etc.): in caso di problemi nel formato delle righe, il costruttore deve lanciare ***BadFormatException*** (fornita) con apposito messaggio d'errore, mentre eventuali problemi di I/O devono essere lasciati fluire all'esterno come ***IOException***. Quanto letto deve essere memorizzato in due proprietà interne, restituite dai due metodi ***getGauge*** e ***getTrains***.

FORMATO DEL FILE DI CONFIGURAZIONE: la prima riga contiene la scala e, dopo uno o più spazi, la parola “gauge” (scritta con qualunque sequenza di caratteri maiuscoli e/o minuscoli). Le righe successive descrivono ciascuna un treno, tramite una serie di dati separati da virgole: nell’ordine si trovano l’identificativo del treno (senza spazi), la lunghezza del treno seguita dall’unità di misura (“cm” o “in”), la velocità del treno al vero seguita anche in questo caso dalla rispettiva unità di misura (“km/h” o “mph”): i valori numerici sono numeri reali, eventualmente con parte decimale espressa nell’usuale notazione con “.”.

ESEMPIO DI FILE LECITO:

N gauge
IC583, 65 cm, 160 km/h
R2961, 68 cm, 140 km/h

- l'interfaccia **LineReader** (fornita) dichiara il solo metodo ***getLine***, che legge dal file i dati di una singola linea
- la classe **MyLineReader** (**da realizzare**) implementa tale interfaccia: il costruttore riceve un **Reader** già aperto, da cui legge la descrizione nel formato sotto specificato. *La prima riga dichiara il nome della linea, le successive le sezioni di cui è composta, in quell'ordine.* Come nel caso precedente, è il costruttore a svolgere tutto il lavoro di lettura, che deve prevedere un'accurata validazione dell'input (eventuali campi mancanti o nulli, separatori errati, etc.): in caso di problemi nel formato delle righe, il costruttore deve lanciare ***BadFormatException*** (fornita)

con apposito messaggio d'errore, mentre eventuali problemi di I/O devono essere lasciati fluire all'esterno come ***IOException***. La linea dev'essere memorizzata in una proprietà interna, restituita da ***getLine***

FORMATO DEL FILE DI LINEA: la prima riga contiene il nome della linea, preceduto dalla parola "Line" (scritta con qualunque sequenza di caratteri maiuscoli e/o minuscoli). Le righe successive descrivono ciascuna una sezione, tramite una serie di dati separati da spazi: nell'ordine si trovano la parola "Section", l'identificativo della sezione stessa (senza spazi) e infine la sua lunghezza (un numero reale, eventualmente con parte decimale espressa nell'usuale notazione con ".").

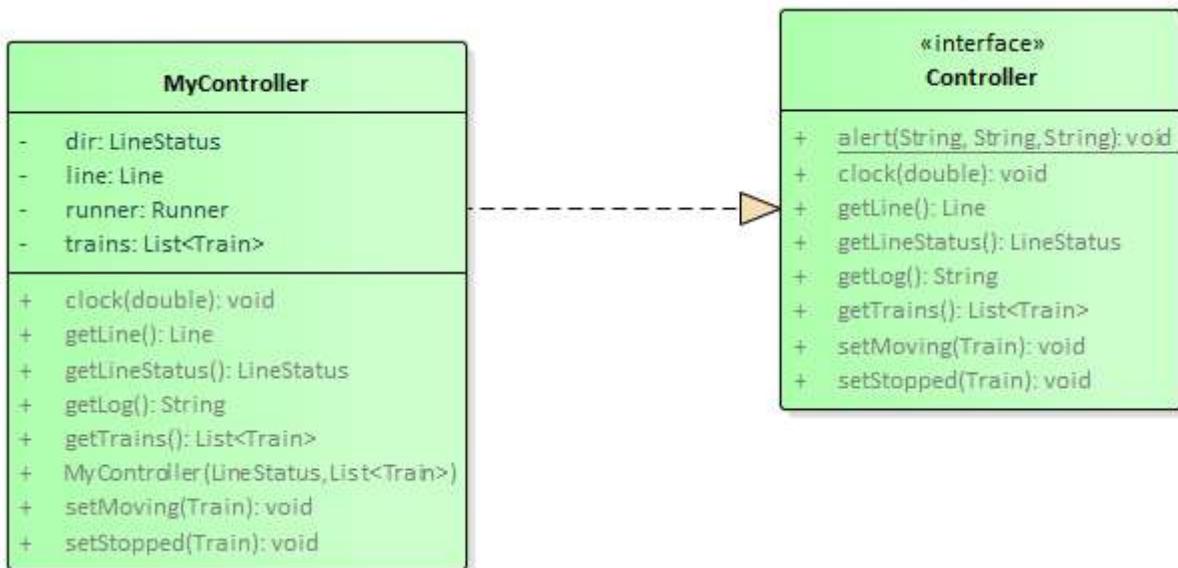
ESEMPIO DI FILE LECITO:

```
Line A-D
Section A-B 90
Section B-C 90
Section C-D 70
Section D-A 70
```

Parte 2

(punti: 8)

Controller (namespace minirail.controller)



Il controller – articolato in interfaccia e implementazione – è fornito già pronto: esso crea e gestisce il **Runner** che governa il plastico, sfruttando a tale scopo il ***LineStatus*** ricevuto e facendo circolare i treni ricevuti come secondo argomento all'atto della costruzione. Conseguentemente, i suoi metodi fanno da ponte con tali entità. In particolare:

- ***getLineStatus*** e ***getLine*** e ***getTrains*** recuperano lo stato della linea, la linea e la lista di treni
- ***setMoving*** e ***setStopped*** impostano lo stato di un dato treno rispettivamente a "in movimento" o "fermo"
- ***getLog*** recupera la stringa prodotta dal runner, che riporta tutto l'accaduto nel sistema fino a questo istante
- ***clock*** attiva l'omonimo metodo del ***Runner***, catturando l'eccezione in modo da mostrarla in un dialogo di ***Alert***.

L'interfaccia **Controller** offre altresì il metodo statico ***alert*** per far comparire una finestra di dialogo utile a segnalare errori all'utente (in questa applicazione, solo nel caso in cui la stampa su file fallisca per qualche motivo).

(segue)

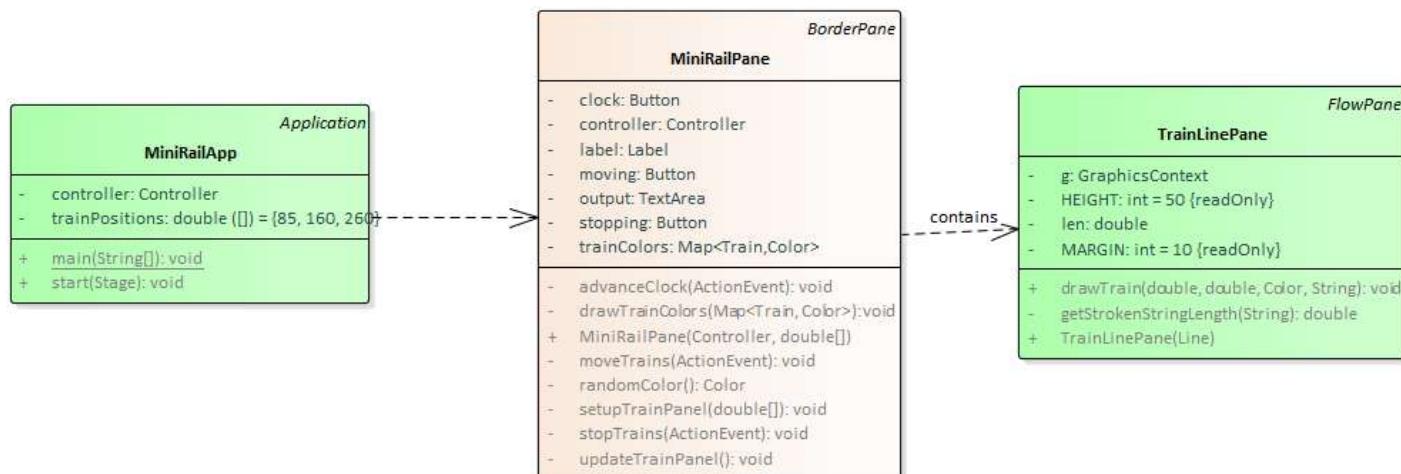
In caso di malfunzionamento dei reader, l'app è già preconfigurata per utilizzare valori di default anziché leggerli da file.

L'interfaccia grafica dev'essere simile (non necessariamente identica) a quella illustrata alla pagina seguente.

All'inizio (Fig.1), la GUI mostra il grafico della linea, in sezioni di lunghezza proporzionale a quella effettiva: sotto sono riportate le distanze progressive dal bordo sinistro (conventionalmente 0). In alto, due pulsanti consentono di cambiare lo stato dei treni (tutti assieme): all'inizio, come mostra l'etichetta in alto, sono tutti fermi (STOPPED). In questo stato, tentando di far partire il simulatore, premendo il **pulsante Clock**, si ottiene un messaggio d'errore (Fig. 2).

Il primo passo da fare è quindi portare lo stato dei treni a MOVING (Fig. 3) premendo l'apposito pulsante: il sistema reagisce modificando l'etichetta in alto. A questo punto è possibile far partire la simulazione: premendo **Clock**, i treni iniziano a muoversi secondo le rispettive velocità, calcolate con periodo di 1/2 secondo (Figg. 4,5,6). A ogni **Clock** i treni avanzano, o restano fermi rispettando le sezioni già occupate. Visivamente, se un treno "esce" dal bordo destro ricompare simultaneamente al bordo sinistro (Fig. 7, 8) per la parte eccedente, così da simulare il circuito ovale.

In qualunque momento è possibile fermare i treni e farli ripartire, coi due pulsanti in alto.



- La classe **MiniRailApp** (fornita) contiene il main di partenza dell'applicazione.
- La classe **TrainLinePane** (fornita) contiene la logica di visualizzazione della linea coi relativi treni:
 - il costruttore riceve la linea da visualizzare
 - il metodo **drawTrain** mostra alla posizione data (pos) un treno di lunghezza data (trainLen), nel colore specificato (color) e scrivendoci dentro l'identificativo passato (trainID).

*Da notare che la visualizzazione non è più modificabile dopo la creazione del pannello: pertanto, per far avanzare i treni occorrerà rimpiazzare l'istanza corrente di **TrainLinePane** con una nuova, aggiornata.*
- La classe **MiniRailPane** (da realizzare) è una specializzazione di **BorderPane** che contiene la GUI dell'applicazione. Come descritto sopra, quindi, contiene:
 - in alto, i due pulsanti **Move trains / Stop trains** per muovere/fermare i treni, con relativa label
 - al centro, il **TrainLinePane** che visualizza graficamente la linea con i treni
 - in basso, la textarea su cui mostra, dopo ogni clock, lo stato attuale del runner (metodo **getLog**)
 - a destra, il pulsante **Clock**: a ogni pressione, occorre
 - far avanzare il clock del controller di $\frac{1}{2}$ secondo
 - emettere sull'area di testo il log del controller

- aggiornare la visualizzazione grafica, costruendo e istanziando un nuovo **TrainLinePane** con le posizioni dei treni aggiornate, da sostituire al precedente

Il costruttore riceve un array di double da utilizzare come *posizioni iniziali* dei treni, nell'ordine con cui essi sono presenti nel file (e quindi in lista). **È sua responsabilità attribuire a ogni treno un colore univoco e mantenerlo coerente durante la simulazione. Il numero di colori non può essere prestabilito e cablato nel codice, deve essere adattato allo specifico numero di treni presenti.**

Si suggerisce di strutturare la classe appoggiandosi a una serie di metodi privati ausiliari, come da diagramma UML (non obbligatorio).

Cose da ricordare

- salva costantemente il tuo lavoro, facendo ZIP parziali e consegne parziali (vale l'ultima)
- in particolare, se ora compila e stai per fare modifiche, salva la versione attuale (non si sa mai..)

Checklist di consegna

- Hai fatto un unico file ZIP (**non .7z!!!**) contenente l'intero progetto?
In particolare, ti sei assicurato di aver incluso tutti i file .java (e non solo i .class)?
- Hai controllato che si compili e ci sia tutto ? [NB: non serve includere il PDF del testo]
- Hai rinominato IL PROGETTO esattamente come richiesto?
- Hai chiamato IL FILE ZIP esattamente come richiesto?
- Hai chiamato la cartella del progetto esattamente come richiesto?
- Dopo aver caricato il file su Esamix, hai premuto il tasto “CONFERMA”, ottenendo il messaggio “Hai concluso l’esame”?

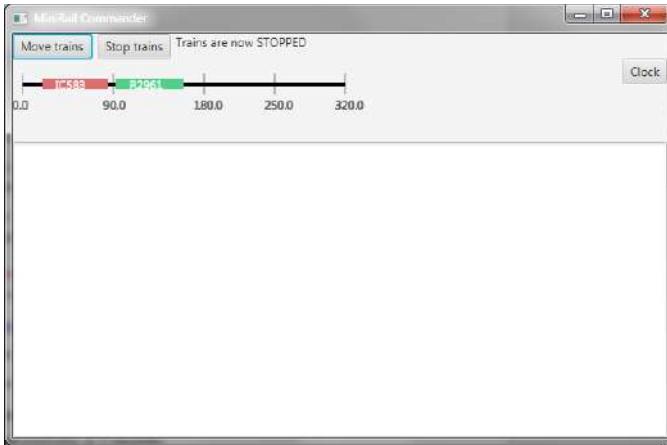


Figura 1

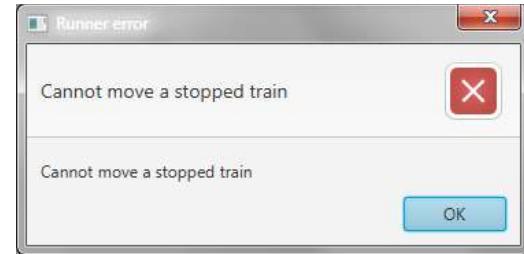


Figura 2

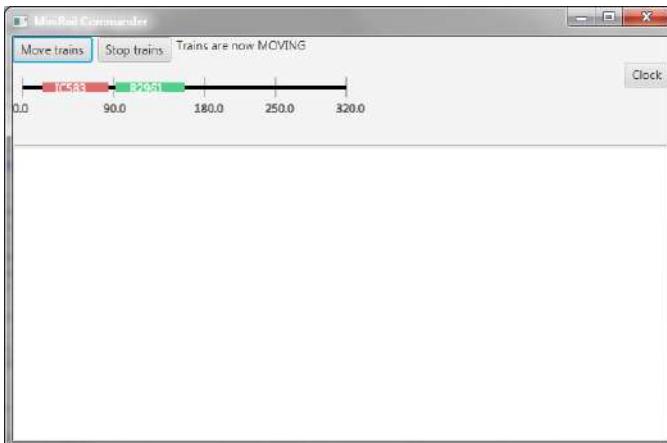


Figura 3

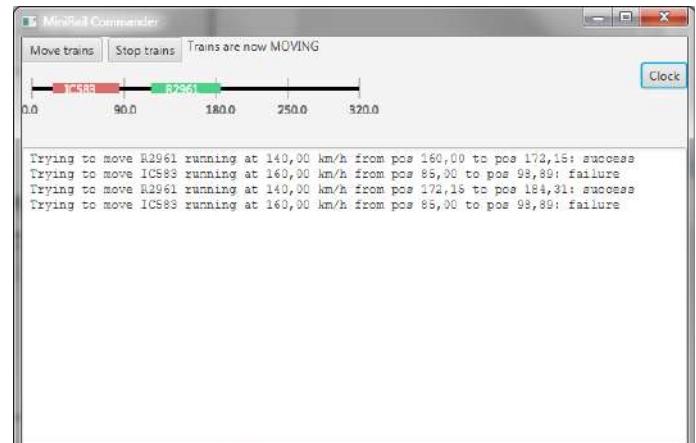


Figura 4

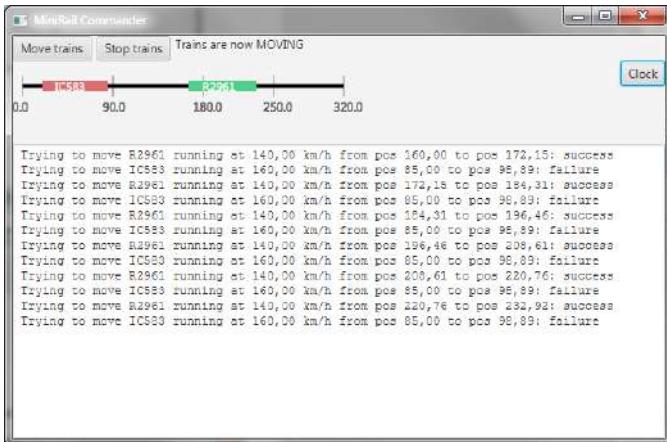


Figura 5

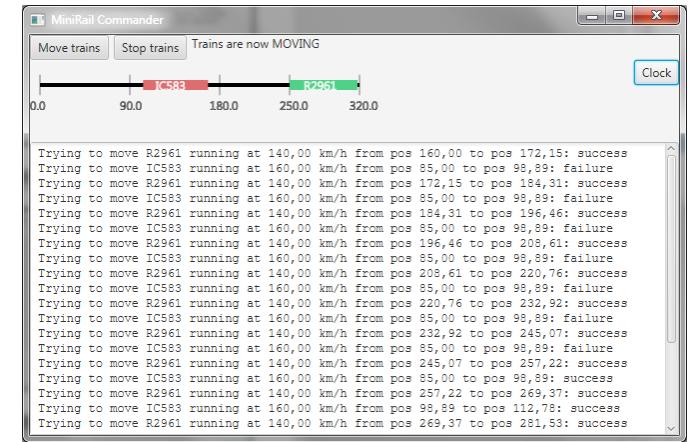


Figura 6

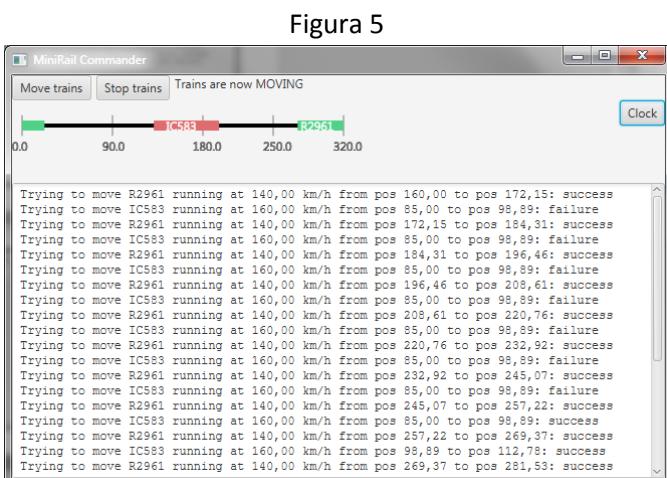


Figura 7

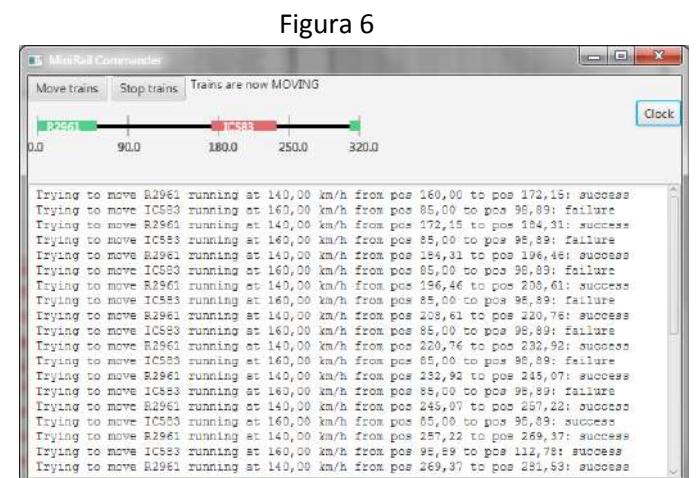


Figura 8

ESAME DI FONDAMENTI DI INFORMATICA T-2 del 10/9/2019

Proff. E. Denti, R. Calegari, A. Molesini – Tempo: 4 ore

NOME PROGETTO ECLIPSE: CognomeNome-matricola (es. RossiMario-0000123456)

NOME CARTELLA PROGETTO: CognomeNome-matricola (es. RossiMario-0000123456)

NOME ZIP DA CONSEGNARE: CognomeNome-matricola.zip (es. RossiMario-0000123456.zip)

NB: l'archivio ZIP da consegnare deve contenere l'intero progetto Eclipse

Si ricorda che compiti *non compilabili o palesemente lontani da 18/30* NON SARANNO CORRETTI e causeranno la verbalizzazione del giudizio “RESPINTO”

Si vuole sviluppare un'app per giocare a *Campo Minato* (detto anche *Campo Fiorito*, *Mine Sweeper* o *Mines Sweeper*).

DESCRIZIONE DEL DOMINIO DEL PROBLEMA

Campo Minato è un solitario che unisce una componente di fortuna a una, prevalente, di abilità. Si gioca su una scacchiera NxN (la dimensione N dipende dal grado di difficoltà desiderato) su cui il computer dispone M mine ($M \ll N^2$), in posizioni sconosciute al giocatore. Lo scopo del gioco è sminare il campo minato, rivelando via via tutte le caselle ma senza toccare mai quelle con le mine: se ciò malauguratamente accade, il giocatore salta per aria e la partita è persa. Se, invece, si riescono a rivelare tutte le caselle evitando quelle con le mine, il giocatore vince la partita.

Per guidare il giocatore nell'attività di sminamento, ogni casella senza mine contiene un *numero che indica quante mine ci sono nelle (max 8) caselle adiacenti*: spesso, al posto dello 0 la casella viene lasciata vuota. Nel caso si riveli una casella con zero mine adiacenti (ossia vuota), *vengono svelate anche le caselle numeriche a loro volta adiacenti a quest'ultima*, ricorsivamente, così da rendere più veloce e accattivante il gioco.

ALGORITMO DI GIOCO

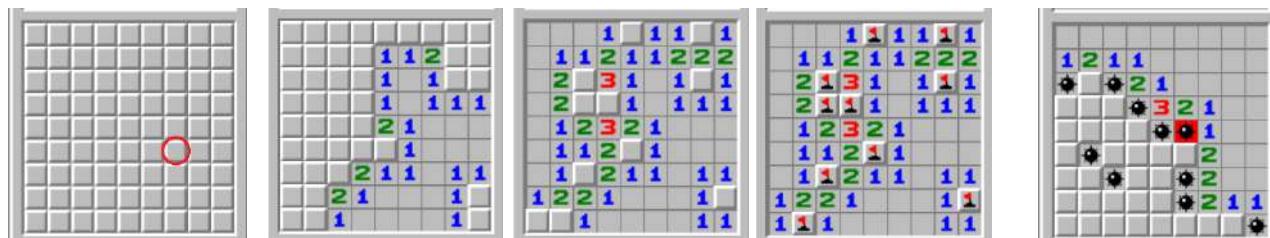
Inizialmente il giocatore svela (facendoci clic sopra) una casella, sperando di non incappare subito in una mina: qui chiaramente la componente fortuna è assoluta. Se la casella contiene un valore numerico, che indica il numero di mine adiacenti, il giocatore può sfruttare tale informazione per decidere quale casella svelare alla prossima mossa (ossia dove cliccare successivamente), operando con abilità. In ogni momento, se il giocatore malauguratamente svela una casella con una mina, salta per aria e perde la partita; se, invece, il giocatore riesce via via a svelare tutte le caselle evitando tutte quelle che contengono mine, vince la partita. Il numero M di mine è naturalmente noto a priori.

ALGORITMO DI RIEMPIMENTO INIZIALE DELLA SCACCHIERA

La parte più interessante del gioco è, paradossalmente, quella che il giocatore non vede – l'algoritmo con cui si calcolano i valori da porre nelle caselle numeriche (quelle che non contengono mine): essi indicano il numero di mine (da 0 a 8) presenti nelle (max 8) caselle adiacenti. A tal fine occorre considerare, per ogni cella di posizione (c,r), le caselle che la circondano, dalla riga r-1 alla riga r+1, e dalla colonna c-1 alla colonna c+1, avendo però cura di considerare i casi particolari relativi alle caselle nelle prime/ultime righe o colonne (che hanno ovviamente meno di 8 caselle adiacenti).

ESEMPIO

Si riportano di seguito alcuni screenshot di una partita su una scacchiera 9x9 con 10 mine. I primi quattro screenshot mostrano una partita vinta, l'ultimo – totalmente distinto dai precedenti – una partita persa. Nel primo caso, dopo varie mosse, nell'ultima azione (terzo screenshot) il giocatore svela l'ultima casella senza mine: il sistema risponde concedendo la vittoria e mostrando (quarto screenshot), a conferma, la posizione di tutte le mine. Nel secondo caso invece il giocatore, mal interpretando le indicazioni numeriche, svela erroneamente una casella con una mina (sfondo rosso) e salta per aria: la partita è persa e il sistema mostra la posizione delle altre mine, per completezza.



Nell'applicazione da sviluppare, per semplicità grafica, al posto delle icone delle mine si userà la lettera ‘M’, mentre al posto delle caselle vuote si utilizzerà il carattere ‘?’; inoltre, non si utilizzeranno i colori (vedere figure in coda al testo).

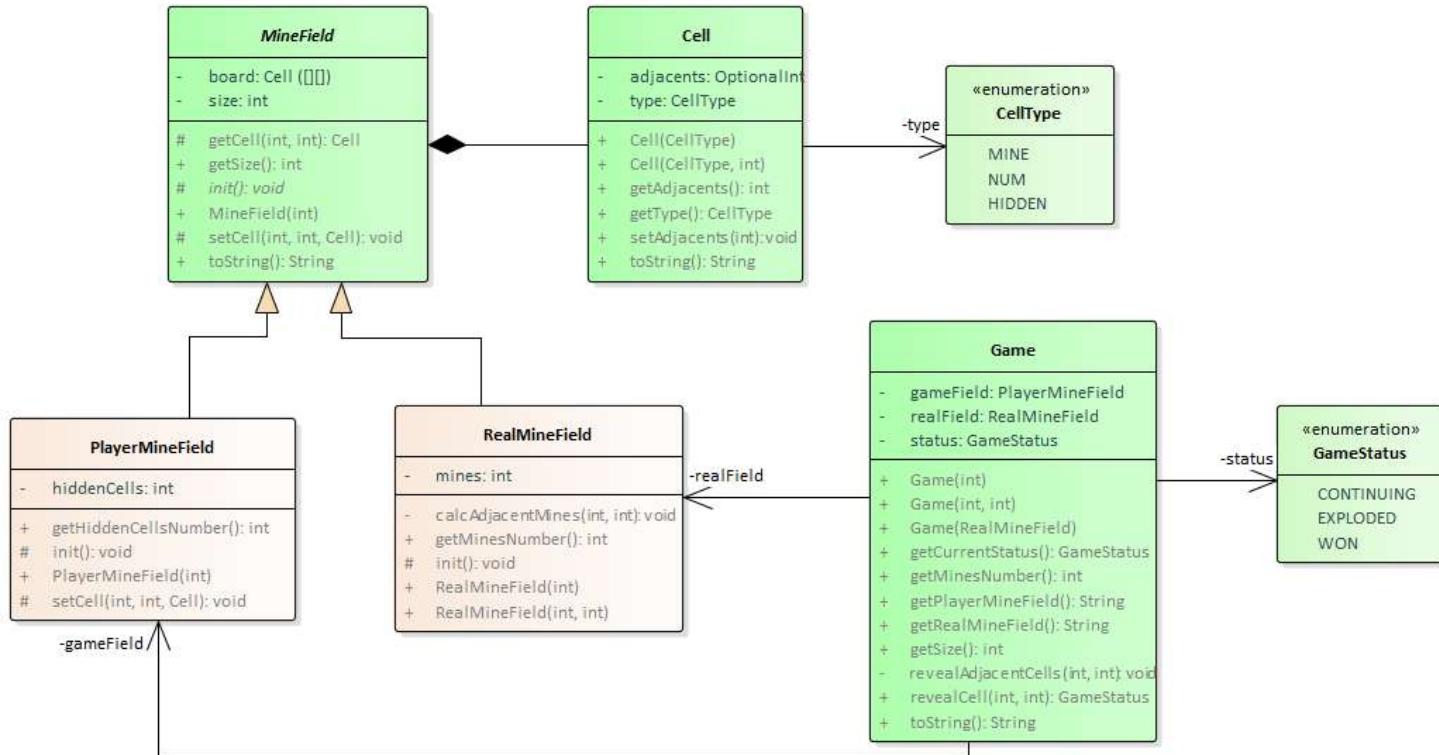
Parte 1

(punti: 24)

Dati (namespace minesweeper.model)

(punti: 17)

Il modello dei dati deve essere organizzato secondo il diagramma UML di seguito riportato:



SEMANTICA:

- L'enumerativo **CellType** (fornito) definisce i tre stati possibili per ogni cella della scacchiera: MINE, NUM, HIDDEN. (Come si vedrà meglio nel seguito, la scacchiera nascosta creata dal computer ha solo celle di tipo MINE e NUM, mentre quella del giocatore ha inizialmente celle tutte HIDDEN, destinate a essere poi sostituite da celle NUM o MINE via via che le celle vengono svelate.)
- La classe **Cell** (fornita) modella una cella della scacchiera: è caratterizzata dal suo tipo (**CellType**) e optionalmente da un intero (utile nel solo caso in cui sia di tipo NUM). Il costruttore a singolo argomento è destinato principalmente a costruire celle di tipo MINE o HIDDEN, mentre quello a due argomenti è specifico per le celle di tipo NUM. Appositi accessori (**getAdjacents**/**setAdjacents**) permettono di recuperare o impostare il valore numerico per le celle di tipo NUM, o di recuperarne il tipo (**getType**). Il metodo **toString** emette, come da specifica del dominio del problema, rispettivamente la stringa “M” per le celle di tipo MINE, “?” per le celle di tipo HIDDEN, e il valore numerico corrispondente per le celle di tipo NUM.
- La classe astratta **MineField** (fornita) implementa una scacchiera di dimensione **size**: questa classe funge da base sia per la scacchiera del giocatore sia per quella del computer, implementate da due classi derivate (v. oltre). Il costruttore riceve la dimensione della scacchiera e alloca la corrispondente matrice **size** x **size** di **Cell**. Resta astratto solo il metodo (protetto) di inizializzazione **init**, destinato a essere invocato dal costruttore delle sottoclassi per specializzare la logica di riempimento iniziale della scacchiera – che è diversa per la scacchiera nascosta del computer e per quella del giocatore (per questo tale metodo non è invocato nel costruttore di **MineField**). Appositi accessori (**getCell**/**setCell**) permettono di recuperare o impostare la cella situata in una data posizione di riga e colonna, o la dimensione della scacchiera (**getSize**): il metodo **toString** produce una descrizione della scacchiera per righe, con le varie celle separate da tabulazioni e un fine riga dopo ogni riga.
- La classe concreta **PlayerMineField** (da realizzare) specializza **MineField** nel caso della scacchiera del giocatore:
 - poiché essa deve essere inizialmente tutta nascosta, il metodo di inizializzazione **init** deve riempirla tutta con celle di tipo HIDDEN.

- la classe deve definire e mantenere inoltre la proprietà `hiddenCellsNumber` – recuperabile con un apposito metodo accessor `getHiddenCellsNumber` – che rappresenta il numero di celle ancora nascoste (inizialmente tutte): tale valore dev'essere poi decrementato ogni volta che una cella HIDDEN viene sostituita da una cella MINE o NUM (tramite la versione specializzata del metodo `setCell`, descritta al punto seguente)
- poiché nella scacchiera del giocatore l'unica azione di modifica prevista è la sostituzione di una cella HIDDEN con una di tipo NUM o MINE, il metodo `setCell` ereditato da `MineField` dev'essere sovrascritto da una versione specializzata, che:
 - richiami il metodo `setCell` ereditato
 - verifichi che non si stia sostituendo una cella HIDDEN con un'altra HIDDEN (cosa manifestamente assurda), lanciando in tal caso `UnsupportedOperationException` con apposito messaggio d'errore
 - infine, decrementi di 1 il valore della proprietà `hiddenCellsNumber`

In tal modo, il gestore del gioco potrà facilmente identificare quando la partita sia terminata con successo (quando il numero di celle rimaste nascoste è uguale al numero di mine), o quando invece debba continuare (se il numero di celle rimaste nascoste è superiore al numero di mine).

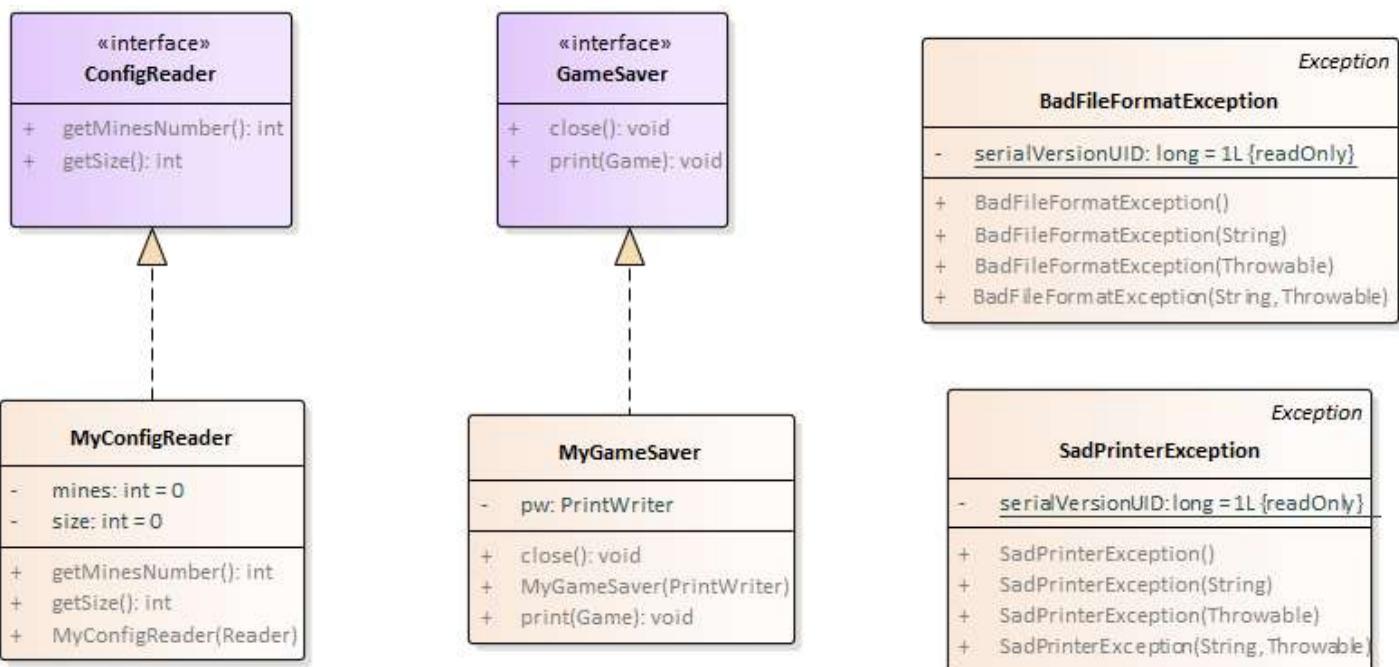
- e) la classe concreta **RealMineField (da realizzare)** specializza `MineField` nel caso della scacchiera del computer:
- il costruttore a un argomento accetta solo la dimensione della scacchiera (`size`), poi rimpalla l'opera sul costruttore a due argomenti, passando per default il valore 10 come numero di mine
 - il costruttore a due argomenti accetta la dimensione della scacchiera (`size`) e il numero di mine (`mines`): dopo aver impostato tali proprietà, richiama il metodo `init` per effettuare il popolamento della scacchiera
 - il metodo di inizializzazione `init`
 - prima, sorteggia `mines` distinte posizioni – ossia coppie (riga, colonna) – in cui piazzare altrettante celle di tipo MINE
 - poi, percorre tutta la scacchiera e inizializza tutte le altre celle con celle di tipo NUM, avendo preventivamente cura di calcolare il numero di mine adiacenti a ciascuna, tramite il metodo privato `calcAdjacentMines(row, col)`
 - il metodo privato ausiliario `calcAdjacentMines` calcola il numero di mine adiacenti a una cella situata in una data posizione di riga (`r`) e colonna (`c`): a tal fine, percorre le (max 8) celle che la circondano, dalla riga `r-1` alla riga `r+1`, e dalla colonna `c-1` alla colonna `c+1`, contando le celle di tipo MINE. Nel farlo deve naturalmente tenere conto dei casi particolari in cui la cella data sia nella prima o ultima riga e/o colonna, nei quali ci sono ovviamente solo 5 o 3 celle adiacenti.
 - SUGGERIMENTO: si consiglia di determinare preventivamente l'escursione massima degli indici di riga (di base, da `r-1` a `r+1`) e di colonna (di base, da `c-1` a `c+1`) onde escludere la riga/colonna precedente nel caso la cella data sia sulla prima riga/colonna, e dualmente la riga/colonna successiva nel caso la cella data sia sull'ultima riga/colonna.
 - NB: naturalmente, nello scorrere gli indici, occorre evitare di considerare la cella stessa!
 - Il metodo accessor `getMinesNumber` restituisce il numero di mine nella scacchiera
- f) L'enumerativo **GameStatus** (fornito) definisce i tre stati possibili del gioco: CONTINUING, EXPLODED, WON.
- g) La classe concreta **Game** (fornita) contiene tutta la logica di gioco:
- i costruttori creano le due scacchiere, una di tipo **RealMinesField** per il computer e una di tipo **PlayerMinesField** per il giocatore, e impostano lo stato iniziale del gioco a CONTINUING.
 - Vari accessori consentono di recuperare lo stato attuale del gioco (`getCurrentStatus`), la dimensione delle scacchiere (`getSize`), il numero di mine (`getMinesNumber`) e lo stato attuale delle due scacchiere sotto forma di stringa (`getPlayerMineField` e `getRealMineField`).

- Il metodo cruciale, ***revealCell***, rivela la cella di coordinate (riga,colonna) specificate, recuperando la cella reale dalla scacchiera del computer e riportandola su quella del giocatore, e aggiornando quindi lo stato del gioco (EXPLDED se la cella rivelata era una mina, CONTINUING o WON in caso contrario). In ossequio alle regole, se la casella rivelata era numerica e conteneva uno 0, si procede ricorsivamente a rivelare anche le celle adiacenti a quest'ultima (tramite il metodo privato ***revealAdjacentCells***): se, dopo tale operazione, il numero di celle ancora nascoste è uguale al numero di mine, la partita è vinta e lo stato del gioco diventa WON: altrimenti, la partita deve continuare e lo stato è CONTINUING.
- Il metodo ***toString*** emette una descrizione completa dello stato attuale del gioco, riportando entrambe le scacchiere con opportune descrizioni.

Persistenza (namespace *minesweeper.persistence*)

(punti: 7)

Questo package definisce due componenti: il ***GameSaver*** per stampare su file (*gameover.txt*) lo stato di una partita, e il ***ConfigReader*** per leggere da file di testo (*config.txt*) la configurazione iniziale (dimensione scacchiera e numero mine).



SEMANTICA:

- l'interfaccia ***ConfigReader*** (fornita) dichiara i due metodi ***getSize*** e ***getMinesNumber***, dall'ovvio significato
- la classe ***MyConfigReader* (da realizzare)** implementa tale interfaccia: il costruttore riceve un ***Reader*** già aperto, da cui legge le due righe di configurazione, nel formato sotto specificato. L'ordine delle due righe non è prestabilito: potrebbe esserci prima la riga relativa alla dimensione o prima quella relativa alle mine, non è dato sapere. È il costruttore a svolgere tutto il lavoro di lettura, memorizzando infine i due valori letti in due proprietà interne, che vengono poi restituite dai due metodi ***getSize*** e ***getMinesNumber***. In caso di problemi di I/O o nel formato delle righe, il costruttore deve lanciare ***BadFormatException*** (fornita) con apposito messaggio d'errore.

FORMATO DEL FILE DI CONFIGURAZIONE: ogni riga contiene una delle due parole “mines” o “size”, scritte con qualunque sequenza di caratteri maiuscoli e/o minuscoli, seguita dal carattere “:” e poi, dopo eventualmente spazi o tabulazioni intermedi, il valore intero corrispondente.

ESEMPI DI FILE LECITI:

MINES: 3	Mines: 3	size: 6
Size: 6	Size: 6	mines:3

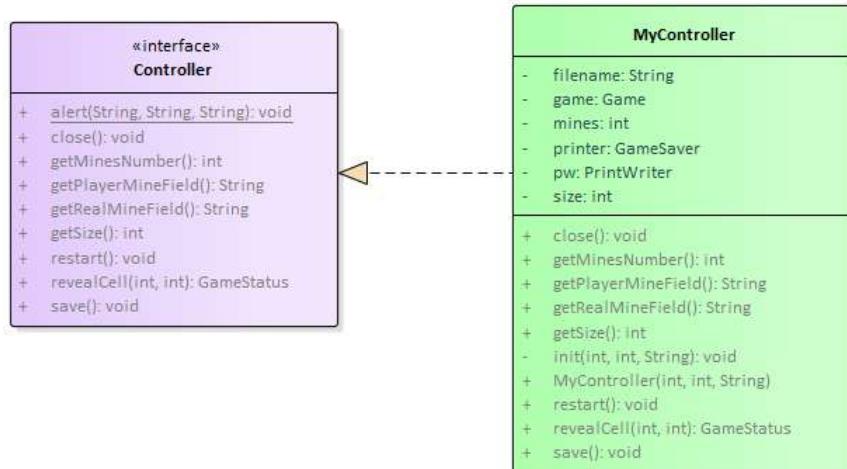
- l'interfaccia ***GameSaver*** (fornita) dichiara i metodi ***print***, che stampa un ***Game***, e ***close***, che chiude il writer di uscita

- d) la classe **MyGameSaver** (da realizzare) implementa tale interfaccia: il costruttore deve ricevere un **PrintWriter**, che poi utilizza nei metodi **print** (per stampare lo stato attuale del gioco) e **close** (per chiudere il writer stesso).
 NB: successive chiamate a **print** causano l'accodamento (append) delle diverse fasi del gioco nello stesso file.

Parte 2

(punti: 6)

Controller (namespace minesweeper.controller)



Il controller – articolato in interfaccia e implementazione – è fornito già pronto: il suo stato interno crea e gestisce il Game con tutto lo stato della partita, nonché il **GameSaver** da usare per le stampe. Conseguentemente, i suoi metodi fanno da ponte con entrambe tali entità. In particolare:

- **save** e **close** richiamano i metodi **print** e **close** del **GameSaver**
- svariati accessori rendono disponibili le proprietà utili del **Game**, richiamando gli omonimi metodi: in particolare, **revealCell** richiama il corrispondente metodo di **Game**, permettendo l'avanzare della partita
- **restart** reinizializza lo stato del controller per una nuova partita (quindi nuovo **Game** e nuovo **GameSaver**)

L'interfaccia **Controller** offre altresì il metodo statico **alert** per far comparire una finestra di dialogo utile a segnalare errori all'utente (in questa applicazione, solo nel caso in cui la stampa su file fallisca per qualche motivo).

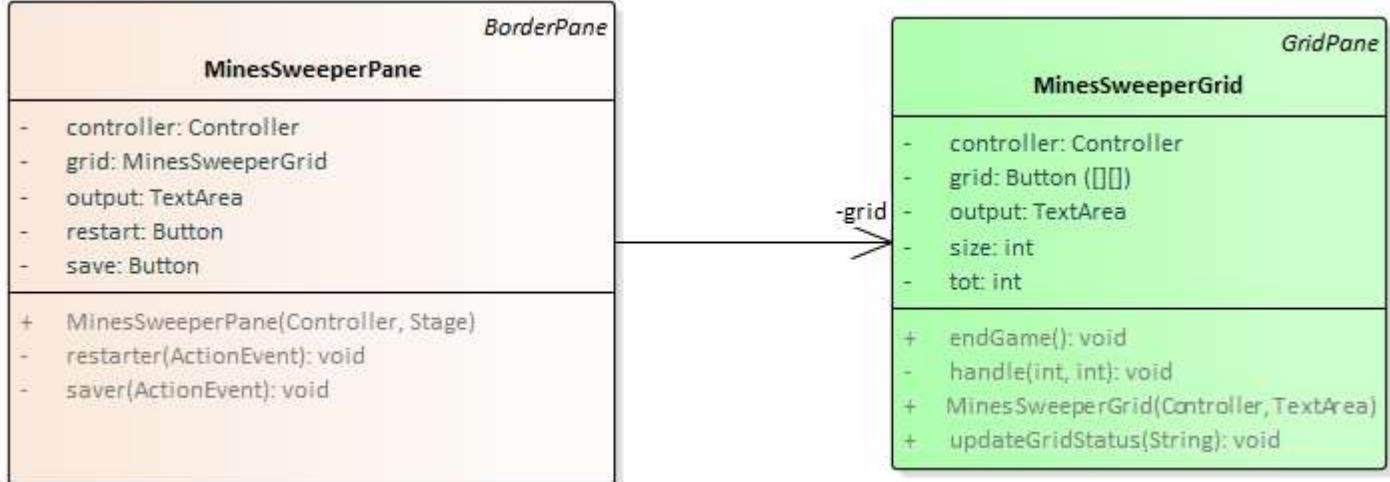
GUI (namespace minesweeper.ui)

(punti: 6)

In caso di malfunzionamento del ConfigReader, usare come main MineSweeperAppMock, che utilizza valori di configurazione di default anziché leggerli da file.

L'interfaccia grafica dev'essere simile (non necessariamente identica) a quella sotto illustrata.

All'inizio (Fig.1), la GUI mostra a sinistra la griglia del giocatore (in questo caso 5x5), a destra l'area di testo, sopra le mine (in questo caso 3) e sotto i pulsanti. Svelando una cella (Fig. 2), il sistema reagisce mostrando la cella cliccata e le eventuali adiacenti (in questo caso no): l'area di testo ripete l'informazione e invita a continuare. Proseguendo con perizia, il giocatore arriva prima o poi a vincere la partita (Fig. 3): in tal caso tutte le celle vengono svelate e l'area di testo mostra, per conferma, anche la griglia reale del computer. Se, invece, il giocatore è sfortunato e clicca su una mina (Fig. 4, sfortunato alla primissima mossa!), il sistema reagisce disabilitando tutta la griglia, mostrando a lato la situazione attuale e la griglia reale, e dando il triste esito dell'esplosione dell'esploratore.



- a) La classe **MinesSweeperApp** (fornita) contiene il main di partenza dell'applicazione.
- b) La classe **MinesSweeperGrid** (fornita) contiene tutta la griglia completa di algoritmo di gioco
 - il costruttore riceve il controller e la textarea (definita nel pane, v. oltre) su cui scrivere
 - il metodo **updateGridStatus** aggiorna la visualizzazione della griglia in base alla stringa ricevuta come argomento (che si suppone nel formato restituito dalla **toString** di **MineField**)
 - il metodo **endGame** termina la partita, disabilitando tutti i pulsanti della griglia e richiamando poi i metodi **save** e **close** del controller per salvare lo stato finale della partita.
- c) La classe **MinesSweeperPane** (da realizzare) è una specializzazione di **BorderPane**:
 - in alto, una label indica al giocatore il numero di mine
 - al centro, una **MinesSweeperGrid** costituisce il pannello di gioco
 - in basso, due pulsanti “Save status” e “Restart”, agganciati a due distinti metodi di gestione, consentono rispettivamente di salvare lo stato attuale del gioco e di resettarlo per fare una nuova partita.

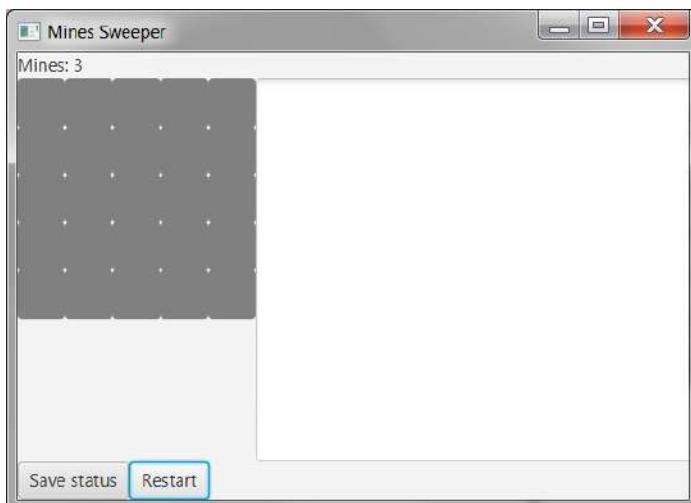


Figura 1

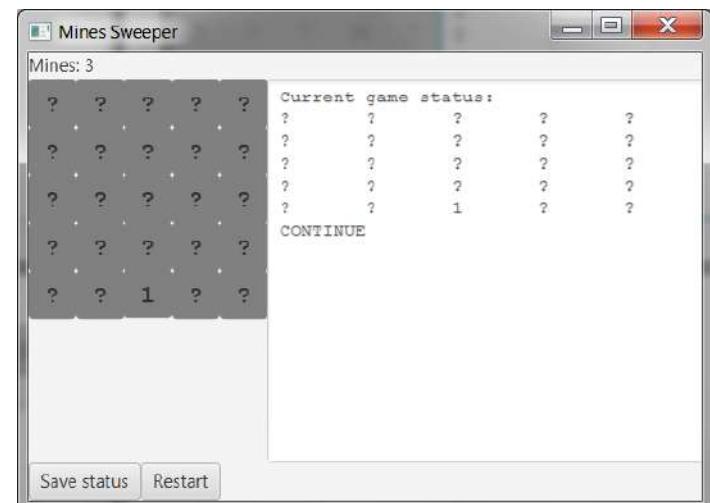


Figura 2

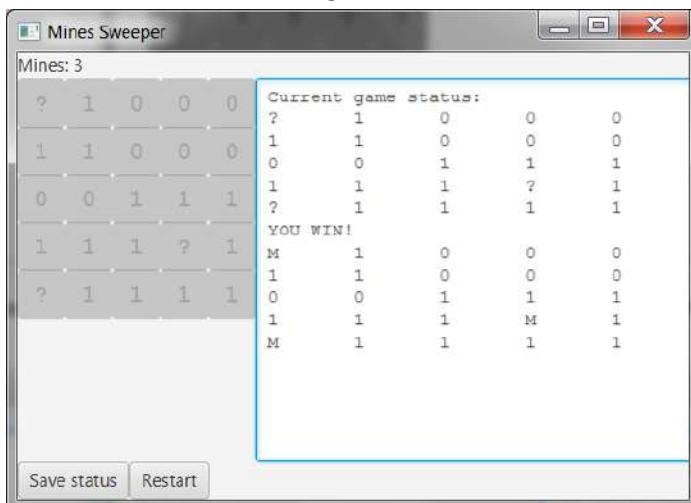


Figura 3

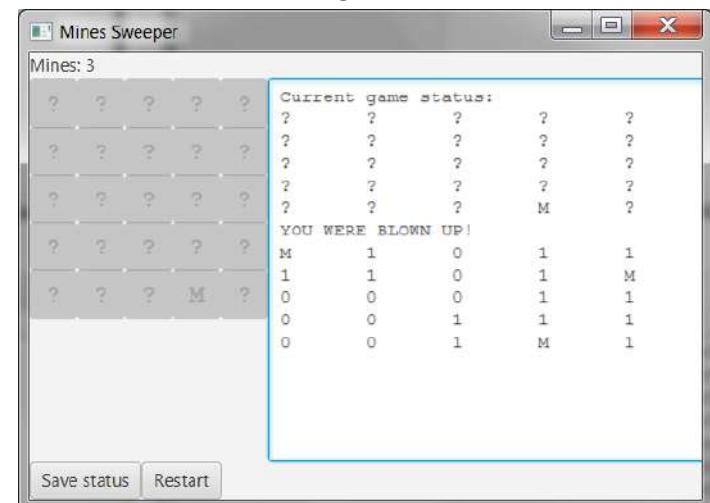


Figura 4

Cose da ricordare

- salva costantemente il tuo lavoro, facendo ZIP parziali e consegne parziali (vale l'ultima)
- in particolare, se ora compila e stai per fare modifiche, salva la versione attuale (non si sa mai..)

Checklist di consegna

- Hai fatto un unico file ZIP (**non .7z!!!**) contenente l'intero progetto?
In particolare, ti sei assicurato di aver incluso tutti i file .java (e non solo i .class)?
- Hai controllato che si compili e ci sia tutto ? [NB: non serve includere il PDF del testo]
- Hai rinominato IL PROGETTO esattamente come richiesto?
- Hai chiamato IL FILE ZIP esattamente come richiesto?
- Hai chiamato la cartella del progetto esattamente come richiesto?
- Dopo aver caricato il file su Esamix, hai premuto il tasto “CONFERMA”, ottenendo il messaggio “Hai concluso l’esame”?

ESAME DI FONDAMENTI DI INFORMATICA T-2 del 23/7/2019

Proff. E. Denti, R. Calegari, A. Molesini – Tempo: 4 ore

NOME PROGETTO ECLIPSE: CognomeNome-matricola (es. RossiMario-0000123456)

NOME CARTELLA PROGETTO: CognomeNome-matricola (es. RossiMario-0000123456)

NOME ZIP DA CONSEGNARE: CognomeNome-matricola.zip (es. RossiMario-0000123456.zip)

NB: l'archivio ZIP da consegnare deve contenere l'intero progetto Eclipse

Si ricorda che compiti non compilabili o palesemente lontani da 18/30 NON SARANNO CORRETTI e causeranno la verbalizzazione del giudizio “RESPINTO”

È stato richiesto lo sviluppo di un'applicazione per la ricerca di percorsi di mezzi pubblici (bus, metro..) in una città.

DESCRIZIONE DEL DOMINIO DEL PROBLEMA

Ogni linea del trasporto pubblico, operata con i mezzi più diversi (bus, tram, metro, traghetti..), è identificata univocamente da un *codice identificativo* (alfanumerico) e si intende operativa *in un'unica direzione, dal capolinea iniziale al capolinea finale*: l'eventuale linea operante nel senso inverso avrà un codice identificativo differente.

A ogni linea sono associate le corrispondenti *fermate*, caratterizzate ognuna dal relativo *orario di passaggio* della corsa: questo è inteso come *differenza in minuti rispetto alla partenza dal capolinea iniziale*, che ha orario di passaggio 0.

Per semplicità non si considerano tempi di percorrenza diversi fra ore di punta e ore di morbida, né fra i diversi giorni della settimana.

Le linee si distinguono in:

- Linee da punto a punto (PaP), che vanno dal capolinea iniziale a un capolinea finale *diverso* dal primo
- Linee circolari, che partono dal capolinea iniziale e, dopo un giro più o meno lungo, vi ritornano: per queste linee, quindi, i capolinea iniziale e finale coincidono sempre.

Un *percorso* dalla fermata X alla fermata Y è un *tragitto senza cambi* che collega X a Y:

- per le linee da punto a punto (PaP), ciò significa semplicemente un tragitto da X verso Y (con $X \neq Y$)
- per le linee circolari, il tragitto può invece anche “scavallare” il capolinea, quindi Y può anche precedere X: in tal caso, il percorso si intende da X fino al capolinea e poi, proseguendo, da lì fino a Y.

La durata del percorso è definita rispettivamente come:

- per le linee da punto a punto (PaP), la differenza (>0) tra gli orari di passaggio alle due fermate Y e X
- per le linee circolari, a seconda della posizione relativa delle due fermate rispetto al capolinea, o come sopra, o (se la differenza è negativa) come somma dei due sotto-tragitti da X al capolinea e dal capolinea a Y.

ESEMPI: nel caso della linea circolare 33 di Bologna, il percorso da Porta Saragozza a Petrarca è svolto nel senso diretto e dura solo 3 minuti, mentre il percorso inverso, da Petrarca a Porta Saragozza, viene coperto dalla stessa linea in ben 35 minuti, facendo tutto il giro dalla stazione (non viene trovato alcun percorso con la linea 32, in quanto essa non ha alcuna fermata di nome “Petrarca”). Un percorso come Porta San Mamolo- stazione viene trovato invece sia con la linea 33 sia con la linea 32 (oltre che con l’immaginaria metropolitana M1 ☺), con tempi di percorrenza diversi – che chiaramente si scambiano effettuando il percorso nel senso inverso.

Al contrario, per le linee monodirezionali da punto a punto, come il “20 direzione Casalecchio”, il percorso da Via Marconi a Porta Saragozza è possibile, mentre quello opposto non lo è (è supportato da un'altra linea, il “20 direzione Pilastro”, che può avere – e infatti ha – fermate diverse e instradamento diverso).

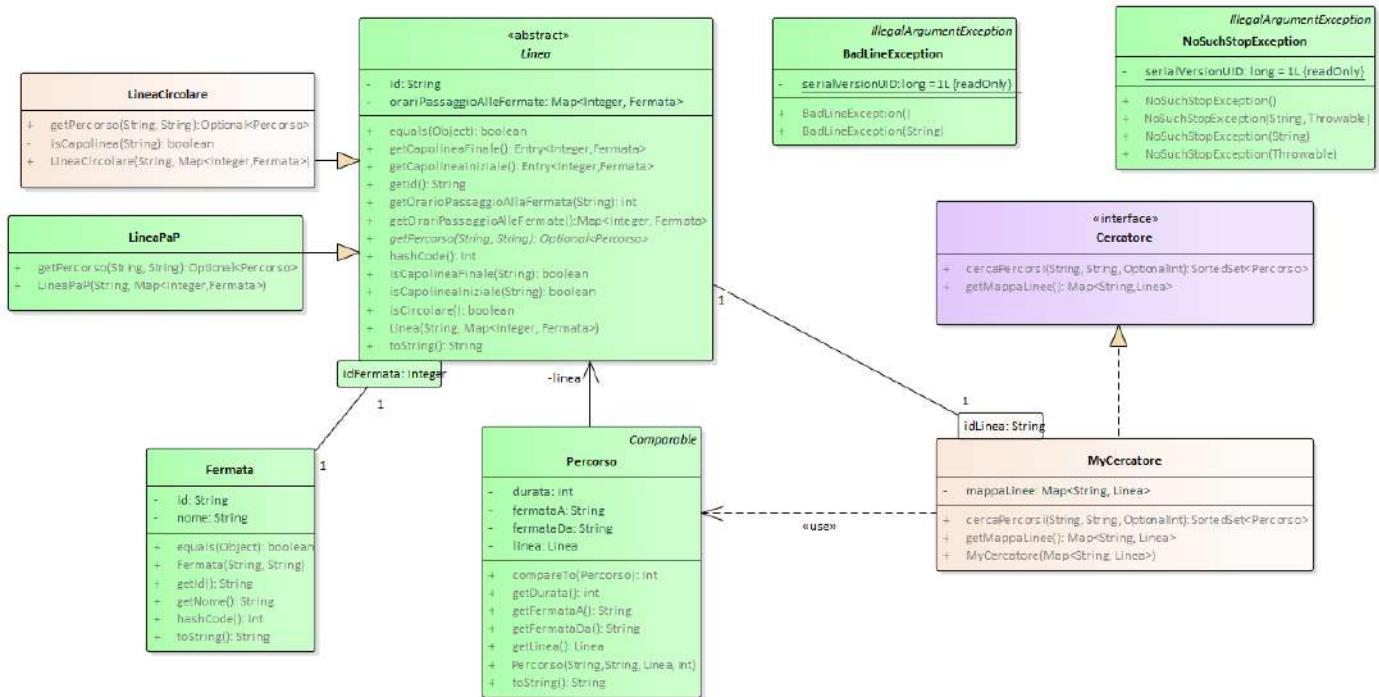
Parte 1

(punti: 23)

Modello dei dati (package bussy.model)

(punti: 15)

Il modello dei dati deve essere organizzato secondo il diagramma UML sotto riportato.



SEMANTICA:

- a) La classe **Fermata** (fornita) rappresenta una fermata su una linea di trasporto pubblico, caratterizzata da identificativo univoco e denominazione; sono ovviamente presenti i relativi metodi accessor e di equality
 - b) La classe **Percorso** (fornita) rappresenta un percorso fra due fermate, caratterizzato da una certa durata (in minuti): è già **Comparable** per durata crescente.
 - c) La classe astratta **Linea** (fornita) rappresenta una generica linea di trasporto con le sue proprietà, recuperabili tramite i metodi accessor. Il costruttore riceve l'identificativo della linea e una **mappa <Integer, Fermata>** che costituisce l'elenco delle fermate, indicizzate in base al minuto di passaggio: il capolinea iniziale ha ovviamente indice 0. I metodi fondamentali sono:
 - **getOrariPassaggioAlleFermata** restituisce la mappa ricevuta dal costruttore
 - **getOrarioPassaggioAllaFermata (String nomeFermata)** restituisce l'intero corrispondente all'orario di passaggio della corsa a quella fermata, o **NoSuchStopException (fornita)** se tale fermata non è presente nella linea;
 - **getCapolineaIniziale** e **getCapolineaFinale** restituiscono la riga della mappa corrispondente rispettivamente alla prima e all'ultima fermata della linea;
 - **isCapolineaIniziale (String nomeFermata)**, **isCapolineaFinale (String nomeFermata)** e **isCircolare** restituiscono *true* se la condizione implicita nel loro nome è verificata
 - **getPercorso (String nomeFermataDa, String nomeFermataA) astratto** restituisce il percorso, se esiste, fra tali due fermate, nella direzione da **nomeFermataDa** a **nomeFermataA**, o un optional vuoto in caso contrario.
 - d) La classe **LineaPAP** (fornita) rappresenta una linea da punto a punto: il costruttore delega la costruzione alla classe base, ma verifica anche che la linea non sia circolare – altrimenti, lancia **BadLineException (fornita)** con opportuno messaggio. Il metodo **getPercorso** è qui concretizzato nel caso semplice di linea punto a punto

monodirezionale: nel caso una delle due fermate non appartenga alla linea o la seconda preceda la prima, viene restituito un **Optional** vuoto, senza lanciare alcuna eccezione.

- e) La classe **LineaCircolare** (da realizzare) rappresenta una linea circolare: il costruttore è analogo a quello del caso precedente, salvo ovviamente la verifica di circolarità, che dev'essere invertita. Il metodo `getPercorso` deve invece considerare anche i percorsi circolari via capolinea, distinguendo quindi due situazioni base:

- **da fermataDa a fermataA, con fermataDa < fermataA (caso standard come PaP)**
- **da fermataDa a fermataA, con fermataDa >= fermataA (da interpretare via capolinea)**

Esse diventano in realtà cinque considerando i casi limite che coinvolgono il capolinea, che è opportuno trattare separatamente per meglio gestire il calcolo dei tempi di percorrenza:

- **da capolinea a fermataA (diversa dal capolinea)**
- **da fermataDa (diversa dal capolinea) al capolinea**
- **da capolinea a capolinea (giro completo)**
- **da fermataDa a fermataA, con fermataDa < fermataA (caso standard come PaP)**
- **da fermataDa a fermataA, con fermataDa >= fermataA (da interpretare via capolinea)**

Come sopra, se una delle due fermate non appartiene alla linea viene restituito un **Optional** vuoto, senza lanciare alcuna eccezione

- f) L'interfaccia **Cercatore** (fornita) rappresenta la vista esterna di un'entità capace di cercare percorsi da una fermata A a una fermata B: il metodo `cercaPercorsi` riceve come argomenti i nomi delle due fermate e un intero optional che rappresenta *la durata massima accettabile del percorso*: eventuali percorsi esistenti ma più lunghi saranno perciò esclusi dal risultato. Il risultato è un `SortedSet<Percorsi>` ordinati dal più breve al più lungo. Il metodo ausiliario `getMappaLinee` restituisce la mappa `Map<String, Linea>` di tutte le linee del trasporto pubblico.
- g) La classe **MyCercatore** (da realizzare) concretizza tale astrazione: il costruttore riceve la mappa `Map<String, Linea>` di tutte le linee del trasporto pubblico (che dev'essere ovviamente non nulla e non vuota). Anche il metodo `cercaPercorsi` deve ovviamente verificare che i nomi delle due fermate non siano nulli, lanciando **IllegalArgumentException** in caso contrario.

Persistenza (package bussy.persistence)

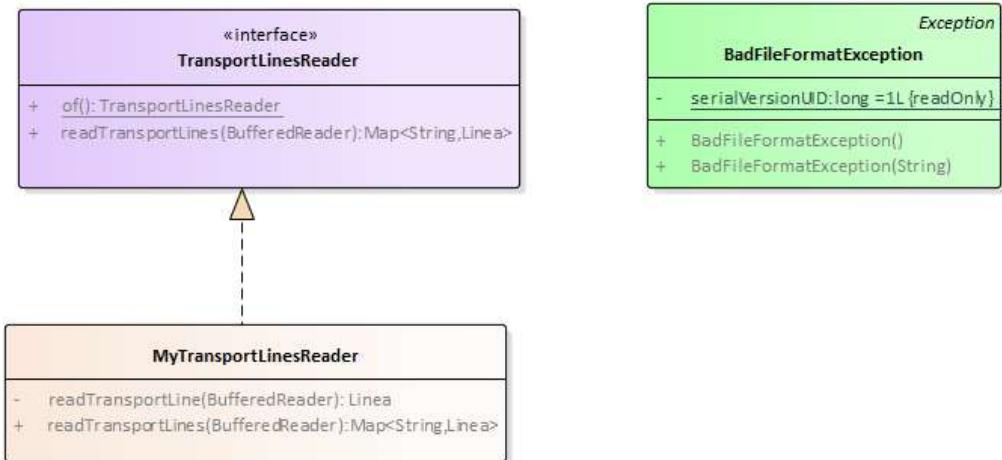
(punti 8)

Il file “`Linee.txt`” contiene la descrizione di tutte le linee del trasporto pubblico, suddivise in blocchi (v. esempio), che si susseguono uno dopo l'altro, in sequenza, senza un ordine relativo specifico e senza che il loro numero sia noto a priori.

Ogni blocco inizia con la dichiarazione **Linea ID**, dove **ID** è un identificativo alfanumerico senza spazi: seguono tante righe quante le fermate. Ogni riga comprende l'orario di passaggio del mezzo (minuti rispetto alla partenza dal capolinea), l'identificativo della fermata e la denominazione della stessa (che può consistere di più parole e contenere ogni carattere tranne le virgolette), separati da virgolette (più eventuali spazi o tabulazioni, non indispensabili). Infine, il blocco è concluso da una riga che inizia con *almeno due lineette*.

```
Linea 32
0, 40, Porta San Mamolo
3, 42, Aldini
5, 44, Porta Saragozza - Villa Cassarini
17, 16, Stazione Centrale
38, 40, Porta San Mamolo
-----
```

La struttura di questa parte dell'applicazione è illustrata nel diagramma UML sotto riportato.



SEMANTICA:

- a) L'interfaccia **TransportLinesReader** dichiara il metodo `readTransportLines` che legge – da un **BufferedReader** ricevuto come argomento – i dati di tutte le linee del trasporto pubblico, restituendo la `Map<String, Linea>` di tutte le linee, indicizzata per identificativo univoco di linea: lancia **IOException** o **BadFormatException** rispettivamente nel caso si verifichino errori di IO o il formato del file differisca da quello atteso. Il metodo statico factory `of` costruisce l'istanza della classe concreta **MyTransportLinesReader**.
- b) La classe **MyTransportLinesReader** (da realizzare) implementa tale interfaccia, fornendo specifici messaggi d'errore nelle eccezioni lanciate, così da distinguere le sorgenti di errore.
Si suggerisce di articolare l'implementazione di `readTransportLines` appoggiandosi a un metodo privato ausiliario che legga un singolo blocco (es. `readTransportLine`), restituendo quindi una singola **Linea** (o null quando il file termina, così da mantenere la semantica classica dei cicli di lettura).

IMPORTANTE: per consentire il test della GUI anche nel caso di *reader* non funzionanti, è fornita la classe **BussyAppMock** che replica **BussyApp** utilizzando dati fissi pre-cablati al posto di quelli letti da file.

L'applicazione deve permettere di scegliere la fermata di partenza e di destinazione fra quelle servite dalle linee di trasporto, cercando e mostrando poi i percorsi disponibili fra esse.

Sebbene il modello dei dati supporti la specifica di una durata massima di viaggio, in questa versione della GUI si sceglie di non offrire tale opzione: il tempo massimo di viaggio sarà quindi convenzionalmente fissato a un'ora.

In alto, due combo di stringhe sono popolate con i nomi di tutte le fermate, in ordine alfabetico (Figg. 1,2,3): inizialmente non è selezionata alcuna fermata. Avvicinandosi alla combo, appositi tooltip (non mostrati) invitano a scegliere una fermata. Dopo aver scelto le due fermate (Figg. 4 e seguenti), premendo il pulsante **Cerca percorso** si ottengono a video i risultati (Figg. 5-10) o l'informazione che non ne sono stati trovati (Fig. 11).

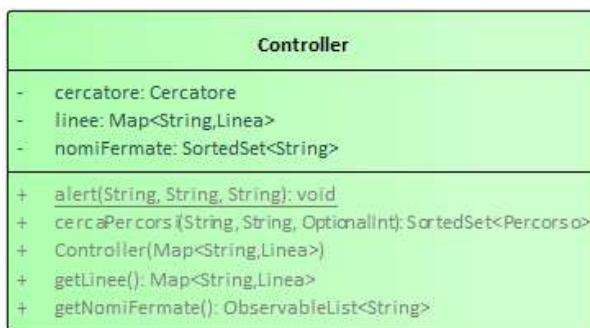
Naturalmente, in caso di linee circolari vengono offerti anche percorsi via capolinea (Figg. 5, 6, 7, 8), purché le relative fermate esistano: ad esempio, poiché la fermata “Petrarca” è presente solo sulla linea 33 ma non sull’omologa 32, cercando un percorso che la coinvolga viene trovato un solo percorso, sia in un verso che nell’altro (Figg. 9, 10). Analogamente, se una linea da punto a punto è disponibile in un’unica direzione (come la metropolitana di fantasia M1), il relativo percorso viene incluso solo in quella direzione e non nell’altra (Figg. 8, 9, 10).

Controller (package bussy.ui.controller)

(punti 0)

Il **Controller** è fornito già implementato:

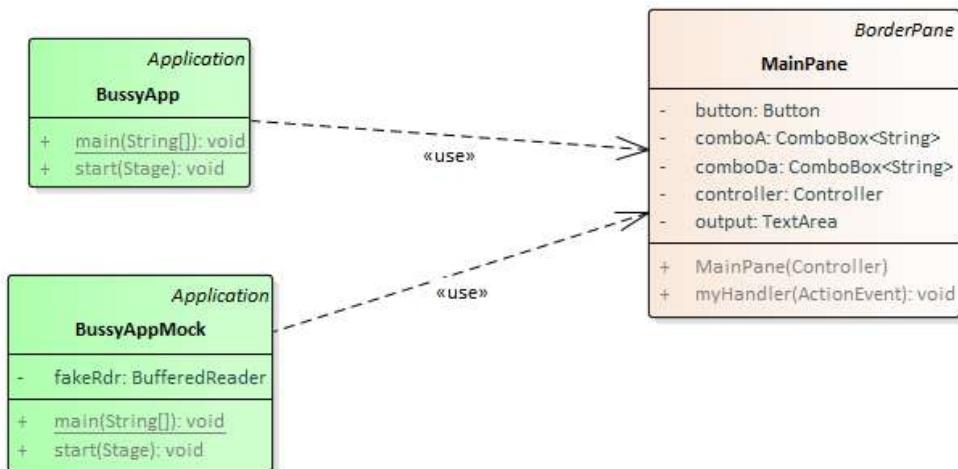
- il costruttore riceve la mappa delle linee del trasporto pubblico, riottenibile dall'apposito accessor `getLinee`
- il metodo `getNomiFermate` restituisce la lista osservabile di tutte le fermate del trasporto pubblico, già ordinata alfabeticamente
- il metodo `cercaPercorsi` è un puro front-end verso l'omonimo metodi di **Cercatore**.



Interfaccia utente (package bussy.ui)

(punti 7)

L'interfaccia utente deve essere simile (non necessariamente identica) all'esempio mostrato di seguito: la struttura generale è quella di un *pannello a bordi*, in cui la parte top contiene le combo, la parte inferiore il bottone, e quella centrale l'area di testo. Se il caricamento preliminare (realizzato da **BussyApp** fornita nello start kit) ha esito positivo, compare la finestra principale dell'applicazione, con le combo pre-popolate con i nomi delle fermate come sopra spiegato; diversamente, viene mostrata una finestra di errore tramite il metodo statico **Controller.alert**.



SEMANTICA

- La classe **BussyApp** (fornita) contiene il main dell'applicazione, il cui metodo `start` legge il file di testo, istanzia il controller e il **MainPane** e infine attiva la scena
- La classe **MainPane** (da realizzare) estende **BorderPane** implementando il pannello principale come sopra spiegato. Solo il bottone scatena eventi: eventuali selezioni nelle combo non devono avere effetto finché non si ri-preme il pulsante. In caso non vi siano percorsi, *non* si deve lanciare **alert**, ma soltanto mostrare l'esito nella finestra dell'applicazione (Fig. 10).

Cose da ricordare

- salva costantemente il tuo lavoro, facendo ZIP parziali e consegne parziali (vale l'ultima)
- in particolare, se ora compila e stai per fare modifiche, salva la versione attuale (non si sa mai..)

Checklist di consegna

- Hai fatto un unico file ZIP (**non .7z!!!**) contenente l'intero progetto?
In particolare, ti sei assicurato di aver incluso tutti i file .java (e non solo i .class)?
- Hai controllato che si compili e ci sia tutto ? [NB: non serve includere il PDF del testo]
- Hai rinominato IL PROGETTO esattamente come richiesto?
- Hai chiamato IL FILE ZIP esattamente come richiesto?
- Hai chiamato la cartella del progetto esattamente come richiesto?
- Dopo aver caricato il file su Esamix, hai premuto il tasto “CONFERMA”, ottenendo il messaggio “Hai concluso l'esame”?

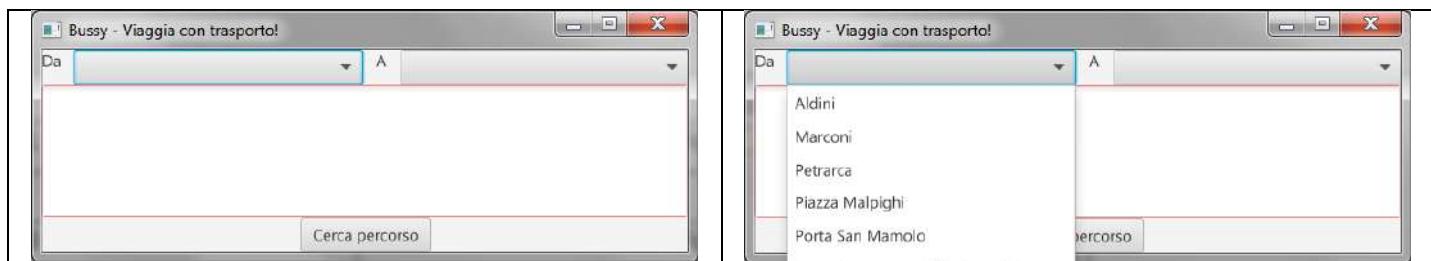


Fig. 1

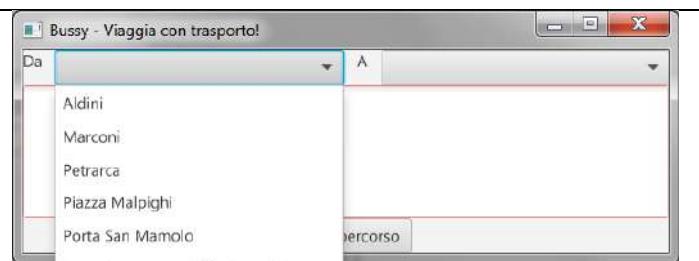


Fig. 2

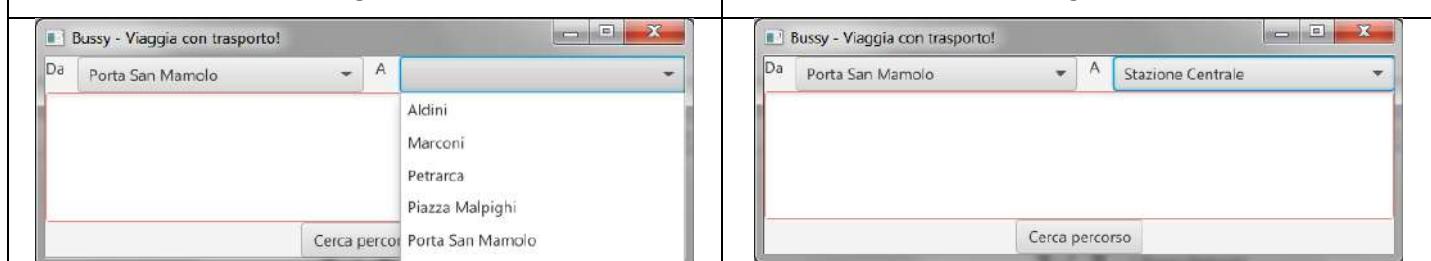


Fig. 3

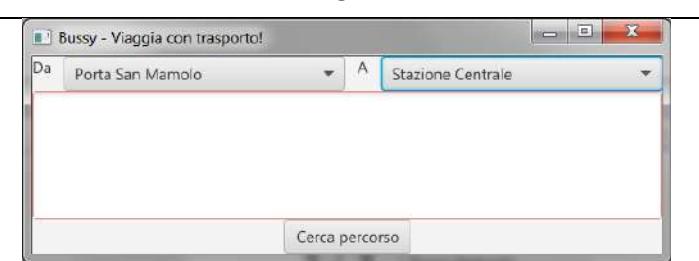


Fig. 4

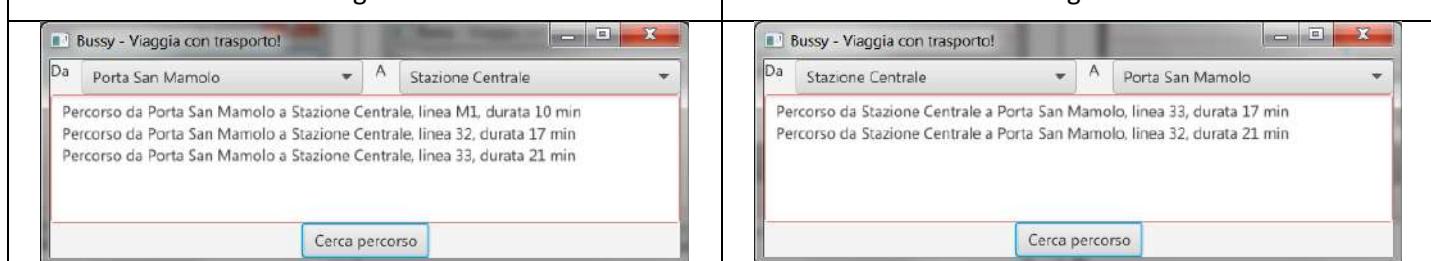


Fig. 5

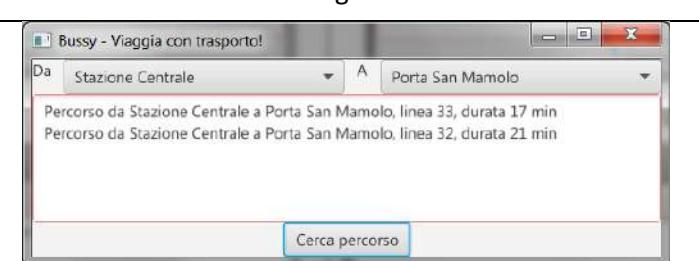


Fig. 6

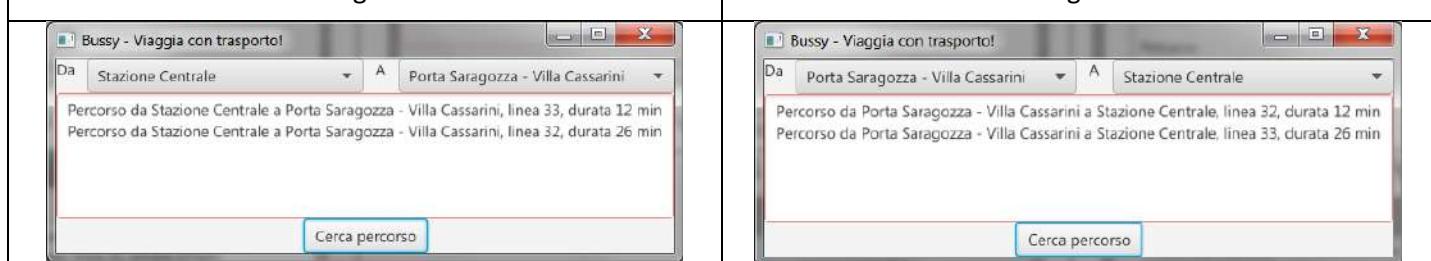


Fig. 7

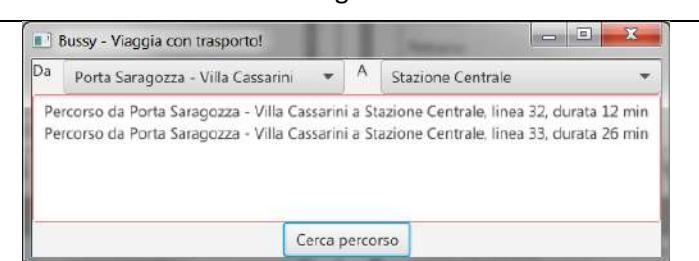


Fig. 8

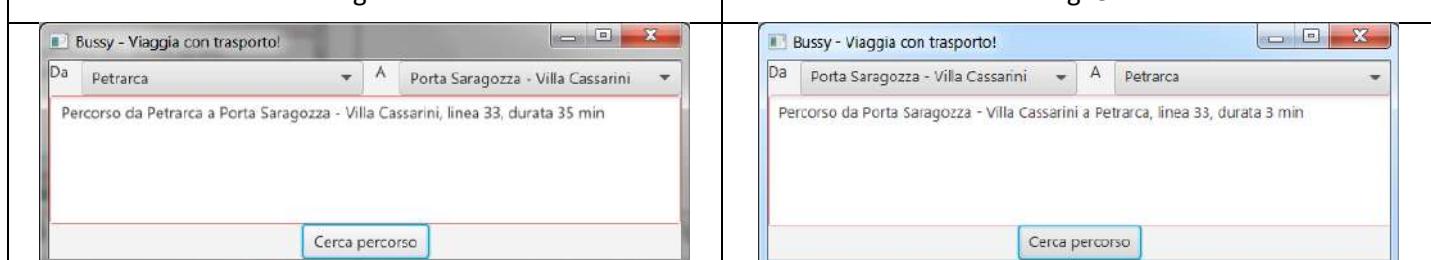


Fig. 9

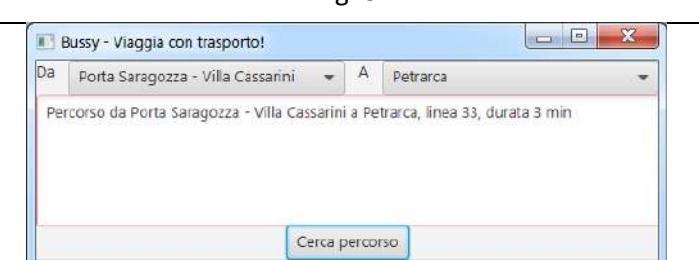


Fig. 10

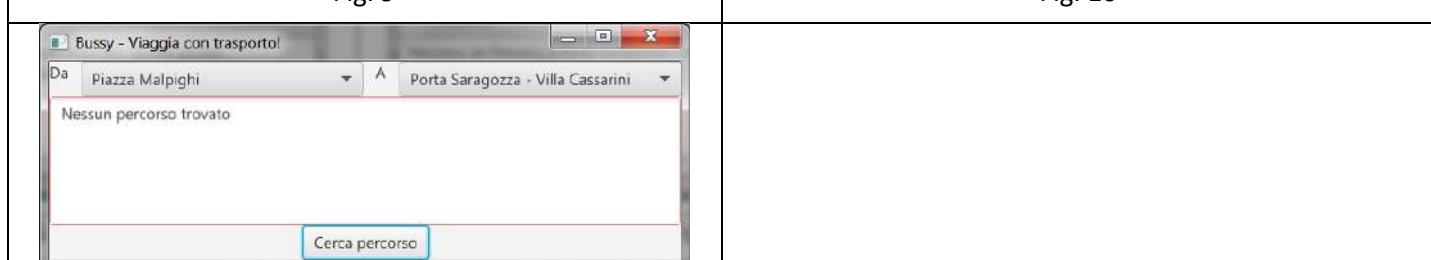


Fig. 11

ESAME DI FONDAMENTI DI INFORMATICA T-2 dell' 8/7/2019

Proff. E. Denti, R. Calegari, A. Molesini – Tempo: 4 ore

NOME PROGETTO ECLIPSE: CognomeNome-matricola (es. RossiMario-0000123456)

NOME CARTELLA PROGETTO: CognomeNome-matricola (es. RossiMario-0000123456)

NOME ZIP DA CONSEGNARE: CognomeNome-matricola.zip (es. RossiMario-0000123456.zip)

NB: l'archivio ZIP da consegnare deve contenere l'intero progetto Eclipse

Si ricorda che compiti non compilabili o palesemente lontani da 18/30 NON SARANNO CORRETTI e causeranno la verbalizzazione del giudizio “RESPINTO”

Poiché oggi giorno è possibile scaricare da appositi siti web un file che descrive l'intero percorso di un volo aereo in una certa data, è stato richiesto lo sviluppo di un'applicazione grafica in grado di mostrare il percorso di tale volo su un (semi-)planisfero.

DESCRIZIONE DEL DOMINIO DEL PROBLEMA

Appositi siti (es. FlightRadar24) consentono di seguire in tempo reale la posizione dei voli aerei, nonché di scaricare un file contenente la traccia completa di un volo, a scelta fra tutti quelli conclusi negli ultimi due o tre giorni. Il file descrive il volo dal momento in cui inizia a muoversi dal gate dell'aeroporto di partenza a quello in cui parcheggia al gate dell'aeroporto di arrivo: pertanto, è composto tipicamente da alcune centinaia di rilevazioni, disposte su altrettante righe, nel formato dettagliato più oltre. Ogni rilevazione è caratterizzata da:

- il timestamp, in formato UTC
- la posizione (latitudine e longitudine), in gradi, con 6 cifre decimali
- l'altitudine rispetto al terreno, in piedi (valore intero)
- la velocità, in nodi nautici (valore intero)
- la direzione, in gradi (nord = 0°) (valore intero)

La prima e l'ultima rilevazione hanno velocità e altitudine 0, a conferma che l'aereo è fermo a terra al gate.

Cose da ricordare

- salva costantemente il tuo lavoro, facendo ZIP parziali e consegne parziali (vale l'ultima)
- in particolare, se ora compila e stai per fare modifiche, salva la versione attuale (non si sa mai..)

Checklist di consegna

- Hai fatto un unico file ZIP (**non .7z!!!**) contenente l'intero progetto?
In particolare, ti sei assicurato di aver incluso tutti i file .java (e non solo i .class)?
- Hai controllato che si compili e ci sia tutto ? [NB: non serve includere il PDF del testo]
- Hai rinominato IL PROGETTO esattamente come richiesto?
- Hai chiamato IL FILE ZIP esattamente come richiesto?
- Hai chiamato la CARTELLA del progetto esattamente come richiesto?
- Dopo aver caricato il file su Esamix, hai premuto il tasto “CONFERMA”, ottenendo il messaggio “Hai concluso l'esame”?

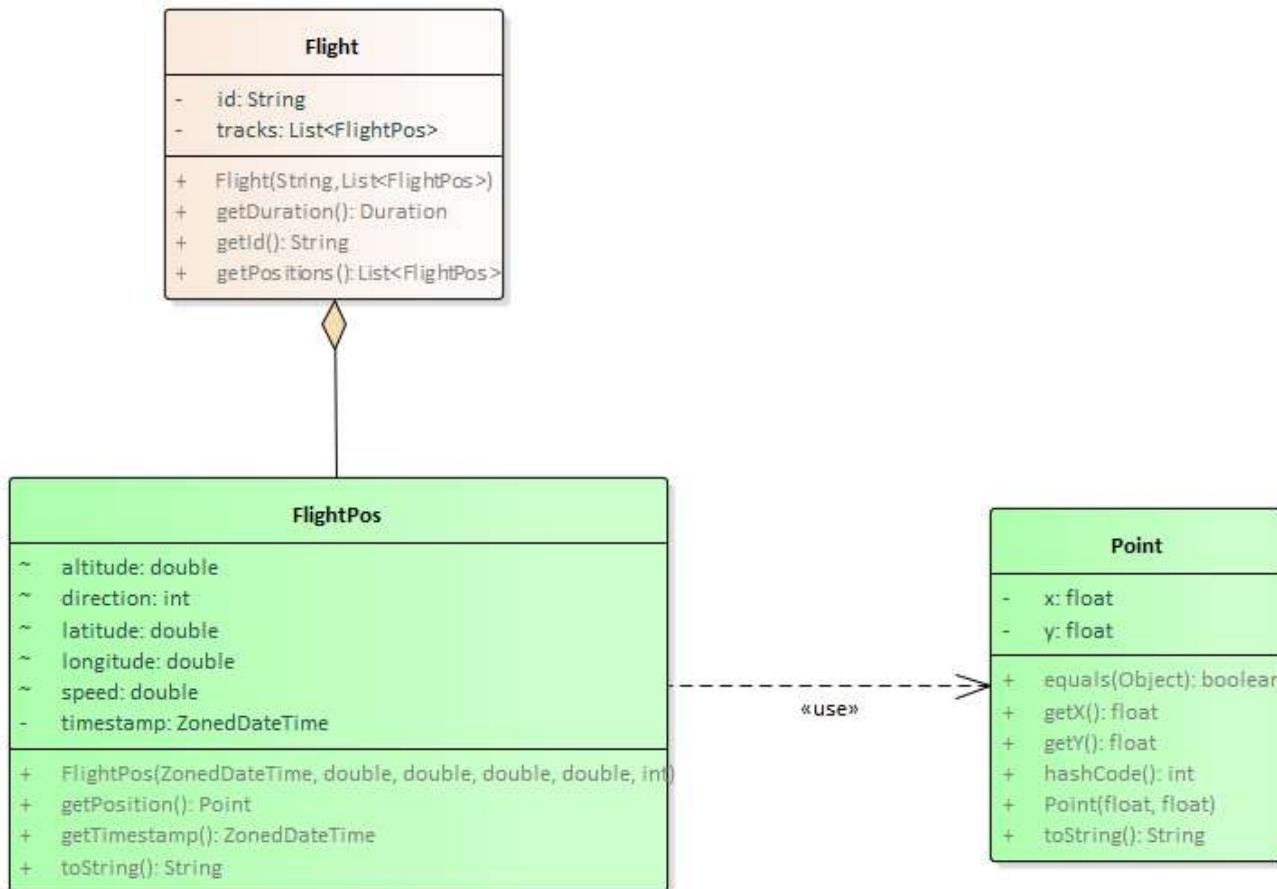
Parte 1

(punti: 16)

Modello dei dati (package `flightTracker.model`)

(punti: 5)

Il modello dei dati deve essere organizzato secondo il diagramma UML sotto riportato.



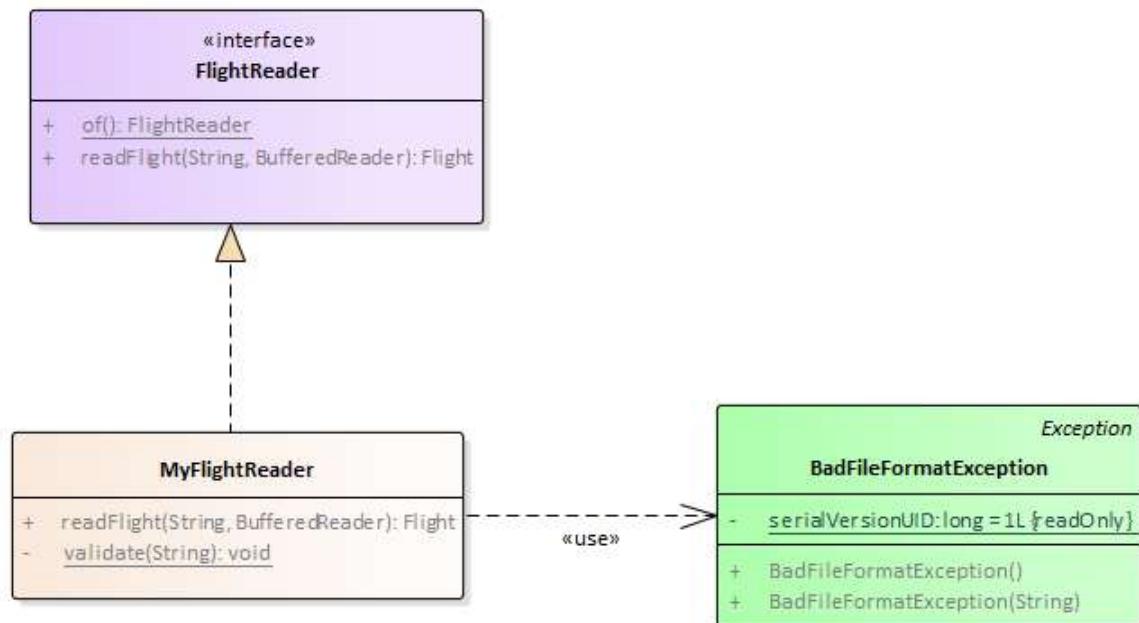
SEMANTICA:

- La classe **Point** (fornita) rappresenta una posizione (latitudine, longitudine), con relativi metodi accessor
- La classe **FlightPos** (fornita) rappresenta una rilevazione, caratterizzata dalle sei proprietà descritte nel *Dominio del Problema*, rese accessibili de appositi accessor
- La classe **Flight** (**da realizzare**) rappresenta un volo, caratterizzato dal suo identificativo univoco e dalla lista delle rilevazioni (**FlightPos**): il costruttore deve effettuare opportuni controlli sulla validità degli argomenti ricevuti, lanciando **IllegalArgumentException** in caso contrario, con idonea messaggistica; la classe deve esporre ovvi metodi accessor, nonché il metodo **getDuration** che calcoli la durata del volo.

Sono forniti svariati file “.csv” nel formato fornito da FlightRadar24: il nome del file incapsula l’identificativo del volo e la data di effettuazione, nel formato *FlightID_AAAAMMGG* (ad esempio “[AZ604_20190510.csv](#)”). La prima riga contiene l’intestazione delle cinque colonne, separate da punti e virgola, mentre le successive contengono ciascuna un rilevazione, i cui campi sono anch’essi separati da punti e virgola. Da notare però che latitudine e longitudine costituiscono un unico campo, separato al suo interno da una virgola:

```
UTC;Position;Altitude;Speed;Direction
2019-05-10T10:54:39Z;45.661972,8.726303;1975;183;356
2019-05-10T10:54:49Z;45.67049,8.725822;2450;182;358
2019-05-10T10:55:11Z;45.689159,8.726234;3100;185;1
...
```

La struttura di questa parte dell’applicazione è illustrata nel diagramma UML sotto riportato.



SEMANTICA:

- L’interfaccia **FlightReader** dichiara il metodo `readFlight` che legge – da un **BufferedReader** ricevuto come argomento – i dati di un volo e li incapsula in un’opportuna istanza di **Flight**, lanciando le eccezioni **IOException** o **BadFileNotFoundException** rispettivamente nel caso si verifichino errori di IO o il formato del file differisca da quello atteso. È anche presente un metodo factory `of` che istanzia un **MyFlightReader**.
- La classe **MyFlightReader** (**da realizzare**) implementa tale interfaccia nel caso specifico del formato dei file .csv di questo caso concreto, fornendo specifici messaggi d’errore nelle eccezioni lanciate, così da distinguere le sorgenti di errore. In particolare occorre validare la riga d’intestazione, che deve contenere esattamente le cinque parole previste (“UTC”, “Position”, “Altitude”, “Speed”, “Direction”) nel giusto ordine e formato.

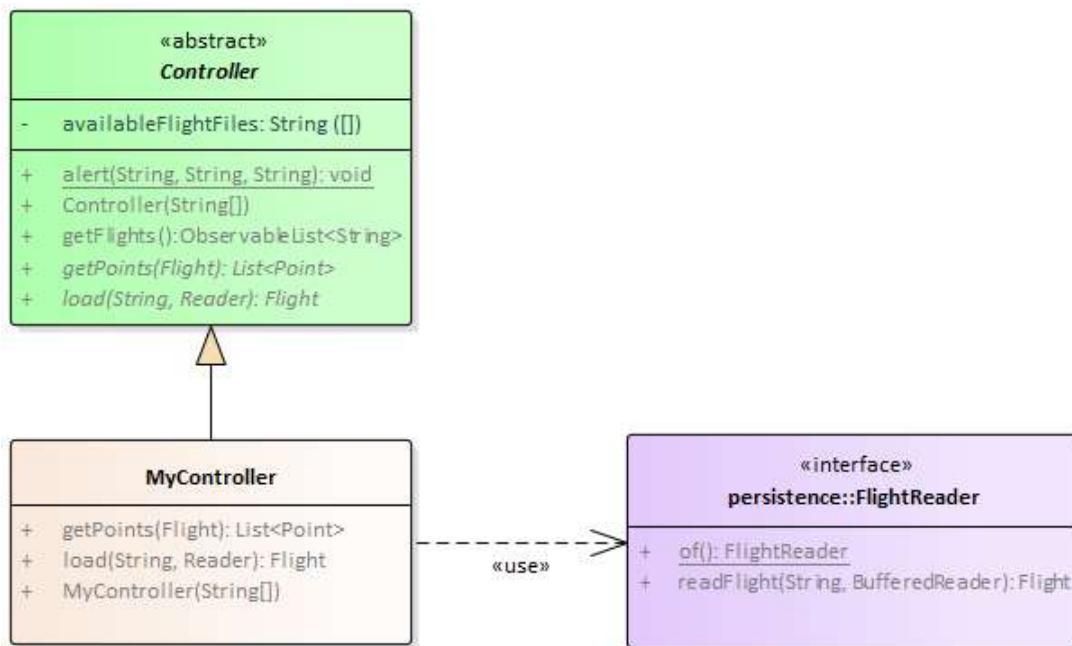
IMPORTANTE: per consentire il test della GUI anche nel caso di *reader* non funzionanti, è fornita la classe **FlightTrackerAppMock** che replica **FlightTrackerApp** utilizzando dati fissi pre-cablati al posto di quelli letti da file (utilizzando a tal fine un apposito **ControllerMock**).

L'applicazione deve permettere di scegliere il volo desiderato fra quelli disponibili e vederne il grafico sull'emisfero.

Con riferimento alle figure seguenti: la combo in alto a sinistra contiene i nomi dei file disponibili (Fig. 1). Avvicinandosi alla combo, compare un tooltip che invita a scegliere un volo (Fig. 2). Scegliendolo (Fig. 3), viene mostrato il relativo grafico. Se successivamente l'utente sceglie un altro volo, di default viene mostrato solo il nuovo volo (Fig. 4). Tuttavia, è possibile selezionare la checkbox a lato per abilitare la selezione multipla, nel qual caso scegliendo altri voli le relative tracce vengono aggiunte al grafico esistente, in colori casuali via via differenti (Fig. 5).

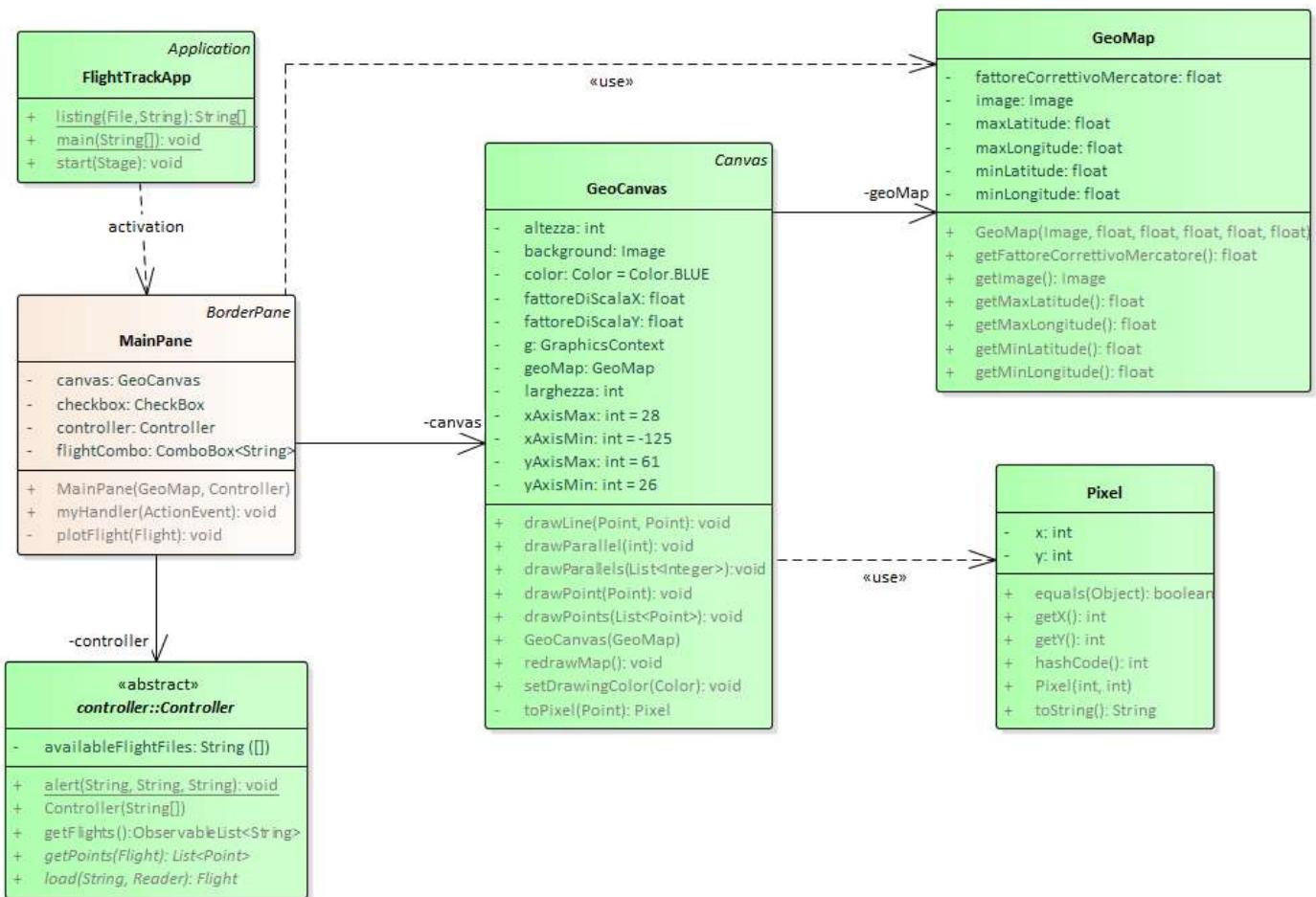
Controller (package `flightTracker.ui.controller`)

(punti 5)



SEMANTICA

- La classe astratta **Controller** (fornita) fornisce un'implementazione parziale del controller dell'applicazione:
 - Il costruttore riceve una **GeoMap** e la lista dei nomi dei file .csv disponibili: effettua opportuni controlli sulla validità degli argomenti ricevuti, lanciando **IllegalArgumentException** in caso contrario;
 - Il metodo **getFlights** restituisce la lista osservabile dei nomi di file corrispondenti ai voli disponibili;
 - Il metodo statico **alert** permette di far comparire un dialogo di avviso per segnalare errori (eccezioni).
- La classe **MyController** (da realizzare) estende **Controller** implementando
 - il metodo **getPoints** che restituisce la lista di **Point** corrispondenti al volo passato come argomento: in pratica, dato un volo, recupera le singole posizioni (latitudine e longitudine) delle varie rilevazioni, le converte ciascuna nel **Point** corrispondente e restituisce la lista così ottenuta (pronta per il metodo **plotFlight** del **MainPane**)
 - Il metodo **load** che, dato un **Reader**, legge – tramite **FlightReader** – il file con la descrizione del volo: se qualcosa va storto (errore di I/O o errori di formato), l'eccezione va semplicemente rilanciata all'esterno, in modo da essere gestita nella GUI.



L'applicazione deve permettere di scegliere il volo desiderato fra quelli disponibili e vederne il grafico sull'emisfero.

La struttura generale è quella di un *pannello a bordi*, di cui sono utilizzati solo la parte top per le barre controlli e il centro per l'emisfero con la traccia di volo. Se il caricamento preliminare (realizzato nella ***FlightTrackApp*** fornita nello start kit) ha esito positivo, deve comparire subito la finestra principale dell'applicazione, con la combo prepopolata con tutti i nomi dei file .csv presenti nella cartella corrente (Fig. 1). Avvicinandosi alla combo, un tooltip suggerisce all'utente cosa fare (Fig. 2). Quando l'utente sceglie un volo, l'applicazione visualizza immediatamente la traccia corrispondente (Fig. 3): cambiando volo, la visualizzazione si aggiorna (Fig. 4) mostrando quest'ultimo.

Se tuttavia viene attivata la checkbox “Voli multipli”, cambiando volo i nuovi si aggiungono, con colori diversi, alla mappa preesistente, che quindi mostra tutte le tracce insieme (Fig. 5).

SEMANTICA

- La classe ***FlightTrackApp*** (fornita) contiene il main dell'applicazione, il cui metodo ***start*** costruisce la ***GeoMap***, produce la lista dei nomi dei file .csv disponibili, istanzia il controller e il ***MainPane***, e infine attiva la scena
- La classe ***Pixel*** (fornita) rappresenta una posizione grafica con coordinate intere
- La classe ***GeoMap*** (fornita) rappresenta una mappa geografica, caratterizzata da un'immagine e dalla corrispondente estensione orizzontale (da longitudine minima a longitudine massima) e verticale (da latitudine minima a latitudine massima); la descrizione è completata da un fattore di scala (fattore correttivo Mercatore) che tiene conto, in modo approssimato, della non linearità delle mappe nella proiezione di Mercatore comunemente utilizzata. Tale valore (1 per mappe che non necessitano di correzione) è normalmente molto prossimo a 1 per mappe che inquadrono solo una piccola porzione di territorio e/o si estendono poco in senso

verticale, mentre dev'essere proporzionalmente ridotto per mappe che abbracciano un'ampia fetta dell'emisfero. Per la mappa fornita nello start kit, un valore ragionevole è 0.87-0.88, impostato nel main.

d) La classe ***GeoCanvas*** (fornita) incapsula tutta la logica di disegno di una lista di punti su un piano cartesiano sovrapposto a una ***GeoMap***. Sono forniti metodi per:

- disegnare un singolo ***Point***, una lista di ***Point***, o una retta fra due ***Point*** (`drawPoint/-s, drawLine`)
- disegnare un parallelo o una lista di paralleli alla/e latitudine/i specificata/e (`drawParallel/-s`)
- impostare il colore di disegno (default: blu) (`setDrawingColor`)
- ridisegnare la mappa allo stato iniziale (ossia senza tracce di voli sopra) (`redrawMap`)

e) La classe ***MainPane*** (da realizzare) implementa il pannello principale che contiene la barra comandi (in alto) e il ***GeoCanvas*** (al centro), gestendo gli opportuni eventi per ottenere il comportamento desiderato. In particolare

- la gestione dell'evento dev'essere incapsulata in un apposito metodo `myHandler`
- nella gestione dell'evento, in caso la lettura del file non vada a buon fine deve comparire un apposito dialogo col messaggio appropriato, distinto per tipo di eccezione, avvalendosi del metodo statico `Controller.alert`

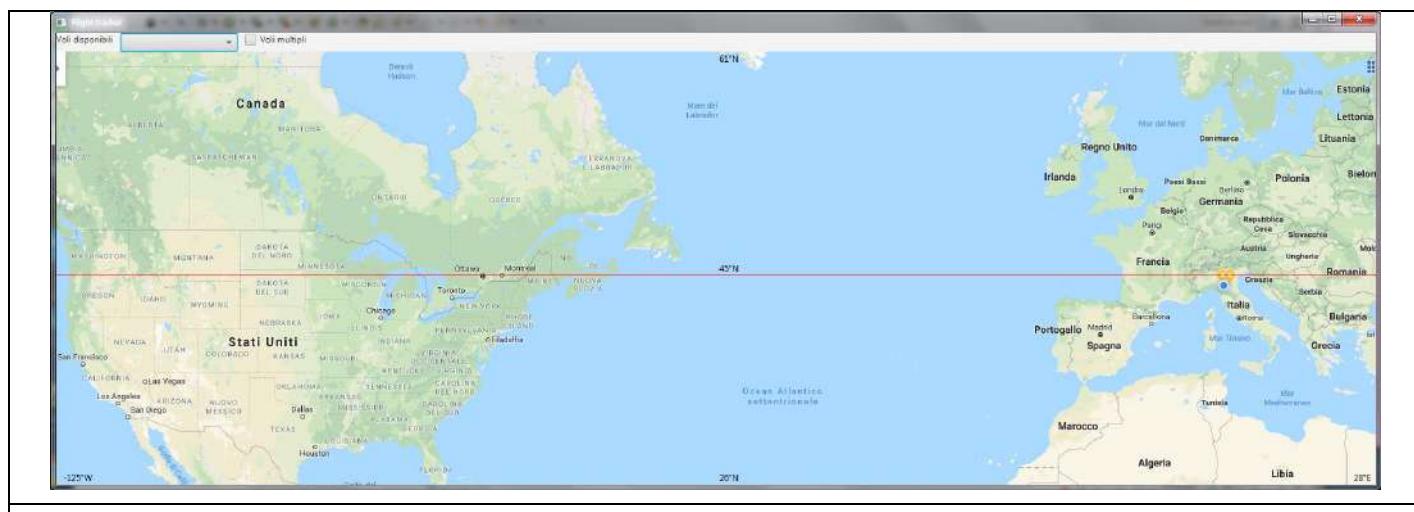


Fig. 1

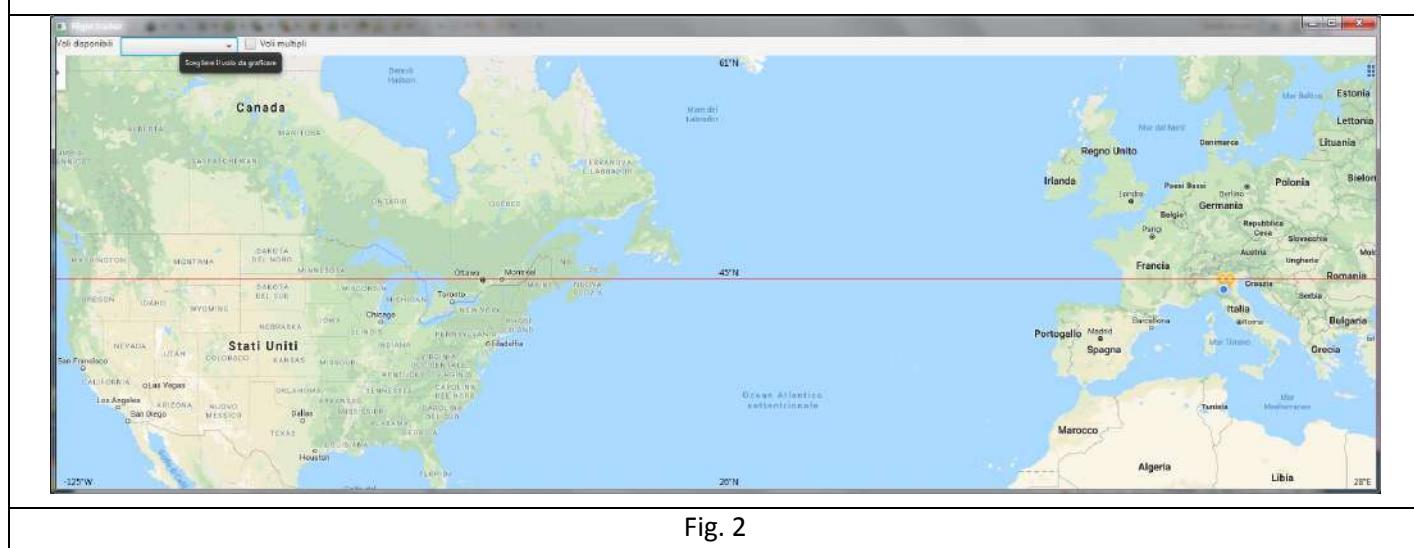


Fig. 2

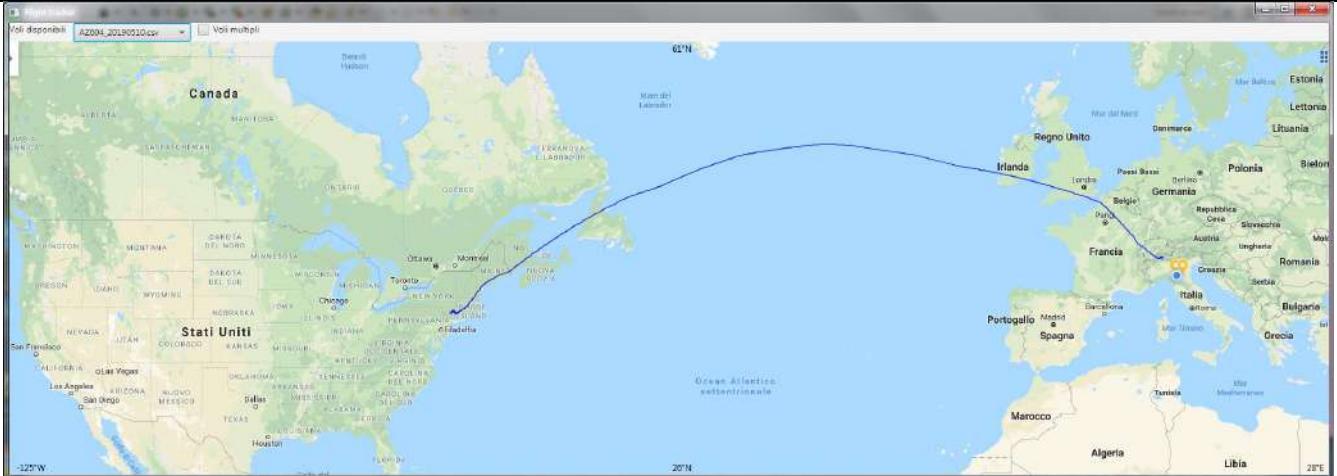


Fig. 3

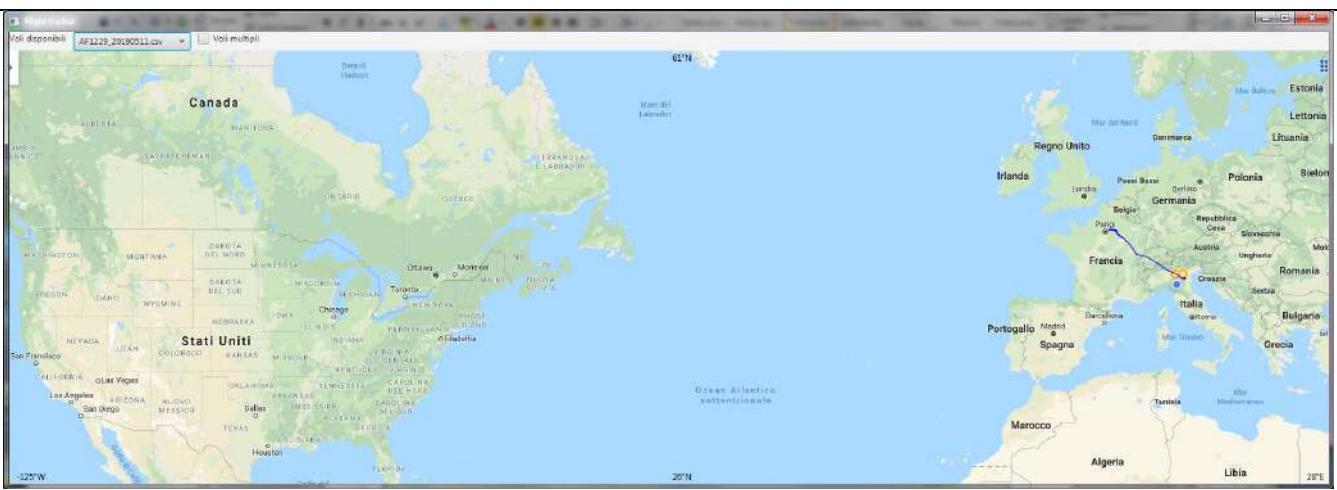


Fig. 4

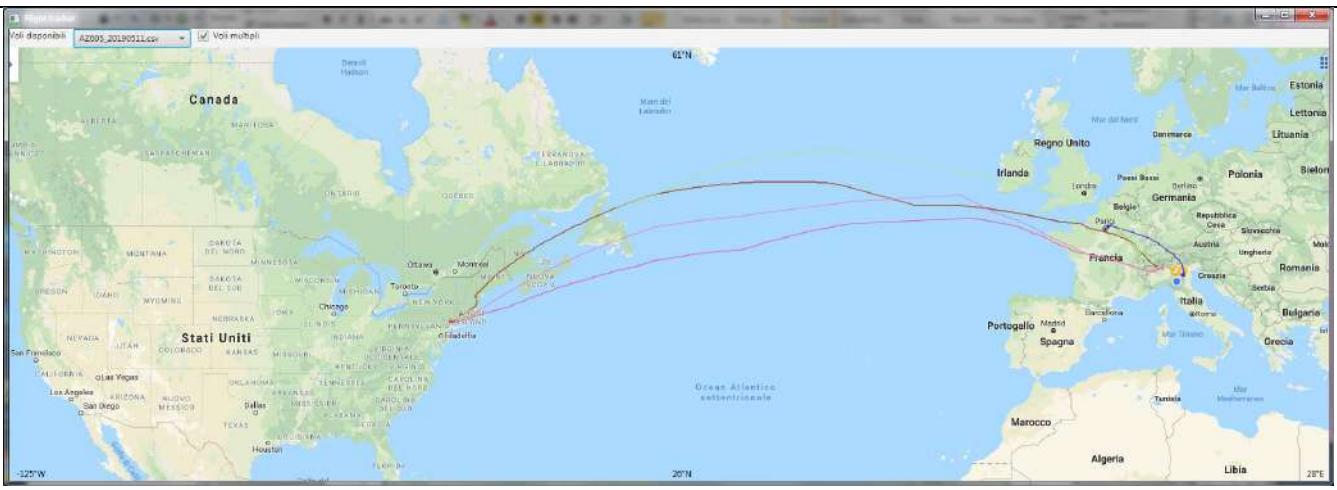


Fig. 5

ESAME DI FONDAMENTI DI INFORMATICA T-2 dell' 11/6/2019

Proff. E. Denti, R. Calegari, A. Molesini – Tempo: 4 ore

NOME PROGETTO ECLIPSE: CognomeNome-matricola (es. RossiMario-0000123456)

NOME CARTELLA PROGETTO: CognomeNome-matricola (es. RossiMario-0000123456)

NOME ZIP DA CONSEGNARE: CognomeNome-matricola.zip (es. RossiMario-0000123456.zip)

NB: l'archivio ZIP da consegnare deve contenere l'intero progetto Eclipse

Si ricorda che compiti non compilabili o palesemente lontani da 18/30 NON SARANNO CORRETTI e causeranno la verbalizzazione del giudizio “RESPINTO”

A supporto dei genitori che devono mettere a letto i bambini raccontando ogni sera una favola diversa, è stato richiesto lo sviluppo dell'applicazione *Il Favoliere*, in grado di sintetizzare automaticamente brevi favole a partire da alcuni elementi noti.

DESCRIZIONE DEL DOMINIO DEL PROBLEMA

Tutte le favole seguono uno schema generale prestabilito, che si compone di quattro parti

- un esordio (“C’era una volta...”) che introduce i *personaggi*
- lo scenario in cui essi si trovano
- una fase di azione in cui essi agiscono
- infine, una conclusione che porta all’inevitabile lieto fine.

I personaggi possono essere positivi (principi, cavalieri, principesse...) o negativi (orchi, lupi, personaggi malvagi vari): ogni scenario contiene sempre almeno due personaggi positivi e uno negativo, che si confrontano nell’azione – con ovvia vittoria finale dei buoni, altrimenti i bimbi non dormono e non fanno dormire neanche i genitori.

A seconda del numero di personaggi, della complessità dello scenario e del tipo di azione, la favola può essere più o meno adatta a bambini di diversa fascia d’età (piccolissimo, piccolo, grandicello, grande) e impressionabilità (per nulla impressionabile, poco impressionabile, impressionabile, molto impressionabile). La complessità dello scenario e la durezza dell’azione sono espresse tramite due indici numerici da 1 (minimo) a 5 (massimo).

Dev’essere garantito il rispetto di due vincoli chiave:

- i bambini *piccolissimi* non devono essere esposti a scenari di *complessità* superiore a 2, i *piccoli* non devono essere esposti a scenari di *complessità* superiore a 3
- i bambini *impressionabili* non devono essere esposti ad azioni di *durezza* superiore a 3, quelli *molto impressionabili* non devono essere esposti ad azioni di *durezza* superiore a 2

Il genitore specifica nella GUI solo la fascia d’età e il grado di impressionabilità del bambino: al resto pensa il sistema.

Cose da ricordare

- salva costantemente il tuo lavoro, facendo ZIP parziali e consegne parziali (vale l’ultima)
- in particolare, se ora compila e stai per fare modifiche, salva la versione attuale (non si sa mai..)

Checklist di consegna

- Hai fatto un unico file ZIP contenente l'intero progetto?
- In particolare, ti sei assicurato di aver incluso tutti i file .java (e non solo i .class)?
- Hai controllato che si compili e ci sia tutto ? [NB: non serve includere il PDF del testo]
- Hai rinominato IL PROGETTO esattamente come richiesto?
- Hai chiamato IL FILE ZIP esattamente come richiesto?
- Dopo aver caricato il file su Esamix, hai premuto il tasto “CONFERMA”, ottenendo il messaggio “Hai concluso l’esame”?

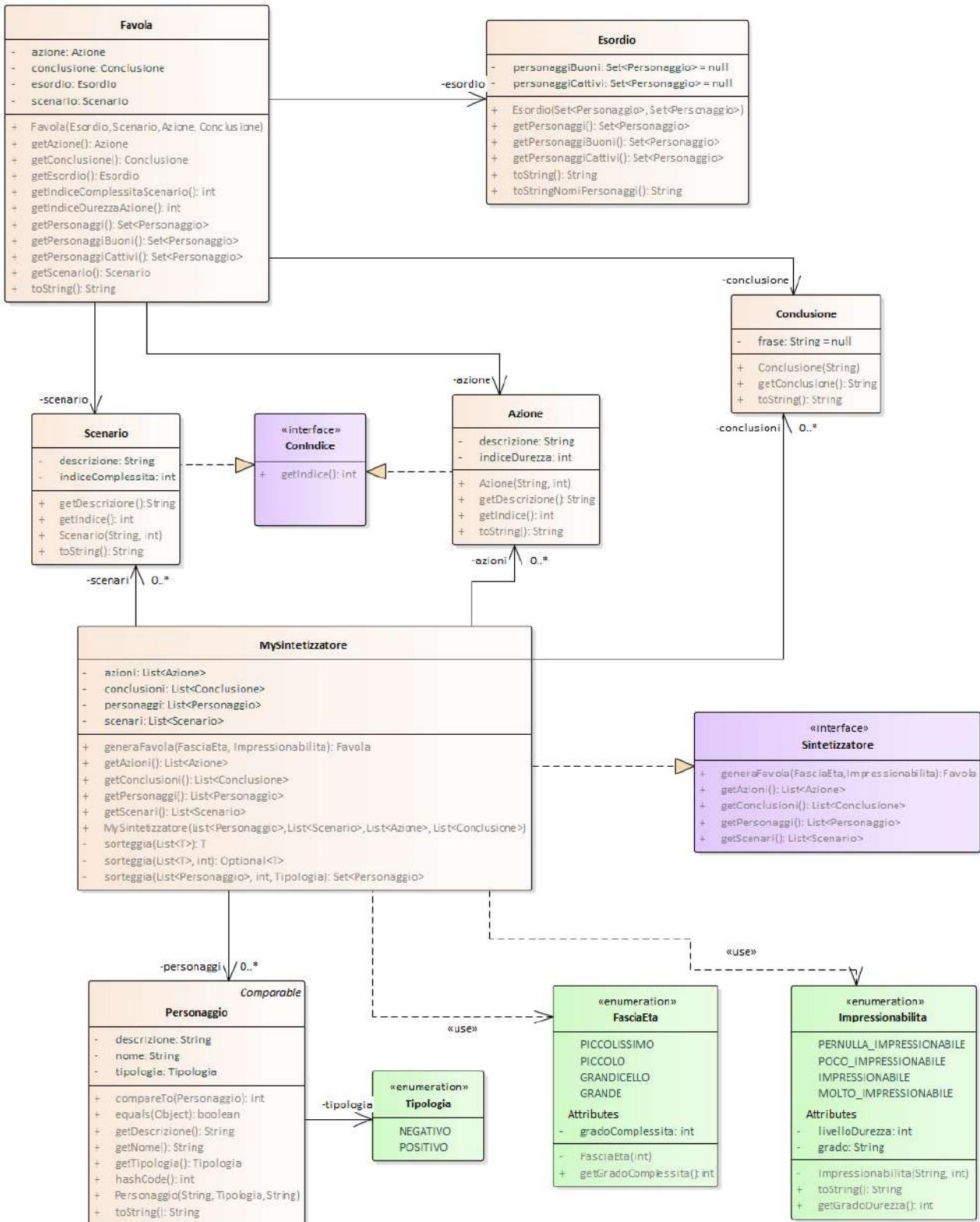
Parte 1

(punti: 20)

Dati (package favoliere.model)

(punti: 10)

Il modello dei dati deve essere organizzato secondo il diagramma UML sotto riportato: **da notare che l'unica classe da realizzare è MySintetizzatore**, essendo tutte le altre fornite già pronte nello Start Kit.



SEMANTICA:

- a) L'enumerativo **FasciaEta** (fornito) rappresenta le fasce d'età dei bimbi (piccolissimi, piccoli, grandicelli, grandi), unitamente al livello di complessità massima corrispondente: in altri termini, a ogni valore dell'enumerativo è associata la complessità massima accettabile per bambini di quell'età, recuperabile tramite apposito accessor

- b) L'enumerativo **Impressionabilità** (fornito) rappresenta i diversi gradi di impressionabilità dei bambini (molto impressionabile, impressionabile, poco impressionabile, per nulla impressionabile) unitamente – anche qui – al livello di durezza massima associato, anch'esso recuperabile tramite apposito metodo accessor
- c) L'enumerativo **Tipologia** (fornito) rappresenta le due tipologie di personaggi (negativo e positivo)
- d) La classe **Favola** (fornita) rappresenta la composizione delle quattro parti descritte nel dominio nel problema, specificate da altrettante classi (**Esordio**, **Scenario**, **Azione**, **Conclusione**): in particolare
 - **Esordio** è composto di **Personaggi**, buoni e cattivi, tutti distinti
 - **Scenario** e **Azione** implementano l'interfaccia **ConIndice**, che cattura il fatto di avere un indice numerico (ciò risulterà utile in fase di sintesi della favola, per trattare scenari e azioni in modo uniforme).
- e) Le quattro classi **Esordio**, **Scenario**, **Azione**, **Conclusione** specificano le rispettive parti della favola, alimentate ognuna tramite un opportuno **SectionLoader<E>** (vedere oltre per i dettagli)
- f) L'interfaccia **Sintetizzatore**, che rappresenta il sintetizzatore delle favole, è fornita nello start kit. A parte gli ovvi accessori, il metodo principale **generaFavola** genera una favola a partire dalla fascia d'età e dal livello di impressionabilità richiesti, lanciando **NoSuchTaleException** (fornita) in caso ciò non sia possibile (perché nessuno scenario o azione rispetta i vincoli minimi di complessità/durezza, o per mancanza di personaggi)
- g) la classe **MySintetizzatore (da realizzare)** implementa **Sintetizzatore** concretizzando il particolare algoritmo di sintesi espresso dal dominio del problema, ovvero:
 - ogni favola ha sempre due personaggi positivi e uno negativo, sorvegliati fra quelli disponibili
 - lo scenario e l'azione sono sorvegliati fra quelli disponibili del giusto grado di complessità/durezza
 - la conclusione è semplicemente sorvegliata fra quelle disponibili

SUGGERIMENTO: predisporre tre funzioni ausiliarie **sorveggia(...)** che fattorizzino il sorteggio delle diverse parti, sfruttando gli **Optional** per esprimere l'eventuale assenza di risultato nel caso con indice (vedere UML).

In particolare:

- **Set<Personaggio> sorveggia(List<Personaggio> lista, int n, Tipologia tipo)**
sorveggia un numero **n** di personaggi di tipologia **tipo** dalla lista data (utile per il sorteggio di buoni e cattivi)
- **<T> T sorveggia(List<T> lista)** sorveggia un elemento dalla lista data (utile per le **Conclusioni**)
- **<T extends ConIndice> Optional<T> sorveggia(List<T> lista, int upperBound)** sorveggia dalla lista data un elemento che abbia un indice inferiore all'**upperBound** passato (utile per il sorteggio di **Scenario** e **Azione**, che devono essere scelti entro un indice prestabilito).

Persistenza (package favoliere.persistence)

(punti 10)

La persistenza è strutturata su quattro file di testo:

- Il file di testo **Personaggi.txt** contiene l'elenco dei personaggi (uno per riga): ogni riga specifica innanzitutto se il personaggio sia positivo o negativo, nella forma “**POSITIVO:**” o “**NEGATIVO:**”, poi ne dà la denominazione e infine, dopo un ulteriore “**:**”, la descrizione.

```
POSITIVO: il principe William : un principe alto e nobile, sempre in sella al fido Alfius, il suo cavallo bianco
NEGATIVO: Urcus : un orco molto cattivo e feroce
...
```

- Il file di testo **Scenari.txt** contiene l'elenco degli scenari (uno per riga): ogni riga contiene prima la descrizione, poi – nella forma “**#N**” – l'indice numerico di complessità (dove N è un intero da 1 a 5)

```
nel bosco incantato di Dentilandia #2
nelle prigioni dell'alta torre del paese di Dentinia #4
...
```

- Il file di testo **Azioni.txt** contiene l'elenco delle azioni (una per riga), descritte tramite semplici frasi: ogni riga contiene la frase, seguita – nella forma “**#N**” – dall'indice numerico di durezza (dove N è un intero da 1 a 5)

```

andò a salvarla con la spada #2
lanciò un incantesimo per carbonizzare gli avversari #4
...

```

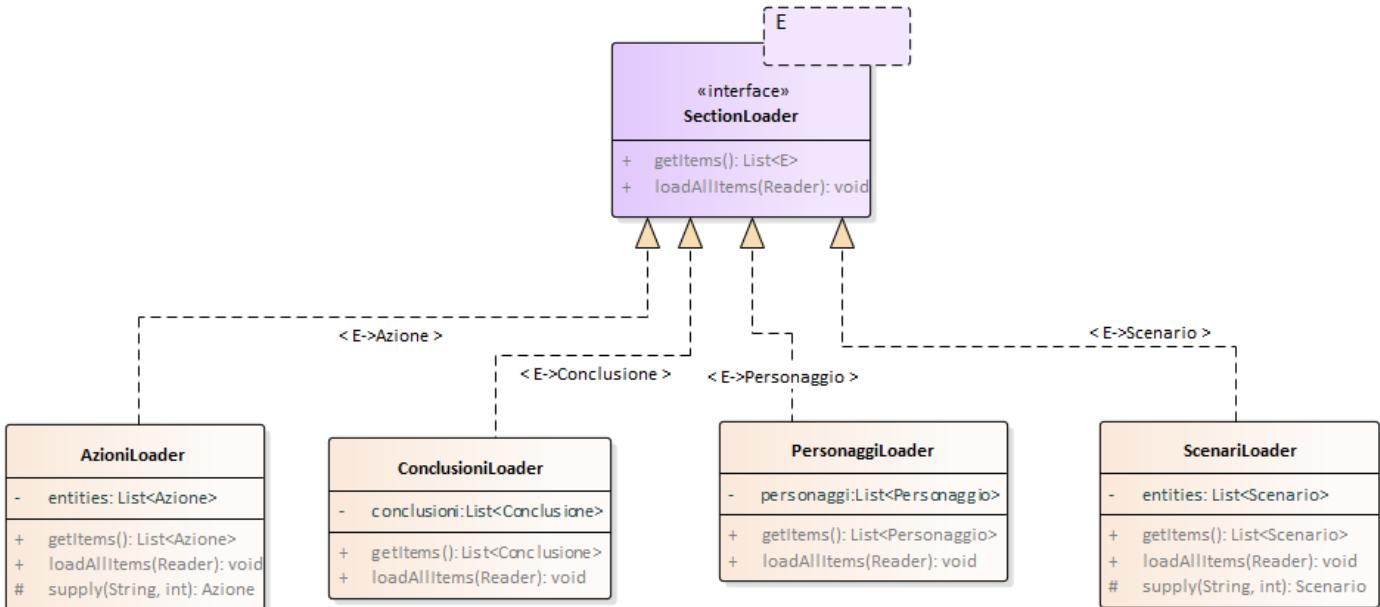
- Il file di testo **Conclusioni.txt** contiene l'elenco delle frasi di chiusura (una per riga)

```

Vissero così infine tutti felici e contenti
Dopo una tale impresa, il cattivo fu sconfitto, i buoni si sposarono e vissero felici per molti anni
...

```

La struttura di questa parte dell'applicazione è illustrata nel diagramma UML sotto riportato.



SEMANTICA:

- L'interfaccia tipizzata **SectionLoader<E>** specifica il reader della generica sezione della favola (destinato quindi a leggere secondo i casi *personaggi*, *scenari*, *azioni*, *conclusioni*):
 - il metodo `getItems` restituisce una `List<E>` (con `E` che sarà nei vari casi rispettivamente **Personaggio**, **Scenario**, **Azione** o **Conclusione**, come indicato con apposite frecce nei commenti sull'UML stesso)
 - il metodo `loadAllItems` si occupa della lettura del file lanciando le eccezioni **IOException** o **BadFormatException**: la prima viene lanciata in caso si verifichino errori di IO, la seconda nel caso in cui il formato del file non sia quello che ci si aspetta; in questo secondo caso, la stringa passata all'eccezione deve segnalare in modo preciso l'errore riscontrato nel file (es. "indice non numerico", "descrizione vuota", "mancanza tipologia in personaggio: XXXXXXXX:").
- La classe **ConclusioniLoader**, che implementa **SectionLoader<Conclusione>**, è fornita nello start kit
- Le tre classi **PersonaggiLoader**, **ScenariLoader** e **AzioniLoader**, sono invece da realizzare e precisamente:
 - PersonaggiLoader** implementa **SectionLoader<Personaggio>**
 - ScenariLoader** e **AzioniLoader**, quasi identiche data l'identica struttura dei file, implementano rispettivamente **SectionLoader<Scenario>** e **SectionLoader<Azione>**.

Si suggerisce (come indicato nel diagramma UML) di fattorizzare in un metodo ausiliario `supply` la creazione dello specifico oggetto **Scenario/Azione**, così da rendere il più possibile invariante il codice dei due loader.

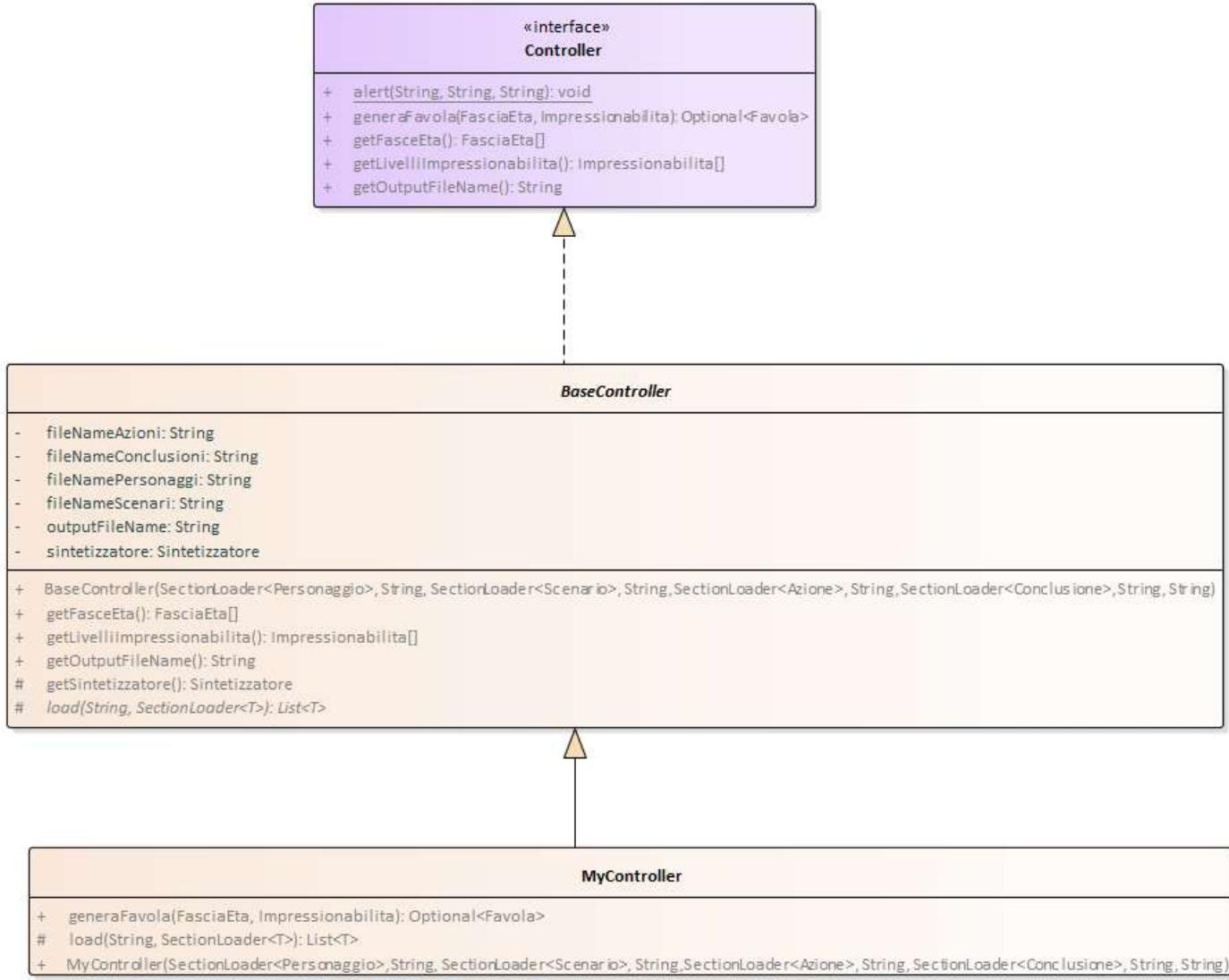
IMPORTANTE: per consentire il test della GUI anche nel caso di *reader* non funzionanti, è fornita la classe **FavoliereMock** che replica **FavoliereApp** utilizzando dati fissi pre-cablati al posto di quelli letti da file (utilizzando quindi un **ControllerMock**).

L'applicazione deve permettere all'utente-genitore di specificare semplicemente età ed indole del bambino.

Con riferimento alle figure seguenti: premendo il pulsante **Genera favola**, l'applicazione deve generare e mostrare sull'area di testo la favola richiesta. Successivamente, ove venga premuto il pulsante **Salva su file**, la favola sarà trascritta sul file di testo **Favola.txt**.

Controller (package `favoliere.ui.controller`)

- a) L'interfaccia **Controller** (fornita) specifica il controller dell'applicazione, che dichiara metodi per
 - recuperare l'insieme dei possibili livelli di impressionabilità e delle possibili fasce d'età
 - recuperare il nome del file di uscita su cui stampare (sua proprietà privata)
 - generare una favola coi livelli di impressionabilità e fascia d'età specificati (il risultato è un Optional perché potrebbero non esserci scenari, azioni o personaggi adatti a tutte le combinazioni)
- b) La classe **BaseController** (fornita) concretizza parzialmente **Controller** prevedendo un unico costruttore che riceve in ingresso tutti gli argomenti necessari (v. UML) e implementando tutti i metodi tranne `generaFavola`; espone inoltre due metodi ausiliari protetti rispettivamente per rendere accessibile dalle sottoclassi il sintetizzatore (metodo accessor `getSintetizzatore`) e per caricare un file tramite il giusto **SectionLoader** (metodo `load`)
- c) La classe **MyController** (anch'essa fornita ☺) estende **BaseController** implementando
 - a. il metodo `generaFavola`: lo fa delegando di fatto il lavoro al sintetizzatore, ma catturando l'eventuale eccezione **NoSuchTaleException** (da questi lanciata) gestendola in modo da restituire l'optional atteso.
 - b. Il metodo `load` che si occupa della lettura della sezione interessata **gestendo opportunamente eventuali eccezioni**: ☺ in particolare, in caso il sistema non riesca a caricare uno dei file (personaggi, scenari, azioni, conclusioni) o vi siano errori di formato, deve far comparire un apposito dialogo col messaggio appropriato (Fig. 4) tramite il metodo statico **Controller.alert**.



Interfaccia utente (package favoliere.ui)

(punti 10)

L’interfaccia utente deve essere simile (non necessariamente identica) all’esempio mostrato di seguito.

Se il caricamento preliminare (realizzato nell’**FavoliereApp** fornita nello start kit) ha esito positivo, compare la finestra principale dell’applicazione (Fig. 1), costituita da un’istanza di **MainPane (da realizzare)** che riceve come unico argomento il **Controller**.

Essa deve mostrare innanzitutto due combobox, precaricate coi valori standard delle fasce d’età e dei gradi di impressionabilità, avendo cura che sia già selezionato un opportuno valore iniziale.

Premendo il pulsante **Genera favola**, l’applicazione genera e mostra sull’area di testo a lato la favola richiesta (Fig. 2): cioè abilita il pulsante **Salva su file**, in precedenza disabilitato, per l’eventuale scrittura della favola sul file di testo. **Nel caso in cui non sia possibile generare una favola con le specifiche richieste, si deve far comparire nell’area di testo il messaggio "impossibile generare una favola coi vincoli richiesti".**

Cambiando le specifiche di età e/o di impressionabilità, si possono generare via via nuove favole (Fig. 3).

Come già anticipato, premendo il pulsante **Salva su file**, la favola precedentemente generata viene stampata sul file di output fornito dal controller. In caso non ci sia alcuna favola da stampare o vi siano errori di scrittura, deve comparire un apposito dialogo col messaggio appropriato (metodo statico **Controller.alert**).

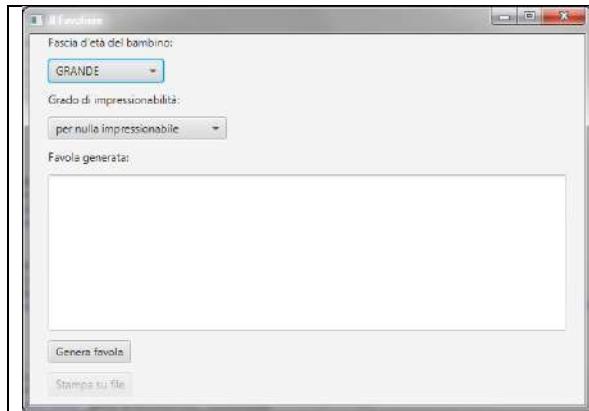


Fig. 1



Fig. 2

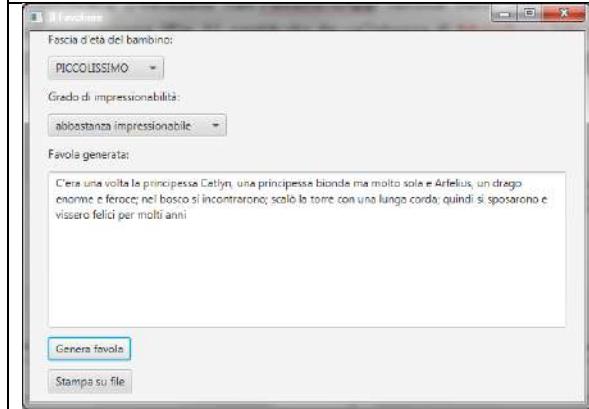


Fig. 3

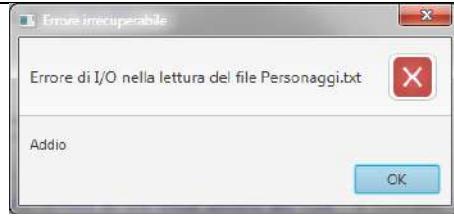


Fig. 4

ESAME DI FONDAMENTI DI INFORMATICA T-2 del 6/2/2019

Proff. E. Denti – R. Calegari – G. Zannoni

Tempo: 4 ore

NOME PROGETTO ECLIPSE: CognomeNome-matricola (es. RossiMario-0000123456)

NOME CARTELLA PROGETTO: CognomeNome-matricola (es. RossiMario-0000123456)

NOME ZIP DA CONSEGNARE: CognomeNome-matricola.zip (es. RossiMario-0000123456.zip)

NB: l'archivio ZIP da consegnare deve contenere l'intero progetto Eclipse

Si vuole sviluppare una semplice app per giocare a *Tris* (detto anche *TicTacToe*, v. figura a lato).

X	O	X
O	X	O
X	O	X

DESCRIZIONE DEL DOMINIO DEL PROBLEMA

Tris si gioca su una scacchiera 3x3 fra due giocatori, che utilizzano due simboli distinti (di norma X e O) per marcare le proprie mosse. I giocatori si alternano: vince il primo che riesce a porre tre suoi simboli in sequenza (orizzontale, verticale o diagonale). Se nessuno ci riesce, non ci sono vincitori e il gioco finisce in parità.

ENTITÀ COINVOLTE

Si desidera che ogni **partita** sia identificata in modo univoco dal timestamp (in formato ISO) reattivo all'istante di inizio della partita stessa. A ogni partita è associata la lista delle configurazioni di **scacchiera** che la descrivono, a partire dalla scacchiera vuota iniziale fino alla configurazione finale in cui o uno vince (cosa che può avvenire anche prima che tutte le nove caselle siano state usate), o il gioco termina in parità. Ogni mossa genera quindi una nuova configurazione della scacchiera, che viene aggiunta in coda alla lista.

ALGORITMO DI GIOCO

Dopo ogni mossa occorre verifica l'eventuale vittoria da parte di un giocatore controllando la presenza di tre simboli uguali in riga, colonna o diagonale.

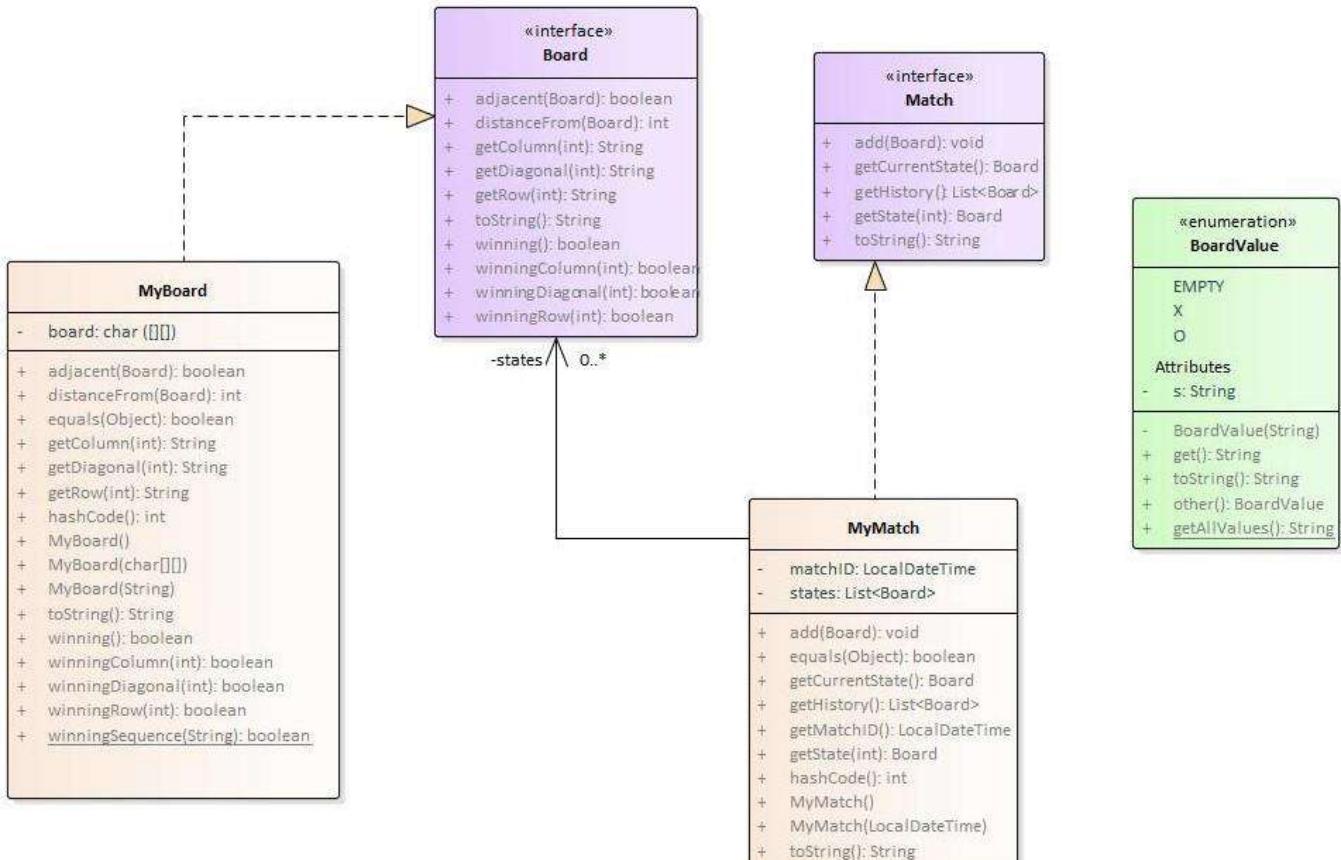
Parte 1

(punti: 19)

Dati (namespace tris.model)

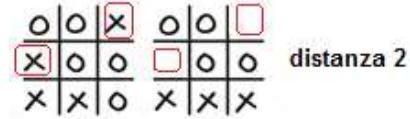
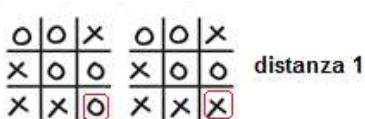
(punti: 17)

Il modello dei dati deve essere organizzato secondo il diagramma UML di seguito riportato:



SEMANTICA:

- a) L'enumerativo **BoardValue** (fornito) definisce i tre stati possibili per ogni singola cella della scacchiera (EMPTY, X, O): a ogni valore simbolico è associata la stringa corrispondente (per EMPTY si usa lo spazio " "), recuperabile sia tramite l'accessore `get` sia tramite `toString`. Il metodo di utilità `other` restituisce invece l' "altro" simbolo rispetto all'attuale (ovviamente è indefinito nel caso il simbolo corrente sia EMPTY), mentre il metodo statico `getAllValues` restituisce la stringa coi tre valori possibili concatenati (ossia, " XO").
- b) l'interfaccia **Board** (fornita) descrive la scacchiera, intesa come matrice 3x3 di **BoardValue**: per convenzione, righe e colonne sono numerate da 0 a 2, mentre le due diagonali sono numerate da 0 a 1. I metodi:
- `getRow`, `getColumn` e `getDiagonal` consentono di recuperare rispettivamente la singola riga, colonna o diagonale di indice pari all'argomento ricevuto, restituendo la stringa corrispondente al contenuto (ad esempio, "XXO", "O O", etc.): lanciano **IllegalArgumentException** se l'indice è fuori range (ossia se non è compreso nell'intervallo 0-2 per `getRow`, `getColumn` e 0-1 per `getDiagonal`);
 - analogamente `winningRow`, `winningColumn` e `winningDiagonal` verificano rispettivamente se la riga, colonna o diagonale specificata (tramite il solito indice nel range 0-2 o 0-1, come sopra) sia vincente, ossia contenga tre simboli uguali;
 - `winning` verifica se la scacchiera nel suo complesso sia vincente, ossia se in essa vi sia almeno una riga, colonna, o diagonale vincente;
 - `distanceFrom` calcola la distanza fra la scacchiera corrente e quella fornita come argomento, intendendosi con ciò il numero di celle diverse fra le due configurazioni di scacchiera (vedi esempi illustrati di seguito);



- `adjacent` verifica se la scacchiera corrente sia adiacente a quella fornita come argomento, intendendosi con ciò che abbia distanza 1 da essa e inoltre che il numero di X e di O presenti nella scacchiera fornita come argomento sia quasi identico (più precisamente, che il numero di X differisca al più di 1 dal numero di O): ciò serve a evitare scacchiere "illegali" in cui lo stesso giocatore faccia due mosse consecutive
 - `toString` deve restituire una stringa adatta a visualizzare la scacchiera su console, quindi con tre simboli per riga separati fra loro da tabulazioni e andando a capo dopo le prime due righe (ma non dopo l'ultima!). Per portabilità fra le piattaforme, il simbolo di "a capo" non dev'essere cablato come '\n' ma dev'essere espresso tramite `System.lineSeparator`.
- c) la classe **MyBoard** (da realizzare) implementa **Board** come da specifica, con le seguenti ulteriori precisazioni:
- devono essere previsti due costruttori, uno con argomento matrice 3x3 di caratteri, l'altro con argomento stringa di 9 caratteri consecutivi da intendersi letti dall'alto al basso e da sinistra a destra, senza separatori di alcun tipo: entrambi i costruttori lanciano **IllegalArgumentException** se l'argomento non ha la giusta dimensione o qualche carattere è diverso dai tre ammessi (quelli associati ai valori di **BoardValue**)
 - devono essere implementate anche **equals** e **hashcode**: in particolare, **per hashcode è sufficiente quella standard generata automaticamente da Eclipse** (`menu Source → Generate hashCode and equals`) mentre **equals** deve considerare uguali due scacchiere a distanza zero una dall'altra;
- d) l'interfaccia **Match** (fornita) descrive la partita intesa come sequenza di configurazioni di scacchiera. Metodi:
- `getCurrentState` restituisce lo stato attuale della scacchiera (che esiste sempre, perché la partita inizia con una scacchiera vuota, ossia con tutte le celle in stato EMPTY)
 - `getState` restituisce lo stato i-esimo della scacchiera: se l'indice è fuori range, ossia supera quello dell'ultimo stato disponibile (quello corrente), viene lanciata **IllegalArgumentException**

- `getHistory` restituisce la lista di tutti gli stati, dalla scacchiera vuota iniziale fino allo stato finale
 - `add` aggiunge sequenza di configurazioni una nuova scacchiera, che diviene lo stato corrente, *purché il nuovo stato sia adiacente a quello precedente*: altrimenti, lancia `IllegalArgumentException`
 - `toString` deve restituire una stringa *adatta a visualizzare su console l'intera partita*, intesa come sequenza delle relative scacchiere separate fra loro da `System.lineSeparator` (non dopo l'ultima); anche qui il simbolo di “a capo” *non* dev'essere cablato come '\n' ma dev'essere espresso tramite `System.lineSeparator`.
- e) la classe **MyMatch (da realizzare)** implementa **Match** come da specifica, con le seguenti ulteriori precisazioni:
- devono essere previsti due costruttori, uno con argomento l'identificativo univoco `LocalDateTime` specificato, l'altro senza argomenti (che utilizza giorno e ora correnti). Entrambi devono inizializzare la lista delle configurazioni con la scacchiera iniziale vuota (ossia, con tutte le celle nello stato `EMPTY`)
 - devono essere fornite due implementazioni standard di `equals` e `hashCode` che si suggerisce di far generare automaticamente da Eclipse (*menu Source → Generate hashCode and equals*)

Persistenza (namespace tris.persistence)

(punti: 2)

Obiettivo di questo componente è stampare su file una partita, secondo il diagramma UML di seguito riportato.

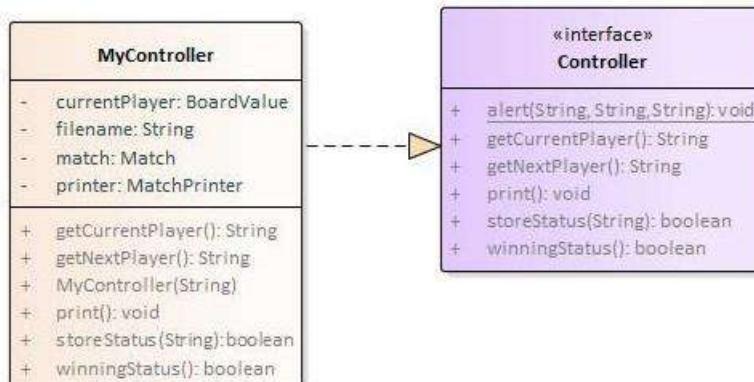


SEMANTICA:

- a) l'interfaccia **MatchPrinter** (fornita) dichiara il solo metodo `print` che stampa un **Match** sul **PrintWriter** fornito.
- b) la classe **MyMatchPrinter (da realizzare)** implementa tale metodo, *assicurando che dopo ogni stampa il buffer di uscita venga opportunamente svuotato*.

Parte 2

(punti: 11)



Controller (namespace tris.controller)

Il controller – articolato in interfaccia e implementazione – è fornito già pronto: *il suo stato interno mantiene lo stato della partita con annessi e connessi* (giocatore attuale, etc.) *nonché il printer da usare per le stampe*. Conseguentemente, i suoi metodi fanno da ponte fra quelli del model e quelli del printer. In particolare:

- `storeStatus` aggiunge alla partita corrente il nuovo stato di scacchiera ricevuto come argomento (sotto forma di stringa di 9 caratteri), verificando al contempo se esso sia vincente (restituisce a tal fine un boolean)
- `winningStatus` verifica se la partita corrente è vinta
- `getCurrentPlayer` restituisce la stringa corrispondente al giocatore corrente (“X” o “O”)

- `getNextPlayer` cambia il giocatore corrente (da "X" a "O", da "O" a "X") e restituisce la stringa corrispondente
- `print` stampa su file la partita corrente (che si suppone terminata)

L'interfaccia **Controller** offre altresì il metodo statico `alert` per far comparire una finestre di dialogo utile a segnalare errori all'utente (in questa applicazione, solo nel caso in cui la stampa su file fallisca per qualche motivo).

GUI (namespace `tris.ui`)

(punti: 11)

L'interfaccia grafica dev'essere simile (non necessariamente identica) a quella sotto illustrata.



La classe **TrisApp** (fornita) contiene come sempre il main di partenza dell'applicazione.

La classe **TrisPane** (pure fornita) definisce il pannello con gli elementi grafici principali (Fig.1), ovvero:

- al top, l'etichetta con l'indicazione del giocatore corrente (ossia, se tocca a X o a O)
- a sinistra la griglia-scacchiera, sotto forma di istanza della classe **TrisGrid**
- a destra, la textarea su cui far comparire via via lo stato della partita
- in basso, il pulsante *Stampa* (inizialmente disabilitato, si abilita solo quando la partita termina)

La classe **TrisGrid** (da realizzare) deve implementare la griglia e l'algoritmo di gioco, interfacciandosi opportunamente con **TrisPane**. Più esattamente:

- il costruttore riceve da **TrisPane** tutti gli elementi su cui deve poter operare e alloca una griglia 3x3 di **Button**, impostati con font *Courier New* grassetto *20 punti* e inizializzati vuoti (ossia col testo corrispondente allo stato **BoardValue.EMPTY**) (Fig. 1). **SUGGERIMENTO:** utilizzare un *GridPane*, ricordando però che il metodo `add/4` prevede, in modo alquanto contro-intuitivo, *prima* l'indice di colonna, *poi* quello di riga.

La pressione di uno dei 9 pulsanti della griglia deve causare l'invocazione di un opportuno metodo di gestione eventi, per comodità denominato `handle(Button)`

- il metodo `toString` fornisce la rappresentazione stringa dello stato attuale della griglia-scacchiera, nel formato di stringa a 9 caratteri consecutivi (adatto per il costruttore di **Board**)
- quando il giocatore corrente preme uno dei 9 pulsanti della griglia-scacchiera di gioco, `handle`:
 - imposta nel bottone premuto il simbolo del giocatore corrente (X o O)
 - disabilita il bottone stesso, in modo che non sia più cliccabile (cella già occupata)
 - aggiunge alla partita corrente, tramite il **Controller** (metodo `storeStatus`), la nuova configurazione di scacchiera appena determinatasi, *verificando contemporaneamente* se qualcuno abbia vinto;
 - appende sulla textarea una nuova riga corrispondente alla nuova situazione della partita (Figg. 2,3,4): tale riga deve includere la configurazione della scacchiera in forma compatta (9 caratteri) e l'indicazione se il gioco continua o la partita è vinta (es. "continua" / "vittoria", "partita patta" o analogo);
 - se lo stato così ottenuto non è finale (vincente o patta) impone, tramite il Controller, di cambiare giocatore (metodo `getNextPlayer`) e adegua la visualizzazione del giocatore nella label (Figg. 1,2,3,...);

- se, invece, lo stato così ottenuto è vincente o partita patta agisce di conseguenza (invocando un metodo di gestione ausiliario che per comodità chiameremo `endGame`) e nell'ordine:
 - disabilita tutti i button della griglia-scacchiera (perché la partita è finita)
 - abilita il pulsante *Stampa* (Fig. 4 o 5) associandolo al suo gestore eventi, così che, se premuto, esso effettui la stampa della partita (tramite `Controller.print`) e subito dopo ri-disabiliti il pulsante *Stampa stesso* (Fig. 6).

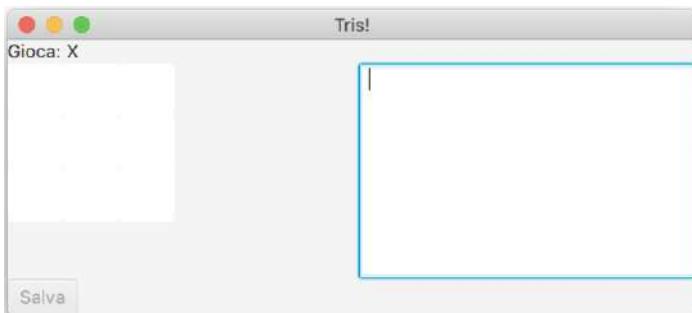


Figura 1



Figura 2

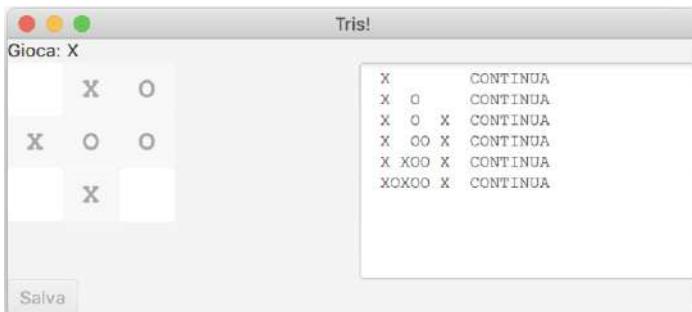


Figura 3



Figura 4



Figura 5

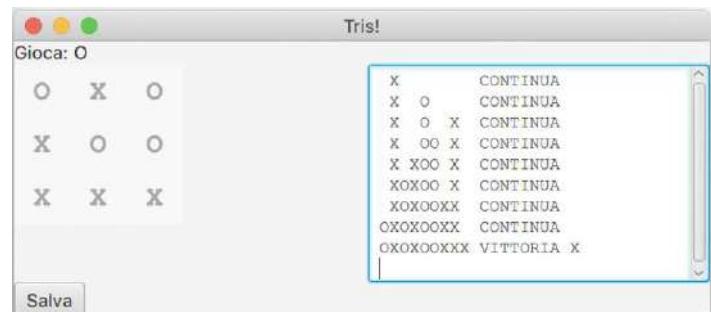


Figura 6

ESAME DI FONDAMENTI DI INFORMATICA T-2 del 9/1/2019

Proff. E. Denti – R. Calegari – G. Zannoni Tempo: 4 ore

NOME PROGETTO ECLIPSE: CognomeNome-matricola (es. RossiMario-0000123456)

NOME CARTELLA PROGETTO: CognomeNome-matricola (es. RossiMario-0000123456)

NOME ZIP DA CONSEGNARE: CognomeNome-matricola.zip (es. RossiMario-0000123456.zip)

NB: l'archivio ZIP da consegnare deve contenere l'intero progetto Eclipse

È richiesto di sviluppare un semplice ma efficace software, denominato *SafeRepo*, per la gestione di un repository di documenti *versionati*: l'obiettivo è cioè quello di mantenere tutte le versioni di un documento, consentendo di recuperare in ogni momento una versione precedente (undo), ripristinarla, etc., in modo pratico e sicuro.

DESCRIZIONE DEL DOMINIO DEL PROBLEMA

Un classico “spazio di archiviazione” – su disco o nel cloud, come Dropbox, Google Drive, etc. – consente di caricare file di ogni tipo, ma ne mantiene un’unica copia (l’ultima): caricando un nuovo file con lo stesso nome di uno già esistente, il precedente va perso.

Un **repository versionato**, invece, non cancella mai nulla: quando si carica una nuova versione di un documento, vengono comunque mantenute anche tutte le precedenti, con la relativa storia (numero di versione, data e ora di caricamento di ognuna). Ciò consente di:

- aggiungere una nuova versione senza mai perdere le precedenti;
- recuperare la versione corrente del documento;
- recuperare una qualunque versione precedente del documento, o indicandone il numero (ad esempio, “la versione n.3”), o, in alternativa, indicando l’istante (data/ora) che interessa (ad esempio, “la versione in essere al 30 dicembre 2018 alle ore 16:15”)
- cancellare virtualmente un elemento dal repository semplicemente aggiungendo una nuova versione vuota, senza con ciò perdere l’accesso a tutte le versioni precedenti.

Concretamente, occorre distinguere la **gestione logica** dalla **gestione fisica**:

- a livello logico, per ogni documento (identificato da un ID univoco) il sistema mantiene in memoria la lista delle sue versioni che però, per ovvi motivi di spazio, *non* incorporano fisicamente altrettante copie del file;
- a livello fisico, il sistema mantiene, in una cartella su disco, le varie copie dei file, opportunamente rinominate in modo da esprimere direttamente sia il numero di versione sia la data/ora corrispondente: da sottolineare che il sistema produce e manipola sempre e solo copie del file indicato dall’utente, mai l’originale.

È compito di *SafeRepo* rendere tutta questa gestione trasparente all’utente, che dovrà poter inserire e recuperare i suoi documenti senza preoccuparsi di dove siano fisicamente archiviati i file e come si chiamino.

MODELLO DEI DATI

Un **documento** è caratterizzato da un identificativo univoco (es. “doc1”) e dal percorso assoluto che identifica il corrispondente file su disco (ad esempio, “C:/enrico/doc1”).

Un **documento versionato** accoppia un documento a un numero di versione (intero crescente nel tempo) e un timestamp in formato ISO che ne caratterizza l’istante di caricamento (ad esempio, la versione 1 di “doc1”, caricata il 31/12/2018 alle 15:39:02.154254, ha come numero di versione 1, come timestamp 2018-12-31T15:39:02.154254 ed è associata al documento “C:/myrepo/doc1_1_2018-12-31T15_39_02.154254” – vedere oltre per i dettagli).

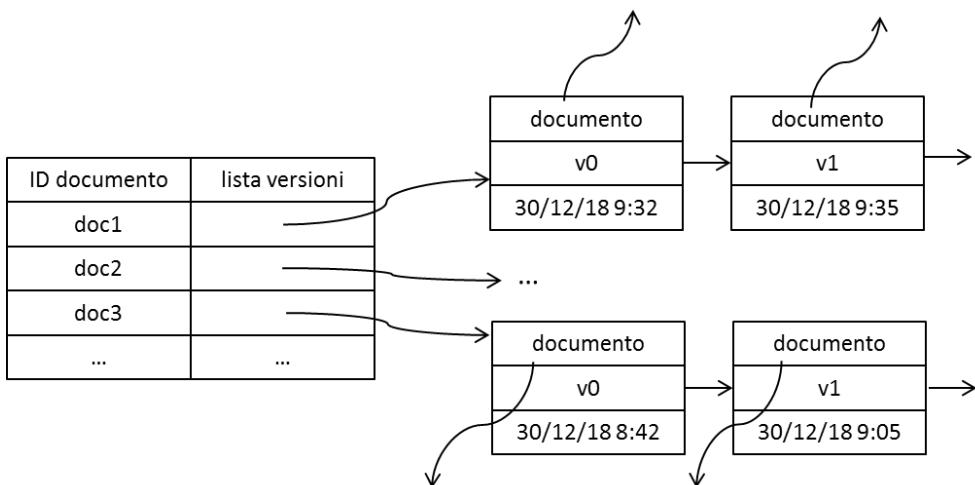
GESTIONE INTERNA DEL REPOSITORY

A livello logico, si desidera che il repository sia organizzato con una mappa che associa a ogni identificativo di documento la lista dei corrispondenti **documenti versionati**, come illustrato nella **figura sottostante**. Le nuove versioni vengono aggiunte in coda alla lista, così che i timestamp siano automaticamente ordinati in senso crescente,. Di conseguenza:

- la versione corrente è l’ultimo elemento della lista
- la versione N-ma è recuperabile accedendo all’N-mo elemento della lista ($N \geq 0$, $N < \text{size della lista}$)
- la versione in essere a un certo istante X è recuperabile filtrando solo gli elementi della lista con timestamp $T \leq X$, e prendendo poi quella corrispondente al massimo valore del timestamp.

Ad esempio, se del documento “doc1” esistono le 7 versioni v0 del 2018-12-30T09:39:02, v1 del 2018-12-30T09:39:03, v2 del 2018-12-30T09:40:04, v3 del 2018-12-30T09:40:05, v4 del 2018-12-30T09:42:06, v5 del 2018-12-30T09:42:07 e v6

del 2018-12-30T09:44:08, chiedendo di recuperare la versione valida all'istante X=2018-12-30T09:40:06 si deve ottenere in risposta la v3, in quanto le successive v4, v5 e v6 sono state inserite solo successivamente all'istante X.



GESTIONE PERSISTENZA NEL REPOSITORY

IMPORTANTE: per prevenire incompatibilità **utilizzare sempre le barre standard '/'** nei percorsi, anche su piattaforma Windows, evitando le barre inverse '\\' (che comunque devono, nel caso, essere espresse con la notazione '\\').

Directory di lavoro: il repository mantiene le sue copie private dei file in una sua cartella interna, configurata all'atto della creazione del repository stesso.

Aggiunta file: ogni volta che l'utente aggiunge al repository una nuova versione di un file, il repository effettua una nuova copia del file dell'utente e la colloca nella sua cartella interna: il nuovo file viene chiamato *IDdoc_versione_timestamp*, dove il carattere ':' presente nel timestamp in formato ISO (che non sarebbe accettato dal file system) è sostituito dal carattere '_' (underscore). Così, ad esempio, la versione 2 del documento "doc1", inserita il 31/12/2018 alle ore 15:39:03.318918, viene copiata in un file interno di nome "doc1_2_2018-12-31T15_39_03.318918": il *path* *documento* della versione 2 di doc1 è perciò "...../doc1_2_2018-12-31T15_39_03.318918", essendo "....." la directory di lavoro del repository (prefissata, come detto, all'atto della creazione del repository stesso).

Eliminazione file: se l'utente elimina un documento, le sue versioni precedenti vengono mantenute: viene quindi creata una nuova versione con un nuovo documento il cui *path* è posto a null. Ad esempio, se l'utente elimina il documento "doc2", di cui erano presenti (supponiamo) tre versioni, viene creata una nuova versione v4 in data/ora corrente, il cui documento ha identificativo "doc2" ma path null.

Ripristino file: se l'utente ripristina una versione precedente (ossia riporta logicamente il documento allo stato in cui era in un tempo precedente), viene semplicemente creata una nuova versione il cui *documento* punta allo stesso file a cui puntava la versione ripristinata. Ad esempio, se l'utente ripristina la versione 4 del documento "doc1" (che portava la data del 2018-12-30T09:42:06), viene creata una nuova versione 7 con timestamp corrente, il cui *documento* punta però allo stesso file a cui puntava la versione 4, ossia al file "doc2_4_2018-12-30T09_42_06".

Prosegue alla pagina successiva

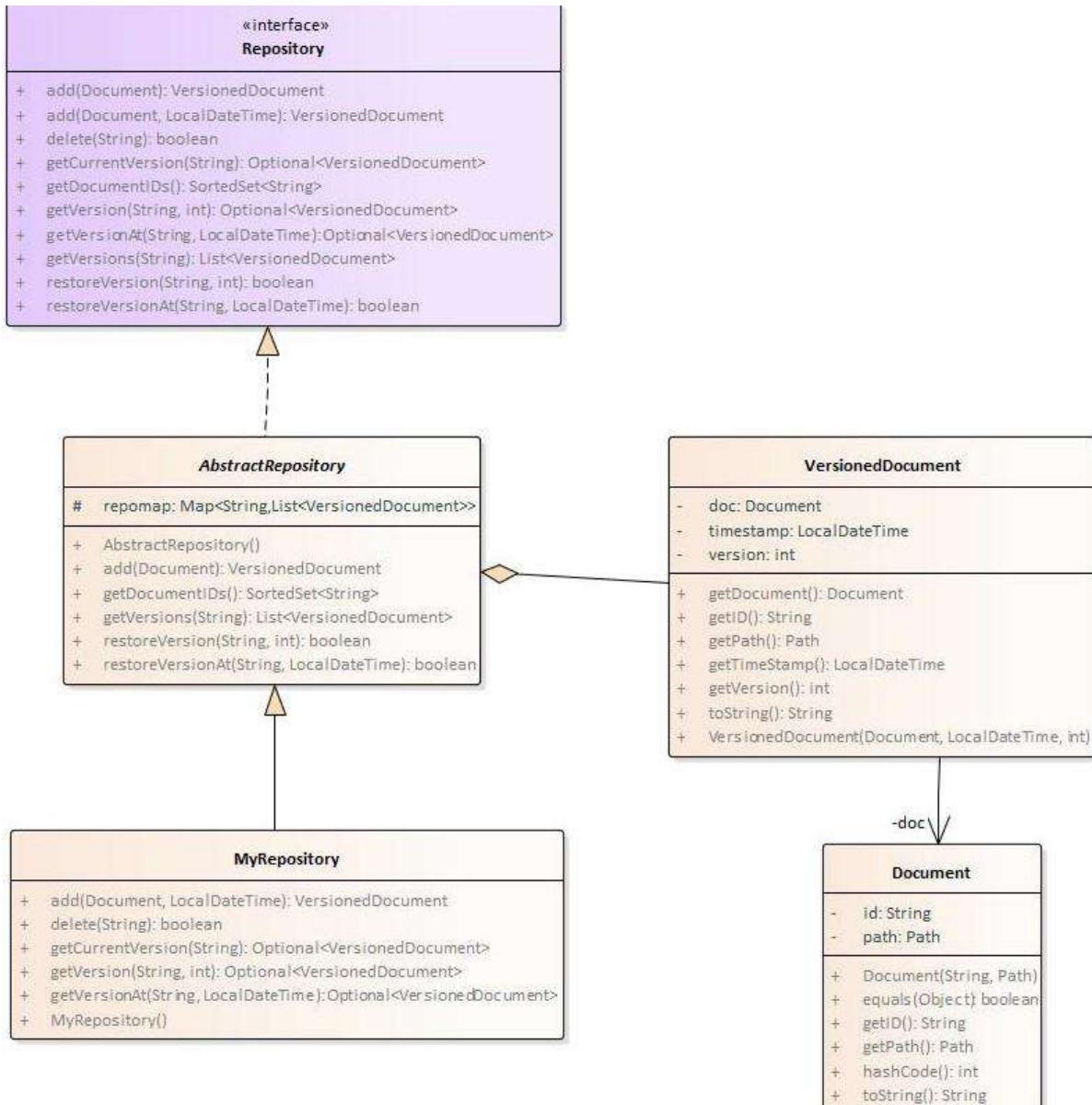
Parte 1

(punti: 20)

Dati (namespace saferepo.model)

(punti: 13)

Il modello dei dati deve essere organizzato secondo il diagramma UML di seguito riportato:



SEMANTICA:

- la classe **Document** (fornita) rappresenta un documento, caratterizzato da identificativo univoco (stringa) e percorso (Path).
- la classe **VersionedDocument** (fornita) rappresenta un documento versionato: incapsula un documento, ed è caratterizzata anche da timestamp (LocalDateTime) e numero di versione (intero non negativo).
- l'interfaccia **Repository** (fornita) descrive l'interfaccia d'uso del repository dal punto di vista logico (non gestisce quindi alcun file: ciò è compito della persistenza descritta più oltre) e consente di:
 - recuperare la versione corrente o precedente di un documento, di cui sia noto l'identificativo univoco: il risultato è un **Optional<VersionedDocument>**;
 - ripristinare una versione precedente di un documento di cui sia noto l'identificativo univoco, specificando o il numero di versione, o il timestamp corrispondente all'istante di tempo che interessa;
 - aggiungere un documento (**Document**) al repository, eventualmente specificando il timestamp da associare (default: **now**); i metodi restituiscono il **VersionedDocument** del nuovo documento inserito;
 - eliminare (logicamente) dal repository un documento di cui sia noto l'identificativo univoco.

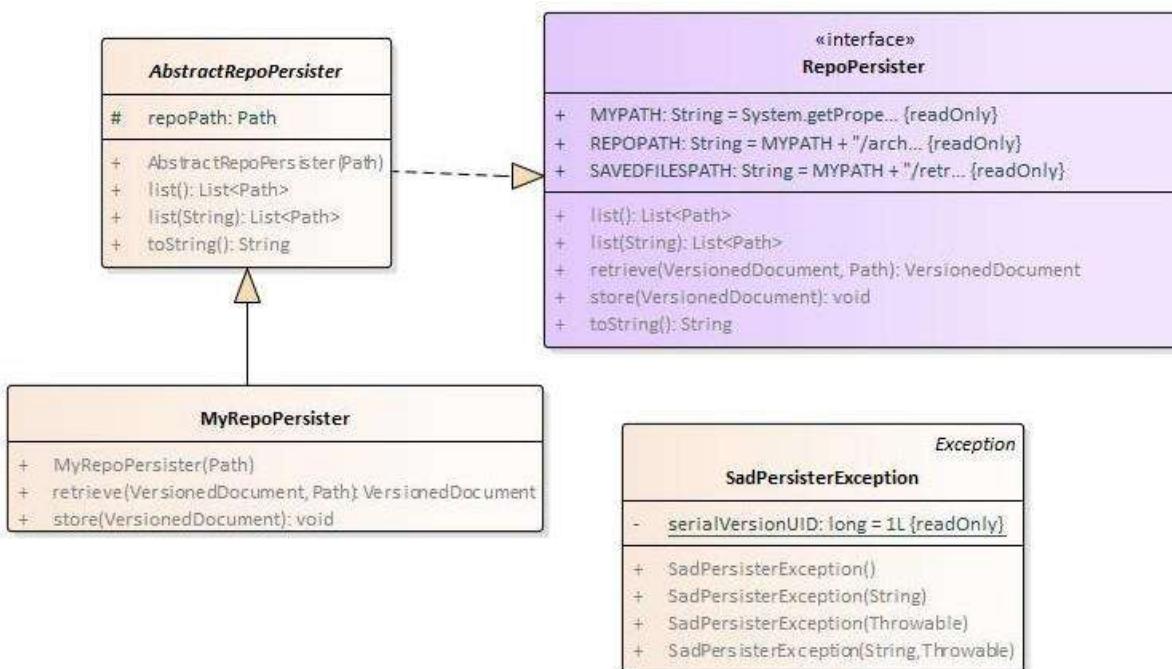
- d) la classe astratta **AbstractRepository** (fornita) implementa parzialmente **Repository**, definendo la struttura dati e i metodi che non catturano direttamente specifiche politiche – ovvero **getVersions**, **getDocumentIDs**, **add/1** (che è un banale shortcut per un caso particolare di **add/2**) - nonché **restoreVersion** e **restoreVersionAt** (in quanto non direttamente coinvolti nella gestione della successiva applicazione grafica).
- e) la classe **MyRepository** (da realizzare) completa l'implementazione di **Repository** estendendo la precedente classe **AbstractRepository**: a tal fine implementa i metodi **add**, **delete**, **getCurrentVersion**, **getVersion**, **getVersionAt** nel modo specificato in precedenza. Più precisamente, tutti i metodi che restituiscono una versione possono restituire un optional **empty** nel caso la versione richiesta non esista: in più, **getVersion** lancia **IllegalArgumentException** in caso sia invocata con un numero di versione negativo, incoerente con l'idea stessa di “numero di versione”.

NB: come da specifica, si ricorda che la mappa (*protected* in **AbstractRepository**) ha come chiave l'ID del documento e come valore una lista di **VersionedDocument**, ordinata per costruzione per *timestamp* crescente.

Persistenza (namespace saferepo.persistence)

(punti: 7)

Questo componente è la controparte fisica del **Repository** precedente: suo compito è gestire le copie dei file su disco con i relativi rename, tramite la coppia di metodi **store/retrieve** (NB: non tutte le funzionalità offerte da Repository a livello logico sono disponibili qui a livello fisico: in particolare, non viene offerto il servizio di ripristino di versioni precedenti, pertanto *non* sono utilizzati i metodi **restoreVersion** e **restoreVersionAt** del repository), secondo il diagramma UML di seguito riportato:



SEMANTICA:

- a) l'interfaccia **RepoPersister** (fornita) dichiara le costanti che definiscono le cartelle di lavoro:

- **MYPATH**, la cartella principale di lavoro del persister: la posizione di default è la home dell'utente
- **REPOPATH** e **SAVEDFILESPATH**, sottocartelle di **MYPATH**, corrispondono alla directory in cui il persister copia i file versionati (metodo **store**) e a quella in cui rende disponibili i file estratti (metodo **retrieve**)

e i metodi per la manipolazione dei **VersionedDocument**:

- **store** memorizza un **VersionedDocument** nella cartella interna del sistema *SafeRepo*, generando il nome di file e il percorso corrispondenti, ed effettuando la copia fisica del file, come da specifiche iniziali;
- **retrieve** recupera un **VersionedDocument** dalla cartella interna del sistema *SafeRepo*, generando il nome di file e il percorso corrispondenti ed effettuando la copia fisica del file da restituire all'utente: tale file viene collocato nella cartella specificata dall'utente come secondo argomento (Path);

- `list` restituisce la lista di tutti i file (Path) attualmente presenti nella cartella interna del sistema `SafeRepo`;
 - `list(documentID)` restituisce la lista dei soli file (Path) disponibili nel sistema `SafeRepo` corrispondenti al documento specificato;
 - `toString` restituisce una frase che specifica il numero di file attualmente presenti nella cartella interna del sistema `SafeRepo`.
- b) la classe astratta `AbstractRepoPersister` (fornita) implementa parzialmente `RepoPersister`, gestendo nel costruttore la cartella di lavoro e concretizzando i metodi `list` e `toString`. I metodi `list` incapsulano l'eventuale `IOException` in una `SadPersisterException` (fornita) con opportuno messaggio d'errore, mentre `toString` la cattura e restituisce "ERROR".
- c) la classe **MyRepoPersister (da realizzare)** completa l'implementazione di `RepoPersister` estendendo la classe `AbstractRepoPersister`: a tal fine essa implementa i due metodi `store` e `retrieve` effettuando le copie dei file come da specifica, gestendo opportunamente i nomi dei percorsi. Anche questi metodi devono incapsulare l'eventuale `IOException` in una `SadPersisterException`. NB: è necessario evitare di cablare nel codice i percorsi delle cartelle di lavoro, utilizzando sempre le costanti definite nell'interfaccia `RepoPersister`

SUGGERIMENTO 1: ricordare i metodi di utilità della classe `Files` (in particolare `copy`) e della classe `Path` (in particolare `resolve` per comporre percorsi). Per copiare fisicamente un file, si suggerisce la sintassi:

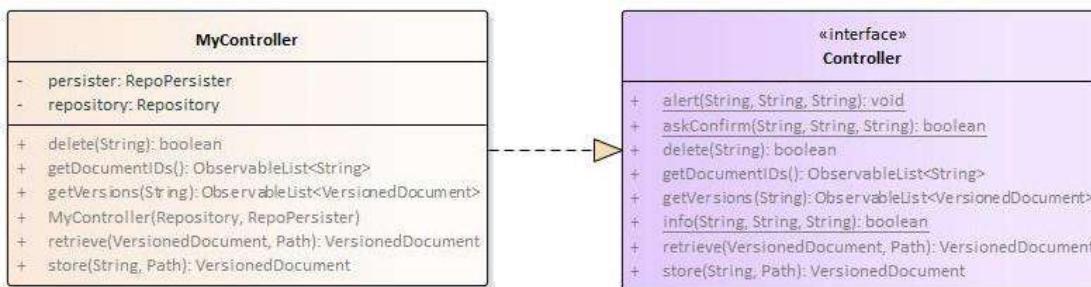
```
Path copiedFile = Files.copy(pathSorgente, pathDestinazione, StandardCopyOption.REPLACE_EXISTING);
```

Parte 2

(punti: 10)

Controller (namespace `saferepo.controller`)

Il controller – articolato in interfaccia e implementazione – è fornito già pronto: i suoi metodi fanno da ponte verso quelli opportuni del repository e del persister. L'interfaccia `Controller` offre altresì tre metodi statici `alert`, `askConfirm` e `info` per far comparire le finestre di dialogo utili a segnalare errori (Fig.10), chiedere conferma dell'intenzione dell'utente (Fig. 8) o fornire informazioni (Fig. 7).



GUI (namespace `saferepo.ui`)

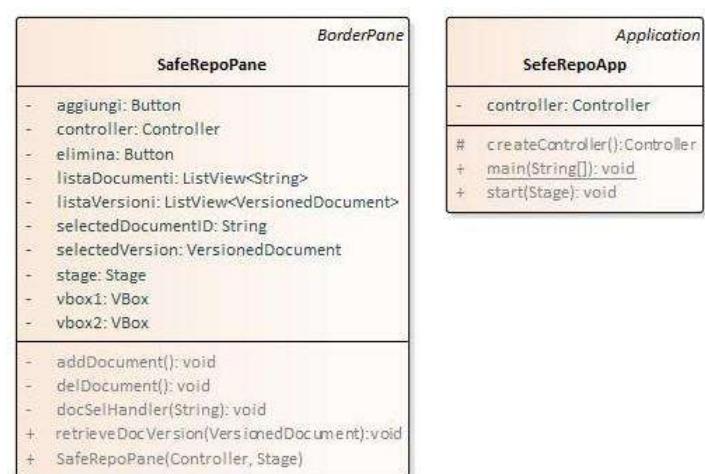
(punti: 10)

L'interfaccia grafica dev'essere simile (non necessariamente identica) a quella sotto illustrata.

La classe **SafeRepoApp** (fornita) contiene il main di partenza dell'applicazione.

La classe **SafeRepoPane** (da realizzare) contiene il pannello principale, organizzato come segue:

- Due componenti `ListView`, inizialmente vuoti, mostrano rispettivamente i documenti disponibili nel repository e le rispettive versioni; nella riga sottostante, due pulsanti (`Aggiungi file / Elimina`



[file](#) – quest’ultimo inizialmente disabilitato) consentono le omonime operazioni (Fig. 1).

- L’utente può aggiungere un documento al sistema *SafeRepo* premendo il pulsante [Aggiungi file](#): compare allora un apposito **FileChooser** (Fig. 2) per scegliere i file da caricare. Confermando, l’elenco documenti viene aggiornato (Fig. 3) con il documento appena inserito. Continuando a inserire documenti, essi compaiono via via nell’elenco, in ordine alfabetico (Fig. 4).
- Cliccando su una voce dell’elenco documenti, la lista a fianco viene immediatamente popolata con le rispettive versioni (Fig. 5): cliccando su una di queste voci, il sistema offre la possibilità di scaricare il corrispondente documento, tramite un apposito **DirectoryChooser** (componente analogo al **FileChooser**, che consente di scegliere cartelle anziché file – Fig. 6): il titolo del chooser indica quale versione si sta scaricando (ad es. “Scaricamento versione #1 di doc2”). Se l’utente conferma, il corrispondente file viene copiato nella cartella di destinazione scelta, e viene mostrato un apposito messaggio di conferma (Fig. 7).
- Se, infine, l’utente desidera eliminare (logicamente) un documento, deve selezionarlo e premere il pulsante [Elimina documento](#): comparirà un apposito dialogo di richiesta conferma (Fig. 8), dopo il quale verrà effettuata la cancellazione, ossia verrà creata la nuova versione collegata al documento vuoto, come da specifica (Fig. 9). Nel caso si tenti di scaricare tale versione, un apposito messaggio d’errore (Fig. 10) informerà della situazione, specificando altresì che rimangono comunque disponibili tutte le versioni precedenti.

NB: evitare di cablare nel codice i percorsi delle cartelle di lavoro del persister: agire sempre tramite controller.

Per comodità, può convenire invece impostare i file/directory chooser in modo che si posizionino sulle cartelle predefinite nel progetto – in particolare sulla sottocartella “testfiles” nel caso del file chooser “aggiungi file”. Ciò può essere fatto chiamando il metodo `setInitialDirectory` con opportuni argomenti, ad esempio:

```
chooser.setInitialDirectory(Paths.get(System.getProperty("user.dir")).toFile());
```

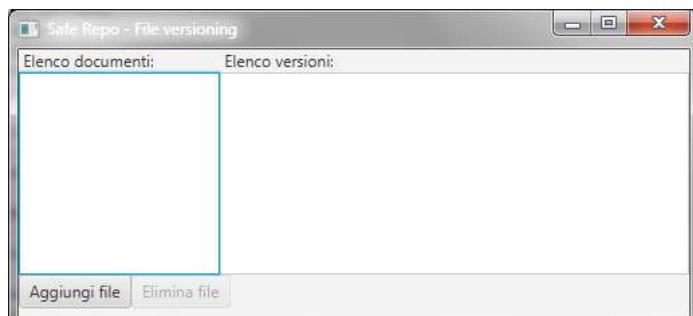


Figura 1

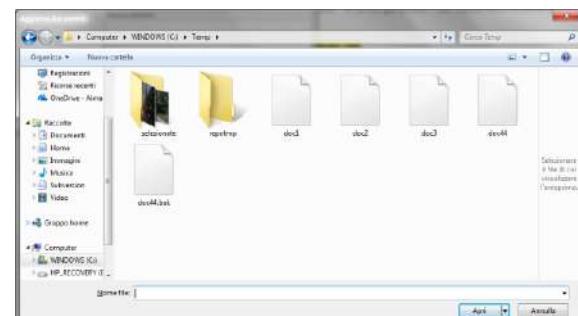


Figura 2

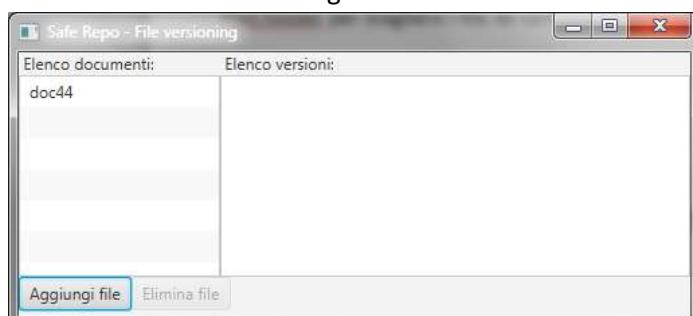


Figura 3

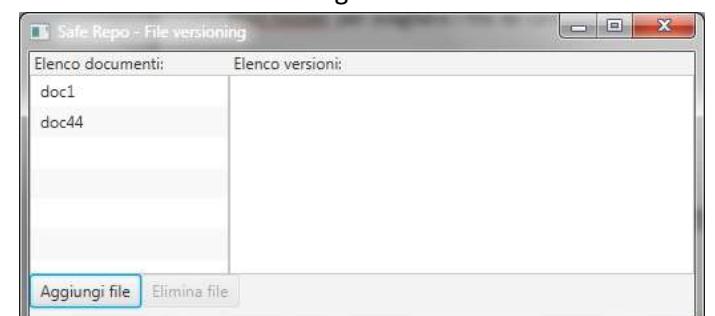


Figura 4

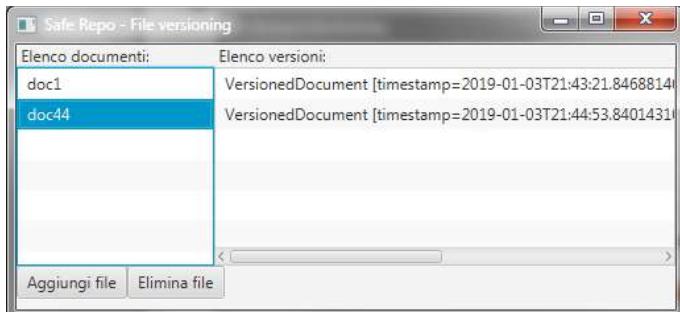


Figura 5

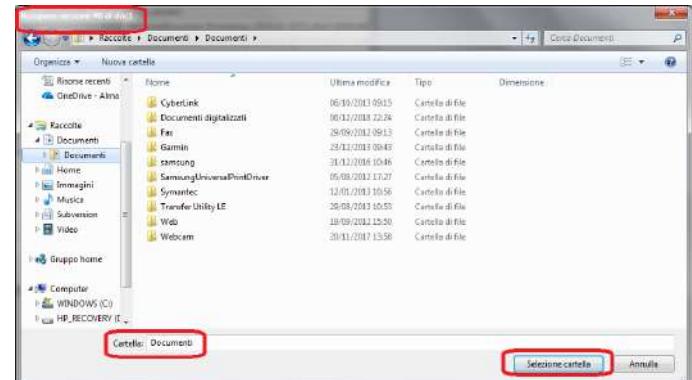


Figura 6

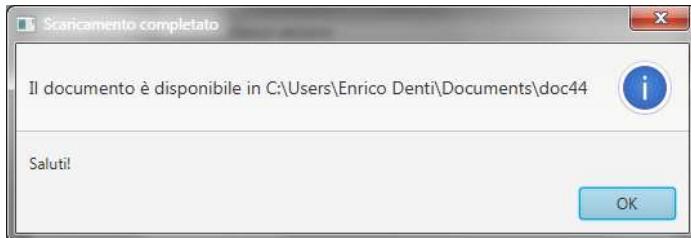


Figura 7

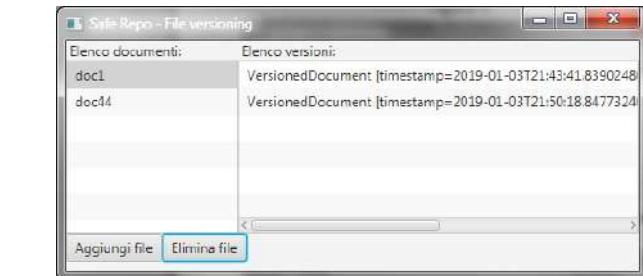


Figura 9



Figura 8



Figura 10

I tre metodi statici **Controller.alert**, **Controller.askConfirm** e **Controller.info** consentono di far comparire le finestre di dialogo rispettivamente per segnalare errori (Fig.10), chiedere conferma dell'intenzione dell'utente (Fig. 8) o fornire informazioni (Fig. 7).

ESAME DI FONDAMENTI DI INFORMATICA T-2 del 12/09/2018

Proff. E. Denti – R. Calegari – G. Zannoni

Tempo: 4 ore

NOME PROGETTO ECLIPSE: CognomeNome-matricola (es. RossiMario-0000123456)

NOME CARTELLA PROGETTO: CognomeNome-matricola (es. RossiMario-0000123456)

NOME ZIP DA CONSEGNARE: CognomeNome-matricola.zip (es. RossiMario-0000123456.zip)

NB: l'archivio ZIP da consegnare deve contenere l'intero progetto Eclipse

La società Rent-a-Bike ha richiesto lo sviluppo del software necessario a gestire il noleggio bici in diverse città.

DESCRIZIONE DEL DOMINIO DEL PROBLEMA

Il noleggio bici è una forma di mobilità ecosostenibile sempre più diffusa nelle grandi città, specialmente nelle forme che permettono la localizzazione, il prelievo e il rilascio delle bici ovunque, senza doversi recare in stazioni predefinite, grazie al GPS di cui ogni bici è dotata, unitamente a un'app per smartphone che ne consente la gestione.

La **tariffazione** è *a tempo*, secondo uno schema fisso, ma i cui costi variano da una città all'altra, che prevede:

- un primo periodo (solitamente di 20-30 minuti), per i noleggi brevi
- successivi periodi (solitamente di 30 minuti ciascuno) per i noleggi più lunghi
- una durata massima (solitamente 12-24 ore) o un orario massimo (solitamente mezzanotte) entro cui la bici va riconsegnata, pena il pagamento di multe salate.

Una **tariffa** è definita quindi per una *specifica città*, ed è caratterizzata da:

- il nome della città a cui si applica;
- il costo (in €cent) e la durata (in minuti) del primo periodo;
- il costo (in €cent) e la durata (in minuti) dei periodi successivi;
- la durata massima (in ore) OPPURE l'orario limite del noleggio.

Un noleggio che superi la durata massima o l'orario limite si dice *irregolare* e verrà sanzionato.

Un **noleggio** inizia quindi in un dato momento (giorno e ora) e termina in un altro momento (giorno e ora): ad esso corrisponde un **costo in euro** calcolato come segue:

1. si calcola la durata del noleggio
2. se ne verifica la regolarità (durata non superiore al massimo o orario limite rispettato)
3. se il noleggio è regolare, se ne calcola il costo applicando la tariffa, a scatti di durata pari ai periodi previsti in quella città (i minuti eventualmente non goduti nell'ultimo periodo vanno persi); altrimenti, si aggiunge anche la sanzione.

ESEMPI DI TARiffe

Bologna, 59 €cent per i primi 20', poi 99 €cent ogni 30'; max 12 ore; sanzione € 7

Reggio Emilia, 30 €cent per i primi 30', poi 50 €cent ogni 30'; max entro 23:59; sanzione € 7

ESEMPI DI NOLEGGI

BO, dalle 7.20 alle 7.45 → durata 25' (noleggio regolare) → € 0.59 + 0.99 = € 1.58

BO, dalle 8.25 alle 8.44 → durata 19' (noleggio regolare) → € 0.59

RE, dalle 7.30 alle 18.40 → durata 11h10 (noleggio regolare) → 23 periodi di 30' → € 11.30

BO, dalle 7.30 alle 19.40 → durata 12h10 (noleggio irregolare) → 1 periodo di 20' + 24 periodi di 30' + multa → € 31.85

Il file di testo [CityRates.txt](#) specifica le tariffe delle varie città, nel formato dettagliato più oltre.

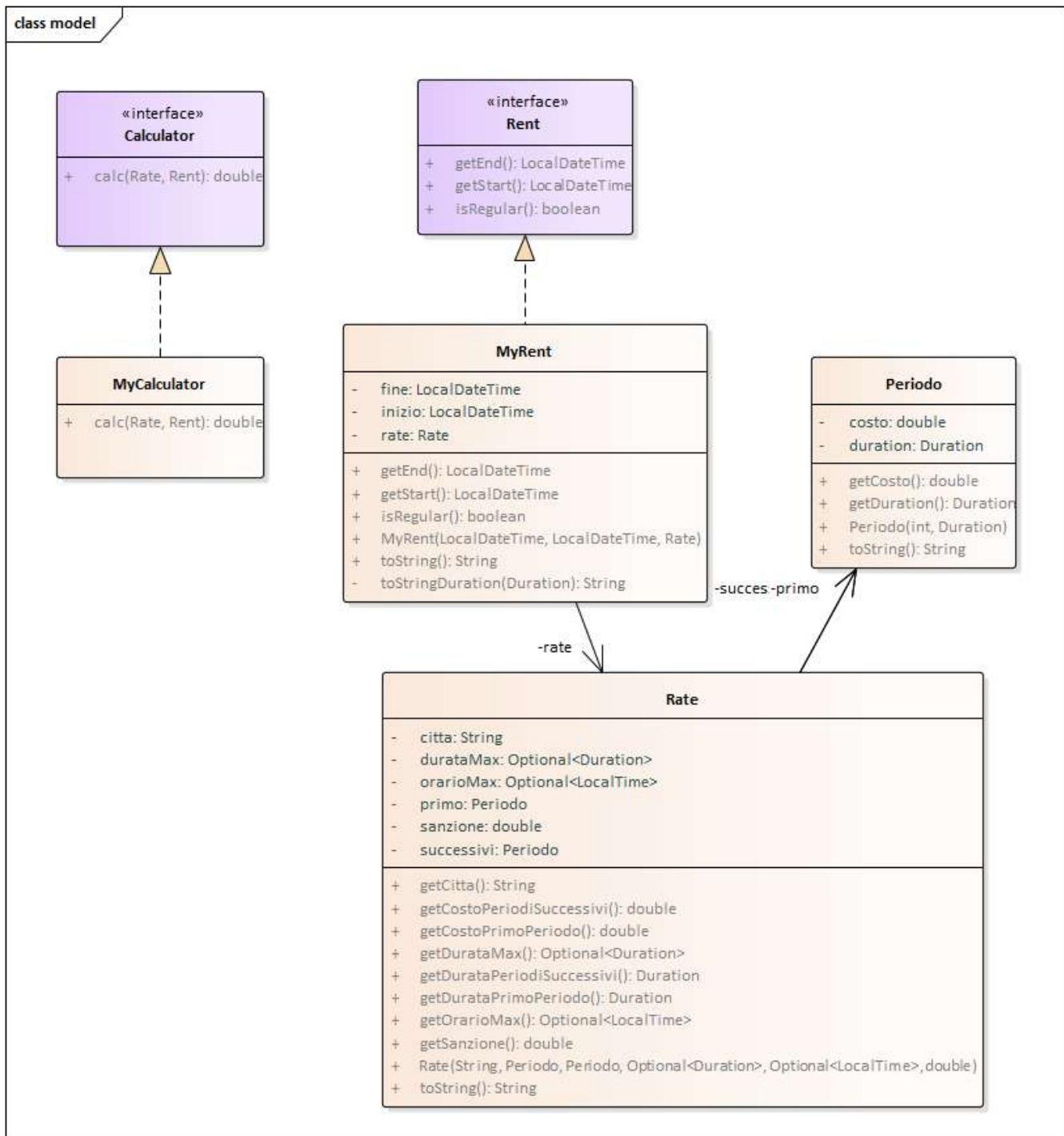
Parte 1

(punti: 19)

Dati (namespace rentabike.model)

(punti: 9)

Il modello dei dati deve essere organizzato secondo il diagramma UML di seguito riportato:



SEMANTICA:

- la classe **Periodo** (fornita) rappresenta un periodo, caratterizzato da costo (cent) e durata (in minuti).
- la classe **Rate** (fornita) espone i metodi che caratterizzano le proprietà di una tariffa: nome città, proprietà del primo periodo (costo e durata), proprietà dei periodi successivi (costo e durata), eventuale durata max e orario di rientro limite (argomenti **Optional**), ammontare della sanzione.
- l'interfaccia **Rent** (fornita) rappresenta un noleggio con le sue caratteristiche (inizio, fine, tariffa applicabile). Il metodo **isIrregular** consente di discriminare i noleggi regolari da quelli irregolari, mentre gli accessori recuperano il momento iniziale e finale del noleggio.

- d) la classe **MyRent (da realizzare)** implementa **Rent** opportunamente: il costruttore riceve i dati nel formato specificato dal diagramma UML.
- e) l'interfaccia **Calculator** (fornita) dichiara il metodo **calc** che effettua il calcolo del costo di un noleggio (**Rent**) ricevuto come secondo argomento sulla base della **Rate** ricevuta come primo argomento;
- f) la classe **MyCalculator (da realizzare)** concretizza **Calculator** implementando **calc** coerentemente a quanto specificato nel Dominio del problema.

Persistenza (namespace `rentabike.persistence`)

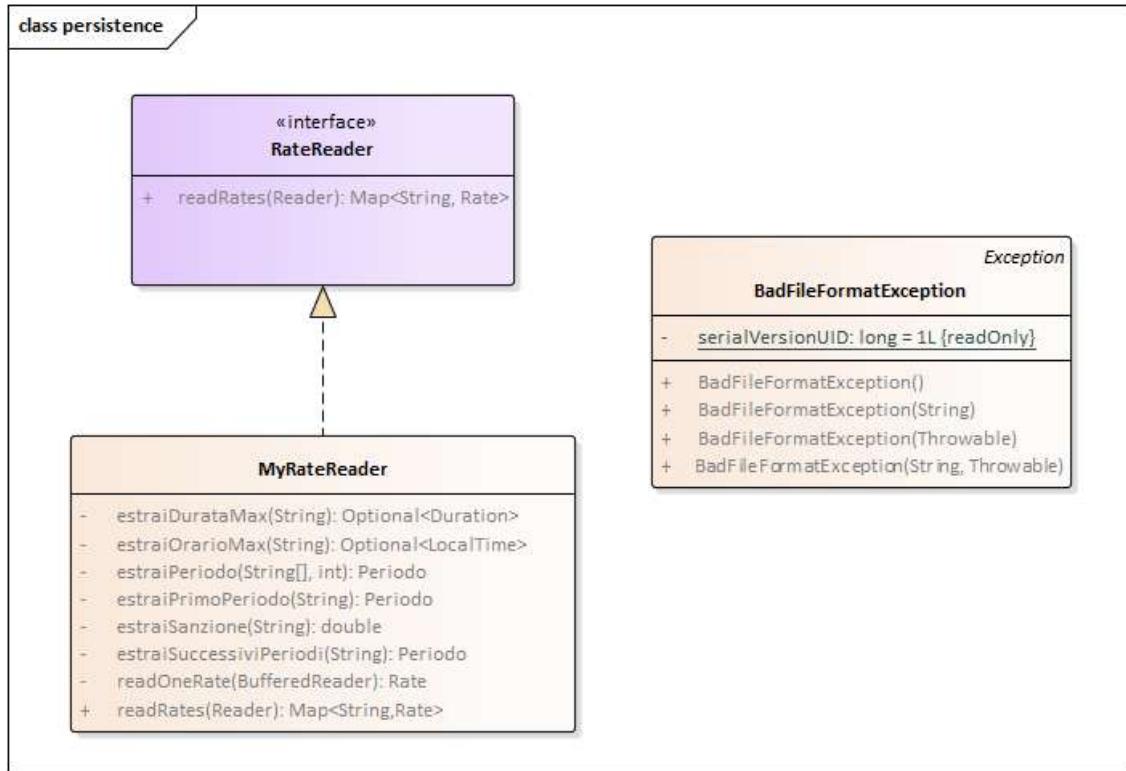
(punti: 10)

Come anticipato, il file `CityRates.txt` specifica le tariffe delle varie città, una per riga. Ogni riga contiene una serie di elementi separati da virgole, ulteriormente organizzati al proprio interno in sequenze di elementi separate da spazi. Più precisamente, nell'ordine si ha:

- nome della città (può contenere spazi)
- il costo e la durata del periodo iniziale
 - costo del periodo iniziale (valore intero seguito dalla parola chiave “**cent**”)
 - la parola chiave “**per**”
 - un valore intero
 - la parola chiave “**minuti**”
- il costo e la durata dei periodi successivi
 - la parola chiave “**poi**”
 - costo (in €cent) del periodo (valore intero seguito dalla parola chiave “**cent**”)
 - la parola chiave “**per**”
 - un valore intero
 - la parola chiave “**minuti**”
- la durata massima o l'orario di rientro massimo
 - la parola chiave “**max**”
 - o un numero intero, seguito dalla parola chiave “**ore**”
oppure la parola chiave “**entro**” seguito da un orario nel formato classico “hh:mm”
- la sanzione
 - la parola chiave “**sanzione**”
 - costo (in €) della multa (valore reale seguito dalla parola chiave “**euro**”)

ESEMPIO DEL FILE `CityRates.txt`

Bologna, 59 cent per 20 minuti, poi 99 cent per 30 minuti, max 12 ore, sanzione 7.50 euro
Reggio Emilia, 30 cent per 30 minuti, poi 50 cent per 30 minuti, max entro 23:59, sanzione 7 euro



L'interfaccia **RateReader** (fornita) dichiara il metodo `readRates` che, dato un **Reader**, legge le tariffe dal file e le restituisce sotto forma di mappa `Map<String, Rate>` indicizzata per nome città.

La classe **MyRateReader (da realizzare)** implementa tale interfaccia effettuando i necessari controlli sul formato del file, lanciando **BadFormatException** (fornita) in caso di errori di formato con messaggio dettagliato sulla specifica parola chiave mancante o elemento errato, o propagando **IOException** in caso di errori di lettura con specifico messaggio d'errore.

Parte 2

(punti: 11)

Controller (namespace `rentabike.ui.controller`)

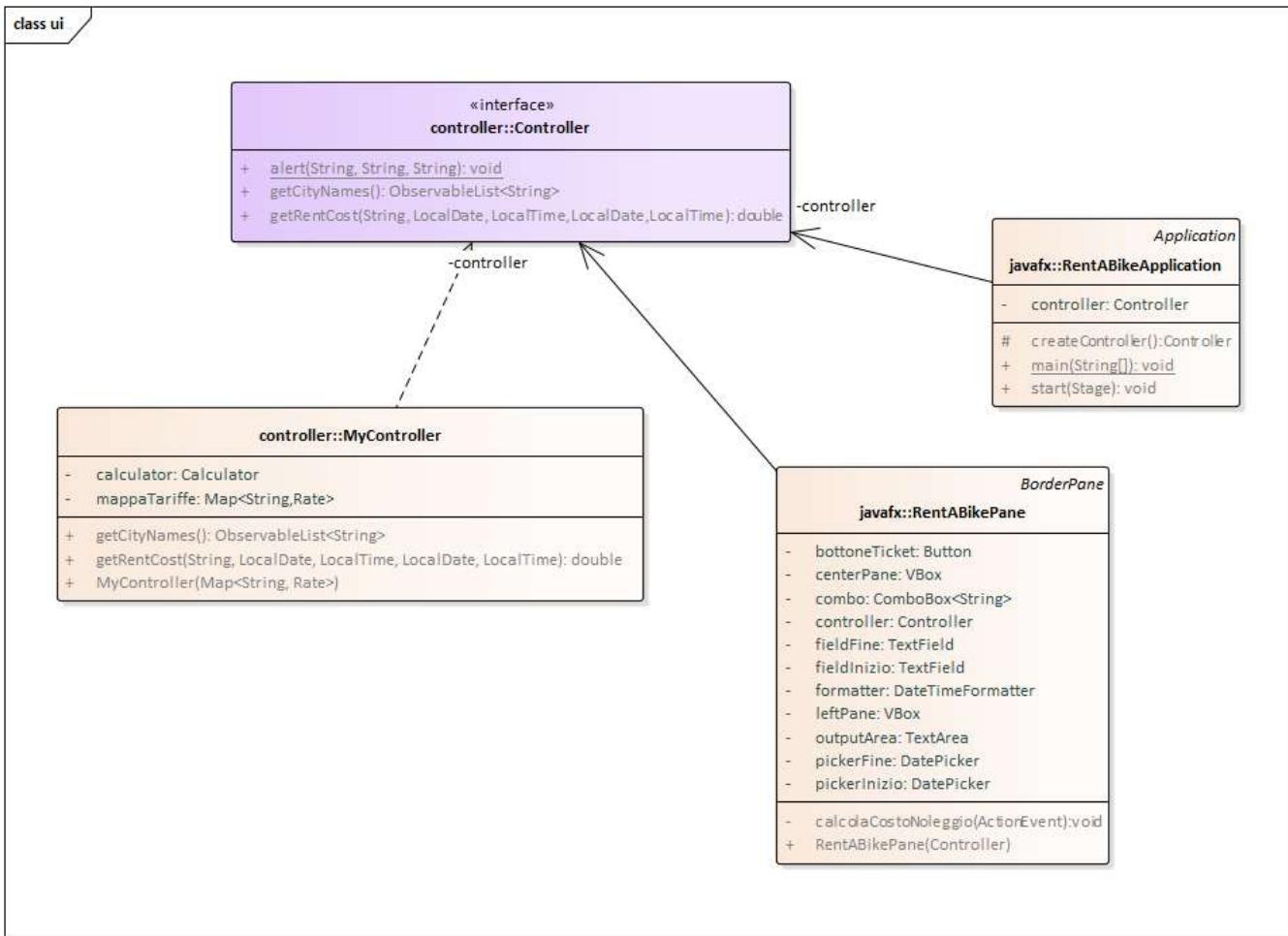
(punti: 3)

L'interfaccia **Controller** (fornita) dichiara il metodo `getCityNames`, che restituisce la *lista osservabile* delle città per cui sono definite le tariffe, e il metodo `getRentCost` che calcola il costo del noleggio. Quest'ultimo riceve come argomenti:

- il nome della città;
- la data (un **LocalDate**) e l'ora (un **LocalTime**) di inizio sosta
- la data (un **LocalDate**) e l'ora (un **LocalTime**) di fine sosta

È fornito inoltre il metodo statico `alert` per far comparire una finestra di dialogo che segnali errori: i tre argomenti rappresentano il titolo della finestra, l'header e il testo del messaggio (Figg. 3 e 4).

La classe **MyController (da realizzare)** deve implementare **Controller** senza effettuare verifiche sugli argomenti, che si suppongono validati dal chiamante.



Interfaccia utente (namespace rentabike.ui.javafx)

(punti: 8)

L'interfaccia utente deve essere simile (non necessariamente identica) all'esempio mostrato nelle figure seguenti.

La classe BikeRentPane (da realizzare), che estende **BorderPane**, deve prevedere:

- in alto, una combo box per la scelta della città, con a fianco, un pulsante a sfondo rosso “*Calcola costo*” per calcolare il costo del noleggio;
- a sinistra, due **DatePicker** e due campi di testo, per inserire data e orario di inizio e fine noleggio;
- a destra, un'area di testo in cui mostrare il costo del noleggio, compreso dell'eventuale sanzione. **Il costo deve essere formattato in Euro tramite esplicito del Locale.ITALY** (NB: Il simbolo della valuta può precedere o seguire l'importo, secondo le convenzioni della versione Java utilizzata).

È altresì compito del pannello effettuare un'accurata verifica dell'input prima di invocare il controller, in particolare:

- verificando che la città selezionata non sia nulla;
- verificando che la data di fine noleggio non sia antecedente la data di inizio noleggio;
- verificando che l'orario di fine noleggio non sia antecedente l'orario di inizio noleggio.

In tali casi l'applicazione deve mostrare opportuni dialoghi, tramite il metodo statico **Controller.alert** (Figg. 3 e 4).

BikeRent - Move yourself!

Città: Bologna

Inizio noleggio: 07/08/2018 10:37

Fine noleggio: 07/08/2018 10:37

Calcolo costo

Costo del noleggio 1,58 €

Figura 1

BikeRent - Move yourself!

Città: Bologna

Inizio noleggio: 07/08/2018 10:20

Fine noleggio: 07/08/2018 07:45

Calcolo costo

Costo del noleggio 1,58 €

Figura 2

BikeRent - Move yourself!

Città: Bologna

Inizio noleggio: 07/08/2018 07:20

Fine noleggio: 07/08/2018 7:45

Calcolo costo

Erre di formato

Errore nel formato orario

L'orario di inizio e fine noleggio deve avere la forma HH:MM

OK

Figura 3

BikeRent - Move yourself!

Città: Bologna

Inizio noleggio: 07/09/2018 10:51

Fine noleggio: 07/08/2018 10:51

Calcolo costo

Erre

Impossibile tornare indietro nel tempo

La fine del noleggio precede l'inizio

OK

Figura 4

ESAME DI FONDAMENTI DI INFORMATICA T-2 del 23/07/2018

Proff. E. Denti – R. Calegari – G. Zannoni

Tempo: 4 ore

NOME PROGETTO ECLIPSE: CognomeNome-matricola (es. RossiMario-0000123456)

NOME CARTELLA PROGETTO: CognomeNome-matricola (es. RossiMario-0000123456)

NOME ZIP DA CONSEGNARE: CognomeNome-matricola.zip (es. RossiMario-0000123456.zip)

NB: l'archivio ZIP da consegnare deve contenere l'intero progetto Eclipse

La società Park-O-Mat deve predisporre il software di un innovativo tipo di parcometro *smart*.

DESCRIZIONE DEL DOMINIO DEL PROBLEMA

Il parcometro regola la sosta nelle zone a pagamento (“strisce blu”), secondo una tariffazione a fasce orarie.

Una **fascia oraria** esprime un intervallo di tempo all'interno di una singola giornata ed è caratterizzata da:

- il giorno della settimana a cui si riferisce (es. Martedì);
- l' orario iniziale e finale (es. dalle 8 alle 13, intendendosi l'intervallo chiuso a sinistra e aperto a destra);
- l'essere *a pagamento* o *gratuita* (queste ultime corrispondono tipicamente a orari notturni o festivi).

Una **tariffa** è definita da un insieme di fasce orarie che definiscono quando la sosta si paga, ed è caratterizzata da:

- un nome univoco;
- il costo orario (in €/ora);
- un insieme di **fasce orarie** in cui il pagamento è attivo (es. Lunedì 8-20, Martedì 8-13 e 15-20, etc.);
- un periodo di *franchigia* iniziale (in minuti), in cui la sosta non viene comunque fatta pagare (può essere zero);
- un periodo *minimo* di sosta tariffata (in minuti), che determina il pagamento minimo effettuabile (può essere zero).

Una tariffa non è quindi tenuta a specificare esplicitamente anche le fasce gratuite, pur potendolo fare.

Una **tariffa valida** è invece una tariffa che copre tutti i giorni della settimana, 24h/24, senza “buchi”, specificando quindi non solo tutte le fasce orarie a pagamento, ma anche tutte le fasce orarie gratuite.

Per **ticket di sosta** si intende un pagamento riferito a una sosta che inizia in un dato momento (giorno e ora) e termina in un altro momento (giorno e ora): ad essa corrisponde un **costo totale** calcolato come segue (**don't panic!** ☺):

1. si pospone l'orario effettivo di inizio sosta di una quantità pari al periodo di franchigia: ad esempio, una sosta che inizi alle 7.30 in una zona dove la franchigia è di 1h si considera iniziare alle 8.30, ai fini del calcolo del costo.
2. si calcola la durata della sosta e, se risulta inferiore al periodo minimo, si tariffa comunque il minimo: ad esempio, se la sosta effettiva è di 45 minuti ma il minimo tariffabile è 1h, viene tariffata comunque 1h.
3. si sommano i costi relativi alle fasce orarie e giorni coperti dalla sosta, considerando naturalmente solo i periodi a pagamento ed escludendo i periodi gratuiti: ad esempio, una sosta che inizi effettivamente alle 16 di un giorno e termini alle 12 di un altro, in una zona in cui si paghi dalle 8 alle 20, dovrà tariffare il periodo 16-20 del primo giorno, escludere il periodo 20-24 dello stesso giorno, far pagare il periodo 8-20 di tutti i giorni intermedi escludendo i corrispondenti periodi 0-8 e 20-24, e far pagare infine il periodo 8-12 dell'ultimo giorno, escludendo il periodo 0-8.

ESEMPI DI FASCE ORARIE

lunedì, 7-20, a pagamento

si intende quindi che si paga dalle 7:00 alle 19:59 comprese

giovedì, 0-7, sosta gratuita

si intende quindi che la sosta è gratuita dalle 0:00 alle 6:59 comprese

ESEMPI DI TARFFE (elencano solo le fasce a pagamento) [NB: questo non è il formato del file, è solo un esempio]

Tariffa H1 lun-dom 7-20 € 0.50/h, franchigia 0, minimo 60 minuti (7 fasce settimanali a pagamento)

Tariffa C lun-mar, gio-sab 8-13 e 15-20, € 0.50/h, franchigia 60 minuti, minimo 60 minuti
 mer 8-13 € 0.50/h, franchigia 60 minuti, minimo 60 minuti (13 fasce settimanali a pagamento)

ESEMPIO DI TARIFFE VALIDA (elenca anche le fasce gratuite così da coprire tutte la settimana 24h/24, 7gg/7)

Tariffa H1 valida lun-dom 0-7, sosta gratuita (7 fasce settimanali gratuite)

 lun-dom 7-20 € 0.50/h, franchigia 0, minimo 60 minuti (7 fasce settimanali a pagamento)

 lun-dom 20-24, sosta gratuita (7 fasce settimanali gratuite)

ESEMPI DI TICKET SOSTA

sosta in zona H1 martedì dalle 7.20 alle 8.45 → durata 1h25 (ok, supera il minimo) → € 0.50 x (1+25/60) = € 0.71

sosta in zona H1 giovedì dalle 7.20 alle 8.05 → durata 0h45 → minimo 1h → € 0.50

sosta in zona C il giovedì dalle 7.30 alle 15 → (franchigia 1h = inizio effettivo 8.30) → durata 4h30 (8:30-13) → € 2.25

sosta in zona C da lunedì 19.30 a martedì 16.00 → (franchigia 1h = inizio effettivo 20.30, paga dalle 8 di martedì)

→ tariffate 6h (dalle 8 alle 13 e dalle 15 alle 16) → € 3.00

sosta in zona C da martedì 19.30 a mercoledì 16.00 → (franchigia 1h = inizio effettivo 20.30, paga dalle 8 di mercoledì)

→ tariffate 5h (dalle 8 alle 13: mercoledì pomeriggio è gratis) → € 2.50

Il file di testo [Tariffe.txt](#) specifica le tariffe, nel formato dettagliato più oltre.

IMPORTANTE: per evitare problemi con Java9 in laboratorio, si suggerisce di far partire Eclipse attraverso lo script ausiliario StartEclipse.bat fornito nello start kit.

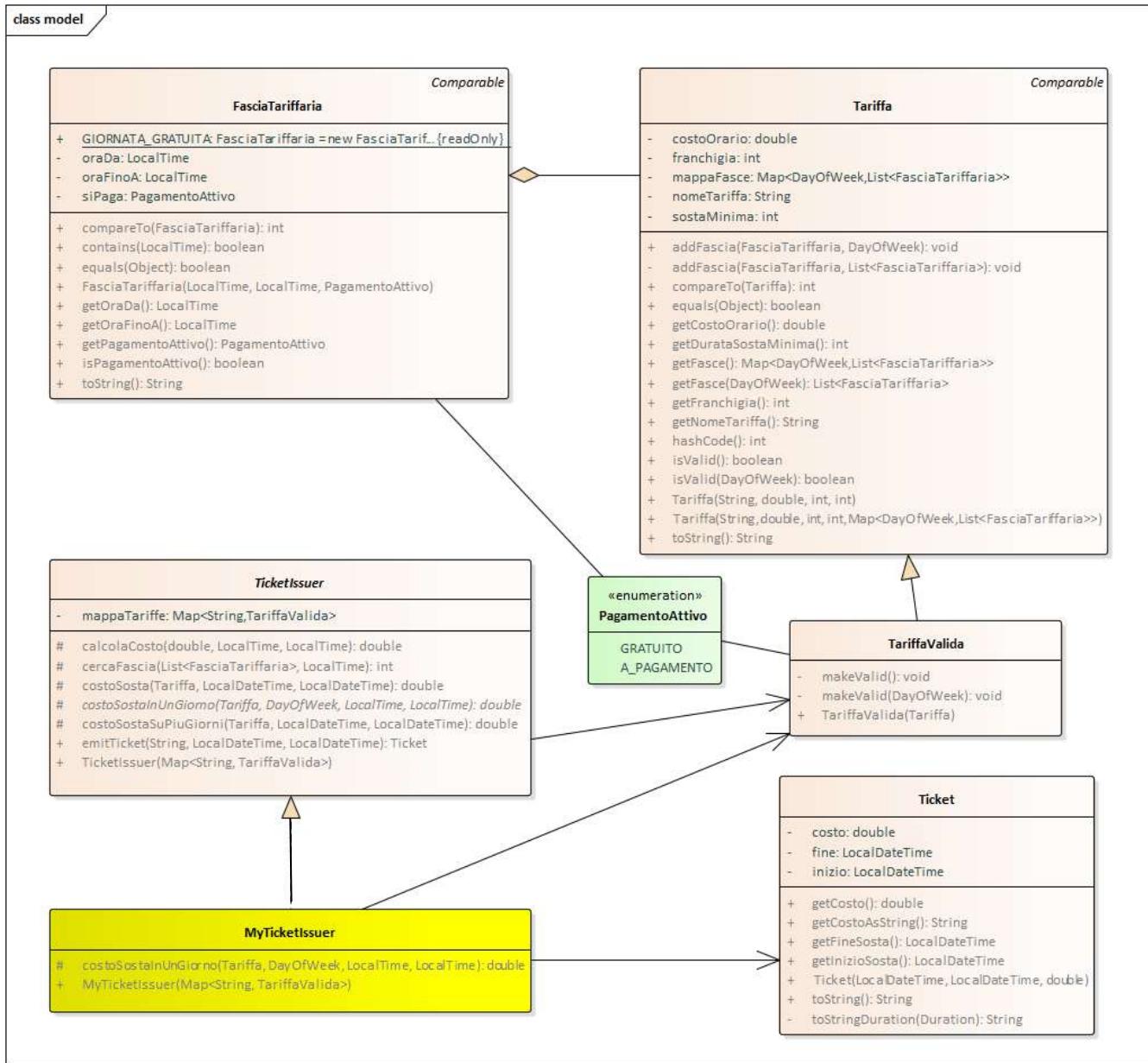
Parte 1

(punti: 20)

Dati (namespace parcomat.model)

(punti: 10)

Il modello dei dati deve essere organizzato secondo il diagramma UML di seguito riportato:



SEMANTICA:

- l'enumerativo di utilità `ItDayOfWeek` (fornito, ma non mostrato nell'UML sopra) definisce i giorni della settimana in italiano e fornisce due comodi metodi di conversione rispettivamente da/verso `java.time.DayOfWeek`;
- la classe **FasciaTariffaria** (fornita) rappresenta un intervallo di tempo (coppia di `java.time.LocalTime`) in un dato giorno della settimana, a pagamento o gratuita (caratteristica espressa dall'enumerativo **PagamentoAttivo**); è **Comparable** sull'orario di inizio fascia, così da permettere un facile ordinamento sequenziale; il metodo `equals` è ovviamente coerente. Oltre agli accessori, il metodo `contains` verifica se un orario appartenga alla fascia stessa.
- la classe **Tariffa** (fornita) è caratterizzata dal suo nome univoco, dal costo orario (€/ora), dalla durata della franchigia, dal periodo minimo tariffato e una o più **FasciaTariffaria**; per comodità di configurazione, al costruttore principale, che riceve tutti questi argomenti, si affianca un costruttore ausiliario che non riceve le fasce tariffarie, le quali possono essere aggiunte successivamente tramite il metodo `addFascia` che aggiunge una data fascia al giorno della settimana indicato, rispettandone l'ordine temporale. Il metodo `isValid`, in due versioni, permette di verificare se la tariffa sia *valida*, ossia copra interamente la giornata, rispettivamente in un dato giorno o nell'intera settimana [NB: di norma NON lo sarà, perché le tariffe solitamente NON specificano le fasce gratuite]

- d) la classe **TariffaValida** (fornita) incapsula una **Tariffa** rendendola automaticamente valida, ossia aggiungendo in automatico le fasce orarie gratuite necessarie a completare la copertura dell'intera settimana. Per una tariffa valida, quindi, il metodo **isValid** ha sempre esito positivo.
- e) la classe **Ticket** (fornita) rappresenta un ticket di sosta con le sue caratteristiche;
- f) la classe astratta **TicketIssuer** (fornita) emette il ticket per un dato periodo di sosta, tramite il metodo pubblico **emitTicket**; a tal fine si appoggia a un nutrito numero di metodi ausiliari (*protected*), descritti sotto.
Fra questi, il metodo astratto costoSostaInUnGiorno calcola l'importo della sosta in un dato giorno della settimana, dall'orario dato come primo argomento (incluso) all'orario dato come secondo argomento (escluso).
- il costruttore riceve una mappa con tutte le **TariffaValida** disponibili, indicizzata per nome;
 - il metodo pubblico **emitTicket** calcola il costo della sosta da un dato momento iniziale (un **LocalDateTime**) a un dato momento finale (un altro **LocalDateTime**) e ne emette il relativo ticket;
 - il metodo **costoSosta** calcola il costo della sosta dal momento iniziale (**LocalDateTime**) al momento finale (**LocalDateTime**), distinguendo il caso in cui la sosta inizi e finisce in un'unica giornata (metodo astratto **costoSostaInUnGiorno**) da quello in cui si estenda su più giorni (metodo **costoSostaSuPiùGiorni**);
 - il metodo di utilità **cercaFascia** cerca in una lista ordinata di fasce orarie (primo argomento) la fascia oraria in cui ricade il **LocalTime** fornito come secondo argomento;
 - il metodo di utilità **calcolaCosto**, dato il costo orario, calcola il costo della sosta fra due istanti (**LocalTime**) di una stessa giornata, gestendo anche il caso particolare in cui l'istante finale sia mezzanotte.
- g) la classe **MyTicketIssuer (da realizzare)** deve concretizzare **TicketIssuer** implementando **costoSostaInUnGiorno** in modo da riflettere lo specifico algoritmo di calcolo del costo descritto nel Dominio del problema (ricordando che le istanze di **FasciaTariffaria** di un giorno sono opportunamente ordinate all'interno della tariffa).

Persistenza (namespace `parcomat.persistence`)

(punti: 10)

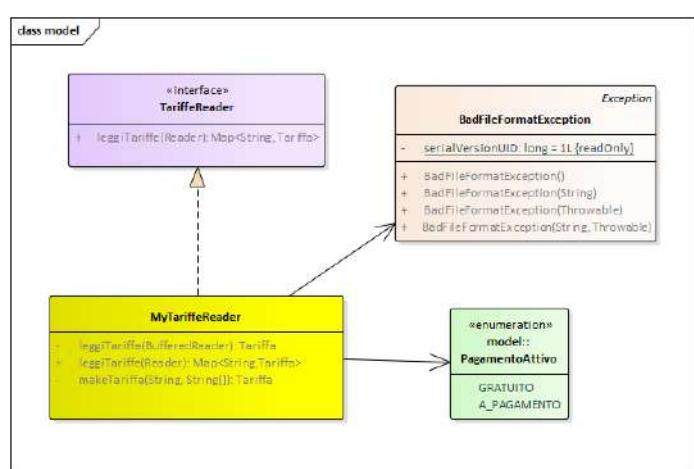
Come già anticipato, il file **Tariffe.txt** specifica le tariffe: ognuna occupa più righe, ciascuna delle quali descrive una possibile fascia tariffaria in un dato giorno della settimana. Il numero di righe che descrivono una tariffa è quindi variabile, dipendendo dalle fasce orarie e dai giorni della settimana coinvolti (v. esempio sotto). Più precisamente:

- la prima riga contiene la parola chiave “**Tariffa**” seguita dal nome della tariffa (che non contiene spazi);
- la seconda riga contiene nell’ordine costo, franchigia e minimo tariffario, così formattati:
 - il costo orario è composto dalla parola chiave “**costo**” seguita da un valore double;
 - la franchigia è composta dalla parola chiave “**franchigia**” seguita da un valore intero;
 - il minimo tariffabile è composto dalla parola chiave “**minimo**” seguita da un intero.
- le righe successive contengono l’elenco delle fasce orarie a pagamento, nel seguente formato:
 - il giorno della settimana, in italiano;
 - la fascia oraria, nel formato “**orainiziale-orafinale**”, dove **orainiziale** e **orafinale** sono interi.
- La riga finale contiene la frase “**Fine tariffa**” .

ESEMPIO DEL FILE Tariffe.txt

```

Tariffa C
costo 0.50, franchigia 60, minimo 60
Lunedì, 8-13
Lunedì, 15-20
Martedì, 8-13
Martedì, 15-20
Mercoledì, 8-13
Giovedì, 8-13
Giovedì, 15-20
Venerdì, 8-13
Venerdì, 15-20
Sabato, 8-13
Sabato, 15-20
Fine tariffa
  
```



```

Tariffa H1
costo 0.50, franchigia 0, minimo 60
Lunedì, 8-20
Martedì, 8-20
Mercoledì, 8-20
Giovedì, 8-20
Venerdì, 8-20
Sabato, 8-20
Fine tariffa

```

L'interfaccia **TariffeReader** (fornita) dichiara il metodo **leggiTariffe** che, dato un **Reader**, legge le tariffe dal file e le restituisce sotto forma di mappa **Map<String, Tariffa>** indicizzata per nome tariffa.

La classe MyTariffeReader (da realizzare) implementa tale interfaccia effettuando i necessari controlli sul formato del file, lanciando **BadFormatException** (fornita) in caso di errori di formato, o propagando **IOException** in caso di errori di lettura con specifico messaggio d'errore.

Parte 2

(punti: 10)

Controller (namespace `parcomat.ui.controller`)

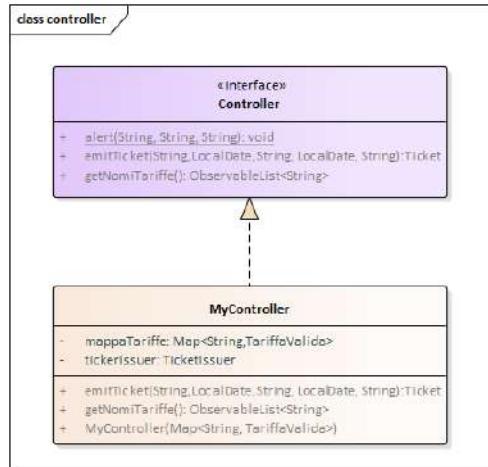
(punti: 5)

L'interfaccia **Controller** (fornita) dichiara il metodo **getNomiTariffe**, che restituisce la *lista osservabile* dei nomi delle tariffe disponibili, e il metodo **emitTicket** che emette il ticket di sosta. Quest'ultimo riceve come argomenti:

- la data di inizio sosta (un **LocalTime**)
- l'orario di inizio sosta (una stringa nel formato **hh:mm**)
- la data di fine sosta (un **LocalTime**)
- la data di inizio sosta (una stringa nel formato **hh:mm**)

È fornito inoltre il metodo statico alert per far comparire una finestra di dialogo che segnali errori: i tre argomenti rappresentano il titolo della finestra, l'header e il testo del messaggio (Figg. 5 e 6).

La **classe MyController (da realizzare)** deve implementare **Controller**, segnalando errore nel caso in cui l'istante finale preceda quello iniziale (Fig. 6).



Interfaccia utente (namespace `parcomat.ui.javafx`)

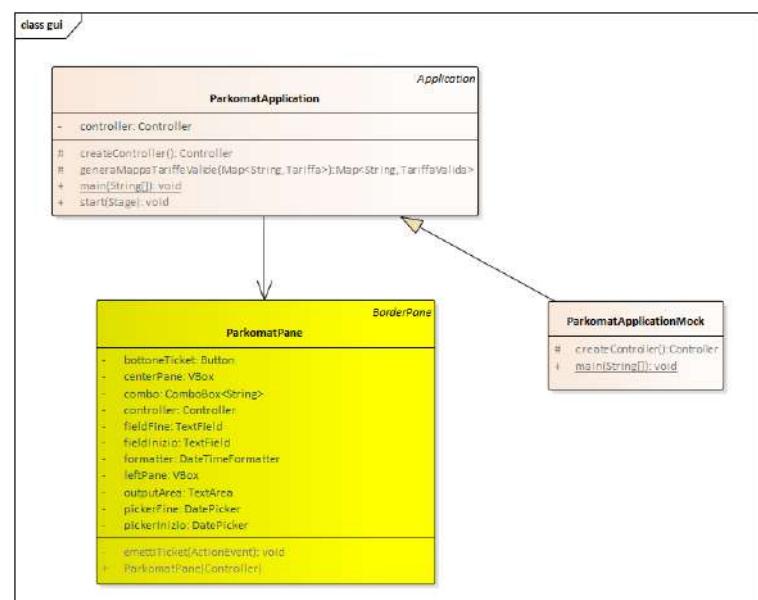
(punti: 5)

L'interfaccia utente deve essere simile (non necessariamente identica) all'esempio mostrato nelle figure seguenti.

La classe ParkomatPane (da realizzare), che estende **BorderPane**, deve prevedere:

- in alto, una combo box per la scelta delle tariffe disponibili;
- a sinistra, due **DatePicker** e due campi di testo, per inserire data e orario di inizio e fine sosta;
- a destra, un'area di testo in cui mostrare il ticket emesso;
- sotto, un pulsante *Emetti ticket* per emettere il ticket di sosta.

In caso di problemi (mancanza di file, errori di lettura o di formato, date fine sosta precedenti l'inizio sosta) l'applicazione deve mostrare opportuni dialoghi, tramite il metodo statico **Controller.alert** (Figg. 5 e 6).



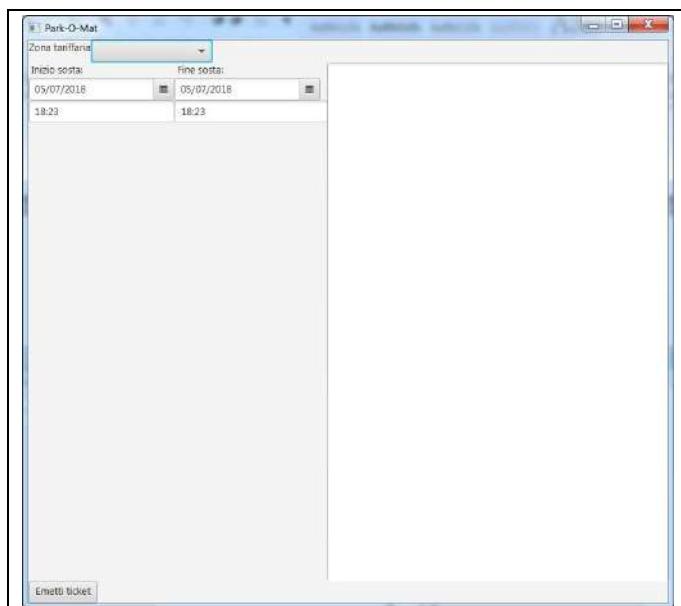


Figura 1

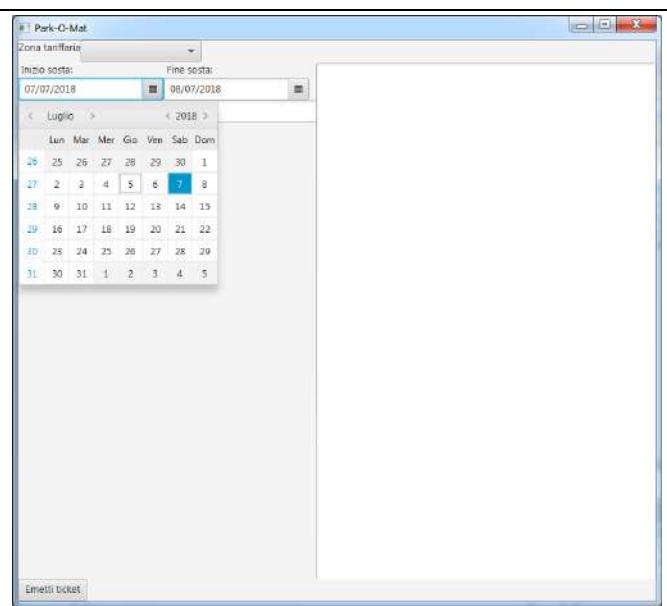


Figura 2

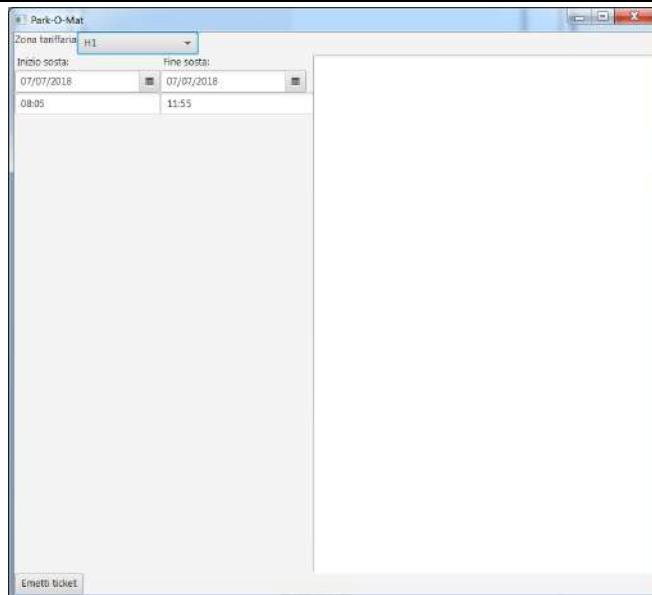


Figura 3

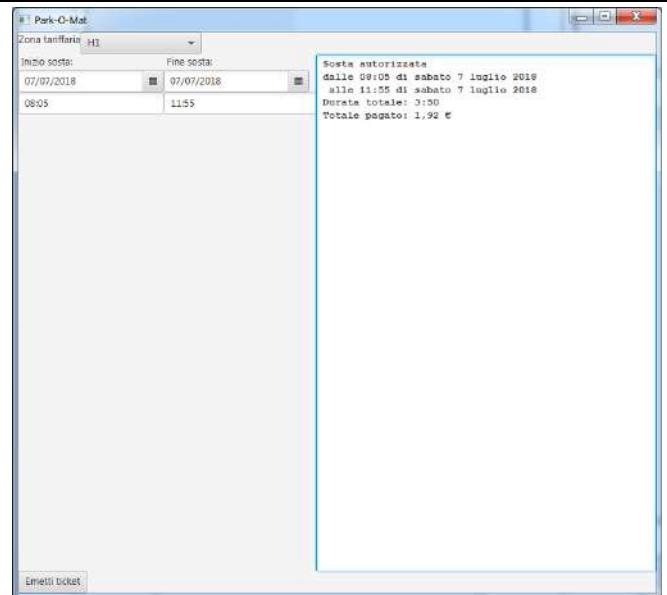


Figura 4

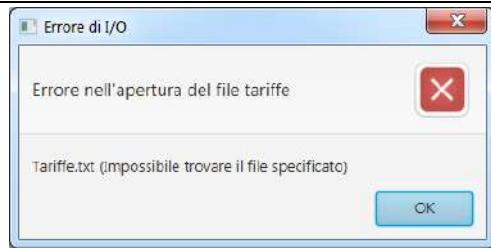


Figura 5

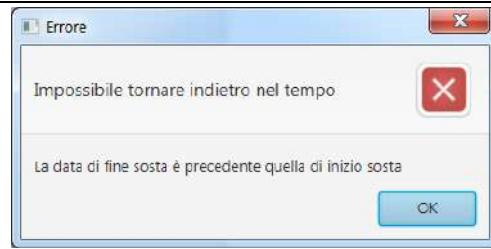


Figura 6

ESAME DI FONDAMENTI DI INFORMATICA T-2 del 04/07/2018

Proff. E. Denti – R. Calegari – G. Zannoni

Tempo: 4 ore

NOME PROGETTO ECLIPSE: CognomeNome-matricola (es. RossiMario-0000123456)

NOME CARTELLA PROGETTO: CognomeNome-matricola (es. RossiMario-0000123456)

NOME ZIP DA CONSEGNARE: CognomeNome-matricola.zip (es. RossiMario-0000123456.zip)

NB: l'archivio ZIP da consegnare deve contenere l'intero progetto Eclipse

La società *CupidOnLine* vuole offrire un servizio di ricerca intelligente dell'anima gemella: sulla base delle caratteristiche proprie e del partner desiderato, l'applicazione deve proporre una serie di candidati/e.

DESCRIZIONE DEL DOMINIO DEL PROBLEMA.

Ogni **persona** in archivio è caratterizzata da una serie di elementi:

- nickname (identificativo univoco)
- sesso
- data di nascita (utile per il calcolo dell'età e del segno zodiacale)
- caratteristiche fisiche (colore occhi, colore capelli, altezza, peso)
- luogo di residenza (città, provincia e regione)

Una **preferenza** è un insieme di caratteristiche che il partner dovrebbe soddisfare: alcune devono essere specificate per forza, altre invece sono opzionali. Più precisamente:

- sesso (obbligatorio)
- range di età (obbligatorio)
- segno zodiacale (opzionale)
- caratteristiche fisiche (colore occhi, colore capelli, altezza, peso) (opzionali)
- luogo di residenza (città, provincia, regione) (obbligatorio)

Una **corrispondenza** è una possibile coppia di persone sicuramente compatibili per sesso e caratterizzata da un **indice di compatibilità ≤ 100** calcolato sulla base della vicinanza fra la propria preferenza e le caratteristiche del potenziale partner, prendendo il **valore minimo** fra i seguenti sotto-indici:

- età: 100 se nel range, -5 per ogni anno fuori range (dal lato più vicino);
- luogo di residenza: 100 nella città, 90 nella provincia, 60 nella regione, 40 fuori regione;
- segno zodiacale: 100 se identico a quello richiesto o non specificato, 90 altrimenti;
- caratteristiche fisiche dimensionali: 100 se identico a quello richiesto o non specificato, altrimenti
-1 per ogni cm o kg di differenza;
- caratteristiche fisiche adimensionali: 100 se identico a quello richiesto o non specificato, 95 altrimenti.

L'insieme dei possibili partner dev'essere **ordinato in senso descendente per indice di compatibilità**, in modo da proporre per primi i partner più interessanti.

ESEMPIO

Persona : Roberto, 24 anni, Toro, capelli neri, occhi castani, 1.78, 61 kg, Bologna (BO, ER);

richiede partner donna , 20-25, Bilancia, capelli biondi, occhi -, 1.70, 58 kg, Bologna (BO, ER)

Candidate proposte:

- Anna, 22, Gemelli, cap. biondi, occhi azzurri, 1.70, 60, Imola (BO,ER)
- Ludovica, 26, Bilancia, cap. neri, occhi castani, 1.75, 51, Bologna (BO,ER)
- Elena, 19, Ariete, cap. neri, occhi neri, 1.65, 57, Modena (MO,ER)

Calcolo:	nickname	età	luogo	zod.	alt.	peso	capelli	MIN	classifica
	Anna	100	90	90	100	98	100	90	2°
	Ludovica	95	100	100	95	93	95	93	1°
	Elena	95	60	90	95	99	95	60	3°

I due file di testo [Iscritti.txt](#) e [Preferenze.txt](#) contengono rispettivamente l'elenco degli iscritti e delle preferenze, nel formato più oltre specificato.

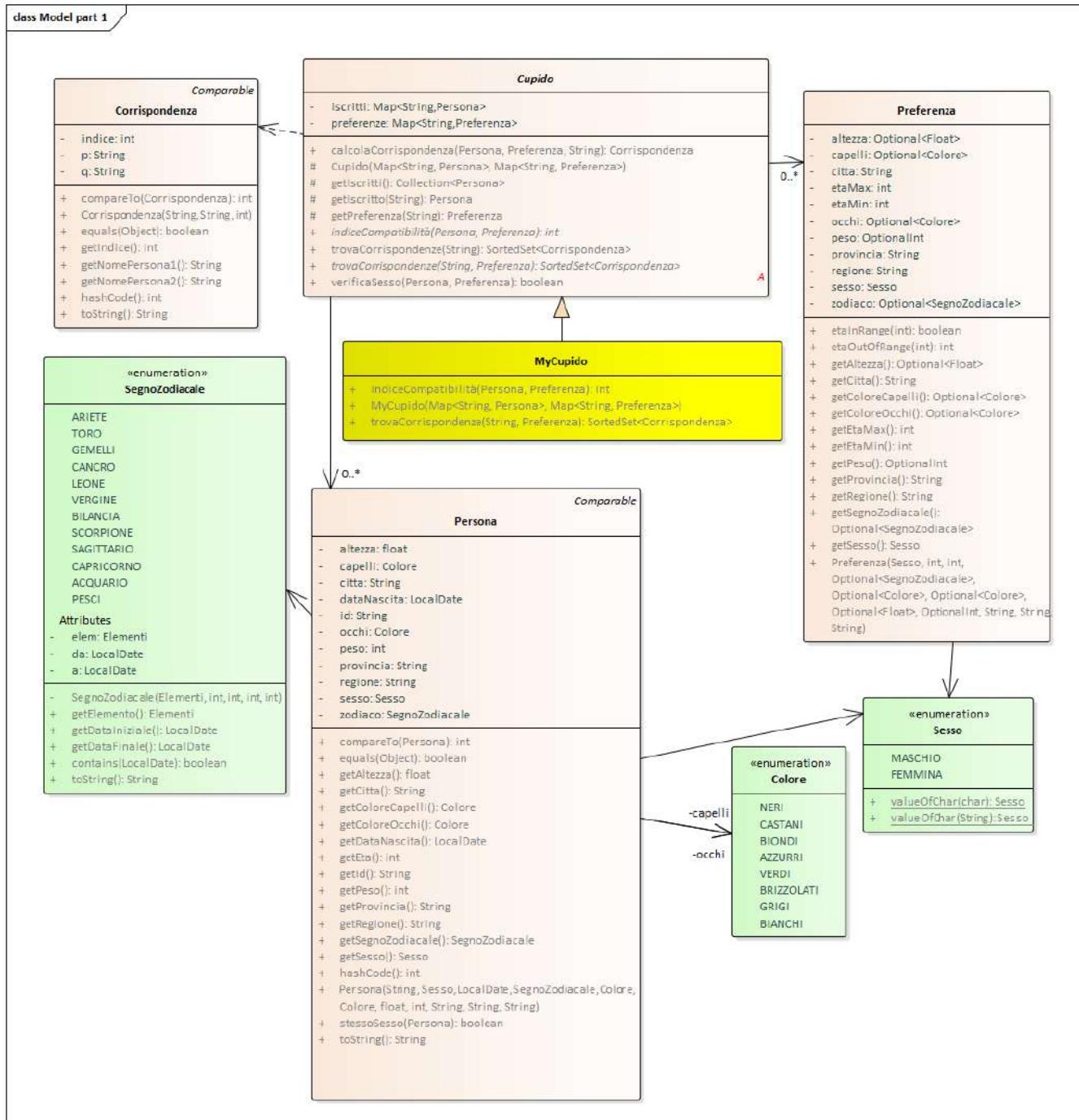
Parte 1

(punti: 21)

Dati (namespace cupidonline.model)

(punti: 12)

Il modello dei dati deve essere organizzato secondo il diagramma UML di seguito riportato:



SEMANTICA:

- l'enumerativo **Colore** (fornito) definisce i colori leciti per capelli e occhi;
- l'enumerativo **Sesso** (fornito) definisce i due sessi: i due metodi factory ausiliari `valueOfChar` restituiscono il valore dell'enumerativo corrispondente al carattere 'M' o 'F' (o stringa unitaria) passato come argomento;
- l'enumerativo **SegnoZodiacale** (fornito) definisce i dodici segni con le rispettive date di inizio e fine: il metodo ausiliario `contains` verifica se una certa data è compresa nel range di un dato segno zodiacale (NB: internamente viene definito e usato un ulteriore tipo enumerativo **Elemento** per esprimere l'elemento terra,acqua,aria,fuoco di ogni segno: ciò non traspare però all'esterno, quindi è irrilevante ai fini di questo compito);

- d) la classe **Persona** (fornita) rappresenta un iscritto dell'agenzia, con tutte le sue proprietà; in particolare il nickname (stringa) è garantito essere un identificativo univoco [quindi può essere usato come chiave nelle mappe..]; **Persona** è anche **Comparable** in senso alfabetico crescente per nickname;
- e) la classe **Preferenza** (fornita) rappresenta una preferenza, con tutte le sue proprietà, *molte delle quali opzionali*; oltre agli accessori, i metodi di utilità **etaInRange** e **etaOutOfRange** rispettivamente verificano se l'età data è nel range ammesso da quella preferenza, o di quanti anni sia lontana dal range ammesso dalla preferenza;
- f) la classe **Corrispondenza** (fornita) esprime la corrispondenza fra due persone, identificate dal loro nickname, con relativo indice di compatibilità; è **Comparable** in senso decrescente per indice di compatibilità;
- g) la classe astratta **Cupido** (fornita) incapsula i dati (iscritti e preferenze) e definisce la maggior parte dei metodi, lasciando astratti solo i due che dipendono da specifici algoritmi. Più esattamente:
- il costruttore riceve gli iscritti e le preferenze, sotto forma di mappe (prodotte dai reader) rispettivamente di tipo **Map<String, Persona>** per gli iscritti, e **Map<String, Preferenza>** per le preferenze; in entrambi i casi, la chiave è il nickname della persona;
 - il metodo **trovaCorrispondenze(String nickname)** trova le corrispondenze per l'iscritto identificato dal **nickname**, utilizzando la sua stessa preferenza: costituisce un caso particolare del metodo successivo (più generale): restituisce un insieme *ordinato in senso decrescente per indice di compatibilità*;
 - **il metodo *trovaCorrispondenze(String nickname, Preferenza pref)*** è astratto e descritto sotto al punto h); il suo obiettivo è cercare in generale corrispondenze fra un iscritto e una preferenza, in modo da gestire sia il caso in cui un iscritto cerchi un partner fra gli altri iscritti (e allora la preferenza sarà la sua), sia il caso in cui un *non iscritto* voglia cercare fra i già iscritti dell'agenzia (e allora la preferenza del non iscritto sarà stata ottenuta separatamente, magari compilando un questionario sull'interfaccia grafica)
 - il metodo **verificaSesso(Persona q, Preferenza pref)** verifica che la persona sia di sesso uguale a quello richiesto dalla preferenza, se quest'ultima non è nulla (altrimenti è comunque falso: non lancia eccezioni);
 - il metodo **calcolaCorrispondenza(Persona q, Preferenza pref, String nickname)** calcola la **Corrispondenza** fra la persona **q** e una preferenza **pref**, associando quest'ultima al **nickname** fornito: in questo modo è possibile calcolare corrispondenze anche fra un iscritto **q** e un non iscritto di cui sia nota la preferenza;
 - **il metodo *indiceCompatibilità(Persona q, Preferenza pref)*** è astratto e descritto sotto al punto h); il suo scopo è calcolare l'indice di compatibilità fra una persona e una preferenza, secondo un certo criterio.
- h) la classe **MyCupido (da realizzare)** estende **Cupido** implementando per la ricerca corrispondenze e il calcolo dell'indice di compatibilità gli specifici algoritmi descritti nel *Dominio del Problema*. Più precisamente:
- il costruttore, che riceve gli iscritti e le preferenze, delega interamente la costruzione alla classe base.
 - il metodo **trovaCorrispondenze(String nickname, Preferenza pref)** trova tutte le corrispondenze di una data preferenza **pref**, associandole poi nella **Corrispondenza** all'identificativo **nickname**: ciò rende possibile cercare corrispondenze anche per persone non iscritte, di cui sia nota la preferenza. Se la preferenza è nulla si intende che non esistono partner corrispondenti (tale aspetto è già verificato nel metodo **verificaSesso**). Il metodo restituisce un insieme *ordinato in senso decrescente per indice di compatibilità*.
 - il metodo **indiceCompatibilità(Persona q, Preferenza pref)** calcola l'indice di compatibilità, compreso fra 0 e 100, fra la persona **q** e la preferenza **pref**, secondo l'algoritmo descritto nel *Dominio del Problema*.

Prendere il *valore minimo* fra i seguenti sotto-indici:

- età: 100 se nel range, -5 per ogni anno fuori range (dal lato più vicino);
- luogo di residenza: 100 nella città, 90 nella provincia, 60 nella regione, 40 fuori regione;
- segno zodiacale: 100 se identico a quello richiesto o non specificato, 90 altrimenti;
- caratteristiche fisiche dimensionali: 100 se identico a quello richiesto o non specificato, altrimenti -1 per ogni cm o kg di differenza;
- caratteristiche fisiche adimensionali: 100 se identico a quello richiesto o non specificato, 95 altrimenti.

Come già anticipato, i due file *Iscritti.txt* e *Preferenze.txt* contengono rispettivamente l'elenco degli iscritti e delle preferenze, uno/a per riga: ogni riga contiene una serie di dati separati fra loro da virgole.

Sebbene il formato dei due file sia simile, i dati degli iscritti sono sempre completi, mentre le preferenze possono contenere una lineetta (“-”) al posto delle caratteristiche opzionali non specificate. In particolare:

- per gli iscritti si specificano identificativo univoco, sesso, data di nascita (formato ISO), colore capelli, colore occhi (preceduti da apposita parola chiave “capelli” o “occhi”, rispettivamente), altezza in metri (numero reale), peso in kg (numero intero), e infine città, provincia e regione di residenza;
- per le preferenze si specificano invece l’identificativo univoco della persona a cui la preferenza è associata e l’insieme delle caratteristiche richieste al potenziale partner, alcune delle quali però sono opzionali e quindi possono essere sostituite da una lineetta (vedere esempio sotto). Più precisamente:
 - **SONO SEMPRE PRESENTI** sesso, *range di età* (nel formato “nn-mm”, dove nn e mm rappresentano l’età minima e massima), città, provincia e regione di residenza;
 - **SONO INVECE OPZIONALI** segno zodiacale (non necessariamente tutto maiuscolo), colore capelli e occhi (la parola chiave “capelli” o “occhi” è però comunque presente), altezza, peso.

Da notare che sesso e range di età sono a inizio riga, mentre le informazioni sulla residenza sono in fondo alla riga: i dati opzionali sono quindi al centro fra questi.

ESEMPIO DEL FILE *Iscritti.txt*

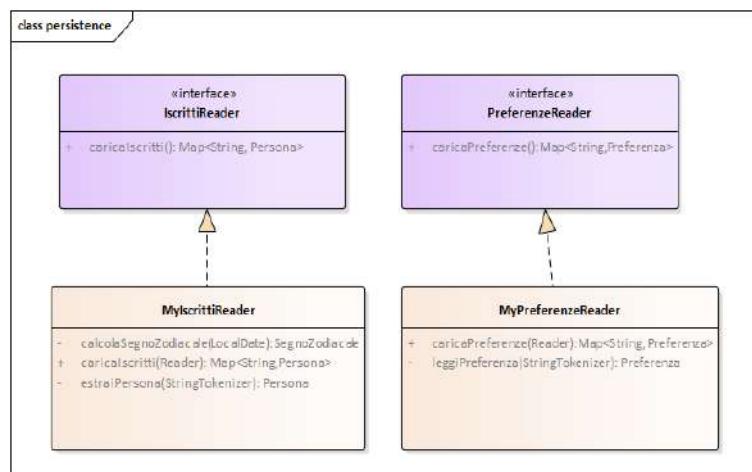
```
Roberto, M, 1994-04-24, capelli neri, occhi castani, 1.78, 61, Bologna, BO, Emilia-Romagna
Armando, M, 1997-08-01, capelli castani, occhi castani, 1.71, 65, Parma, PR, Emilia-Romagna
Eufrasio, M, 1993-10-30, capelli biondi, occhi azzurri, 1.82, 66, Firenze, FI, Toscana
Anna, F, 1996-06-18, capelli biondi, occhi azzurri, 1.70, 60, Imola, BO, Emilia-Romagna
Ludovica, F, 1992-10-14, capelli neri, occhi castani, 1.75, 51, Bologna, BO, Emilia-Romagna
Elena, F, 1999-04-10, capelli neri, occhi neri, 1.65, 57, Modena, MO, Emilia-Romagna
```

ESEMPIO DEL FILE *Preferenze.txt*

```
Roberto, F, 20-25, Bilancia, capelli biondi, occhi -, 1.70, 58, Bologna, BO, Emilia-Romagna
Armando, F, 17-25, -, capelli -, occhi -, -, -, Parma, PR, Emilia-Romagna
Anna, M, 20-29, Gemelli, capelli biondi, occhi azzurri, 1.70, 58, Imola, BO, Emilia-Romagna
Ludovica, M, 24-35, -, capelli -, occhi -, 1.80, -, Bologna, BO, Emilia-Romagna
Elena, M, 18-23, -, capelli neri, occhi azzurri, 1.75, 58, Modena, MO, Emilia-Romagna
```

Le due interfacce *IscrittiReader* e *PreferenzeReader* (fornite) dichiarano i metodi *caricaliscritti* e *caricaPreferenze* che, dato un *Reader*, restituiscono rispettivamente la mappa *Map<String, Persona>* degli iscritti e la mappa *Map<String, Preferenza>* delle preferenze, richieste dal costruttore di *Cupido*.

Le due classi *MyIscrittiReader* (da realizzare) e *MyPreferenzeReader* (fornita) implementano tali interfacce effettuando i necessari controlli sul formato del file, lanciando *BadFormatException* (fornita) in caso di errori di formato, o propagando *IOException* in caso di errori di lettura con specifico messaggio d’errore.



Parte 2

(punti: 9)

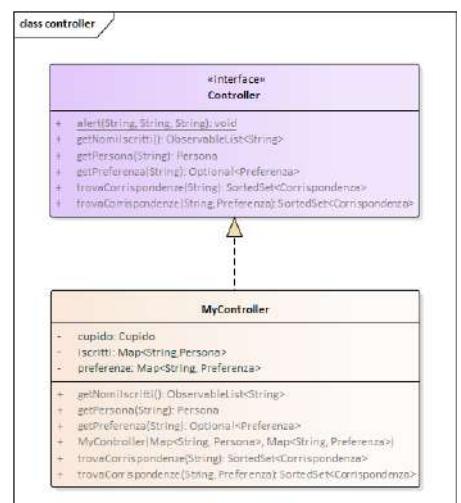
Controller (namespace `cupido.ui.controller`)

L'interfaccia **Controller** (fornita) dichiara i metodi **`getNomiscritti`**, **`getPersona`**, **`getPreferenza`** e due versioni di **`trovaCorrispondenze`**:

- **`getNomiscritti`**: restituisce la lista osservabile dei nomi degli iscritti;
- **`getPersona`**: restituisce la **Persona** corrispondente al nickname fornito;
- **`getPreferenza`**: restituisce, se esiste, la **Preferenza** della persona il cui nickname è specificato come argomento;
- **`trovaCorrispondenze`** (due versioni overloaded): richiama gli analoghi metodi di **Cupido**.

Essa fornisce inoltre il metodo statico `alert` che fa comparire una finestra di dialogo all'utente, utile per segnalare errori: i tre argomenti rappresentano il titolo della finestra, l'header e il testo del messaggio (v. Figg. 5 e 6).

La classe **MyController** (pure fornita) implementa **Controller** memorizzando internamente gli iscritti e le preferenze.

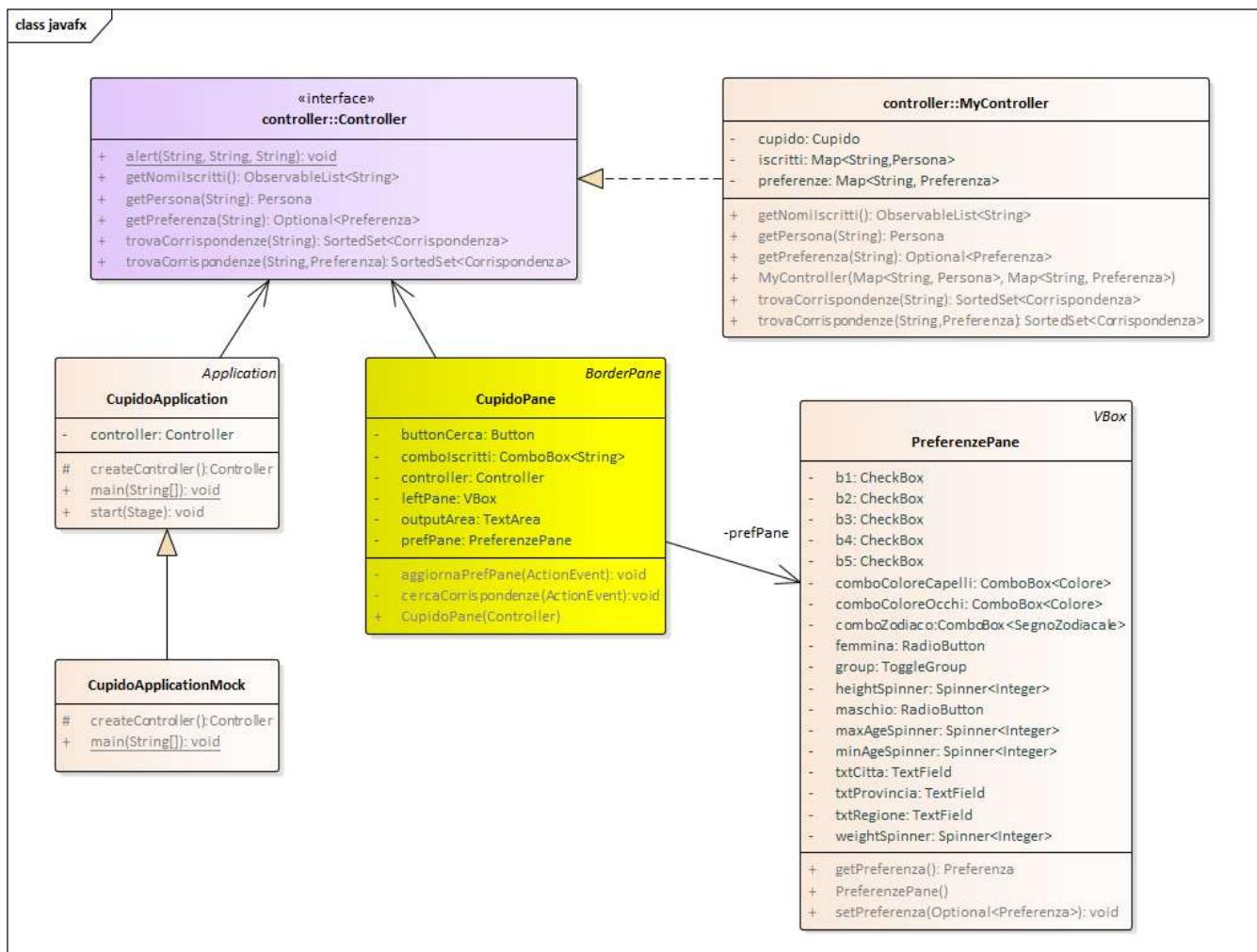


Interfaccia utente (namespace `cupido.ui.javafx`)

(punti 9)

L'interfaccia utente deve essere simile (non necessariamente identica) all'esempio mostrato nelle figure seguenti.

La classe **PreferenzePane** (fornita), offre già pronto un completo pannello per mostrare/leggere una preferenza utente, tramite i due metodi **`getPreferenza`** / **`setPreferenza`** (v. Figg. 1-4, parte sinistra).



La classe **CupidoPane** (da realizzare), che estende **BorderPane**, deve prevedere (v. Fig. 1):

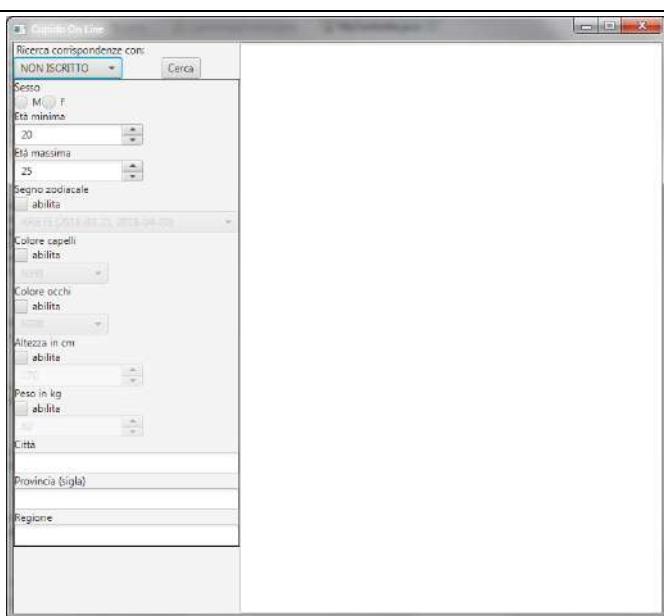
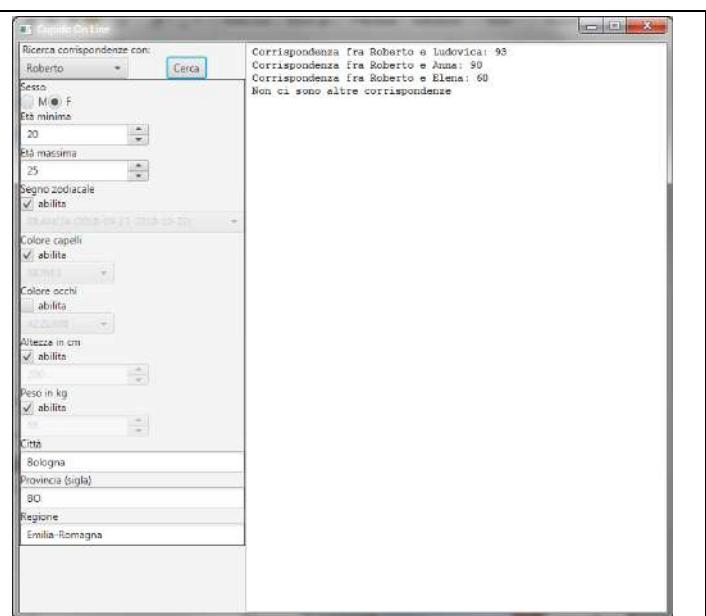
- una combo box per la selezione delle persone;
- un pulsante *Cerca* per scatenare la ricerca delle corrispondenze;
- sotto a questo, un pannello di tipo **PreferenzePane**;
- sulla destra, una textarea in cui mostrare i risultati

Il formato delle stampe (v. figure) è quello prodotto dalla *toString* di **Corrispondenza**, una per riga, completato da una frase finale che informa che non vi sono altre corrispondenze: tale frase è l'unica mostrata nel caso non vi sia alcuna corrispondenza.

All'inizio la combo è popolata con i nickname di tutte le persone disponibili, **più il valore “NON ISCRITTO” che dev'essere posto all'inizio** (Fig. 1). Se si sceglie una persona iscritta, il pannello preferenze sottostante ne mostra la preferenza (Fig. 2) [infatti, in questa modalità il pannello **PreferenzePane** viene usato solo come dispositivo di output]: quando l'utente preme il pulsante *cerca*, l'applicazione cerca le corrispondenze possibili e le mostra in ordine decrescente di indice di compatibilità (Fig. 3): se non ve ne sono, viene comunque mostrata la frase finale, per indicare che l'applicazione ha già operato e terminato (Fig. 4).

In alternativa, si può cercare corrispondenze per un NON ISCRITTO: in tal caso l'applicazione utilizzerà l'insieme di preferenze specificato dall'utente nel pannello **PreferenzePane**, inserendo in ogni campo i valori desiderati (Fig. 5): da notare che i campi opzionali devono essere singolarmente abilitati, altrimenti non saranno inclusi nella preferenza e quindi neppure nel successivo calcolo delle corrispondenze.

In caso di problemi (mancanza di uno o dell'altro file, errori di lettura, problemi di formato, etc.) l'applicazione deve mostrare opportuni dialoghi, sfruttando il metodo statico **Controller.alert** (Fig. 6).

 <p>Figura 1</p>	 <p>Figura 2</p>
--	--

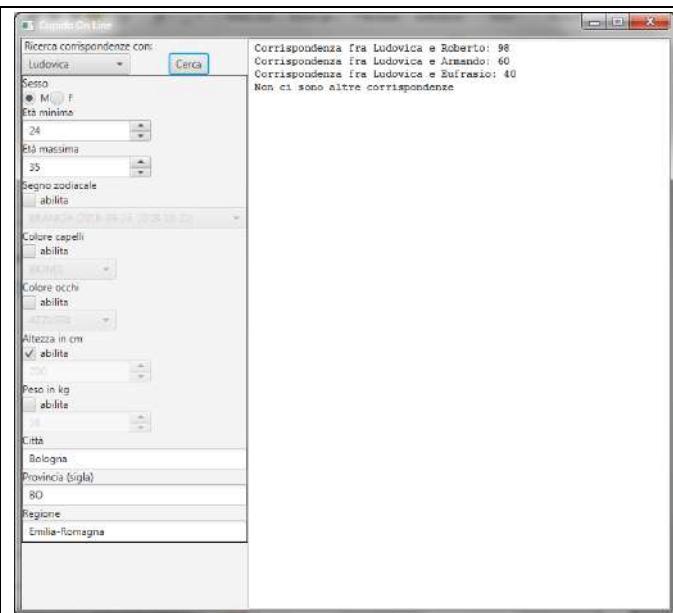


Figura 3

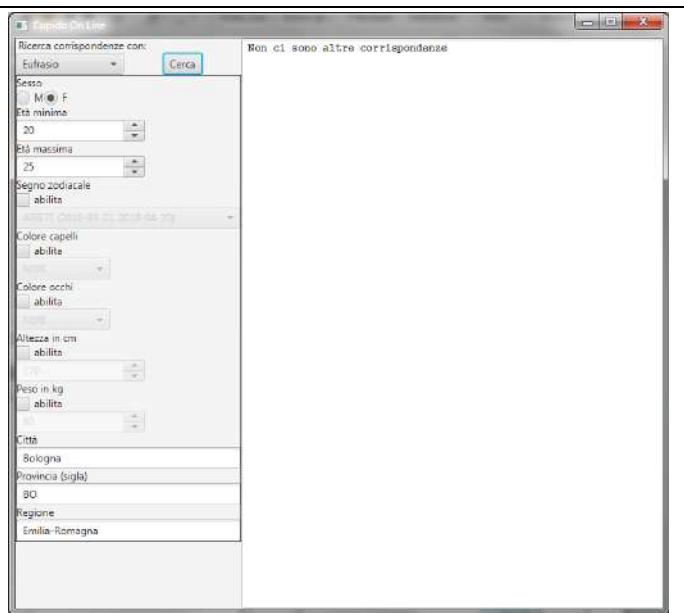


Figura 4

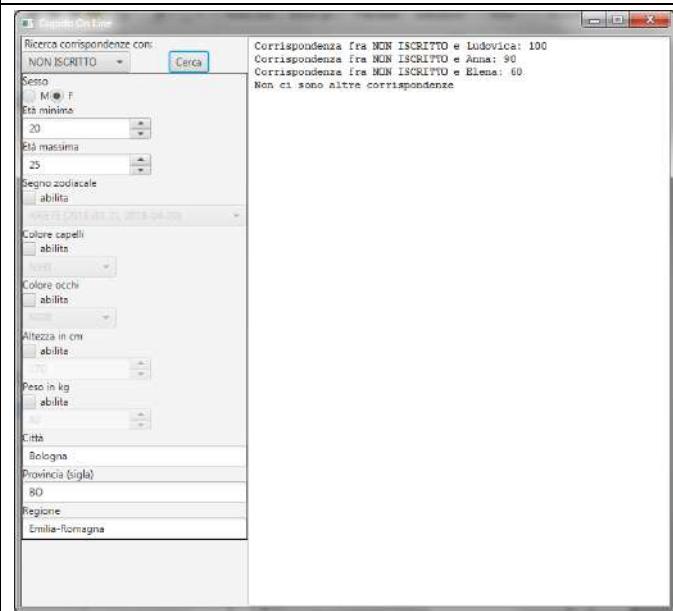


Figura 5

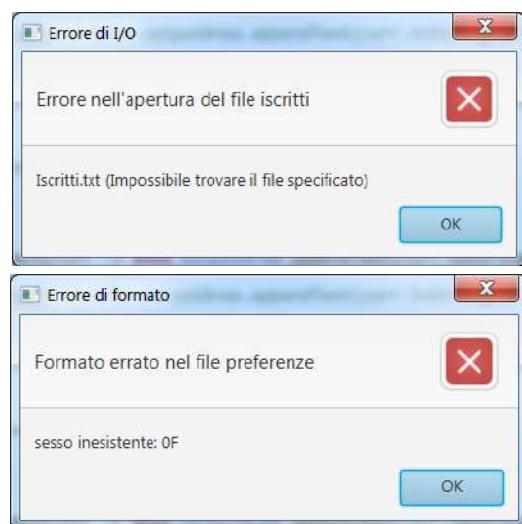


Figura 6

ESAME DI FONDAMENTI DI INFORMATICA T-2 del 13/06/2018

Proff. E. Denti – R. Calegari – G. Zannoni Tempo: 4 ore

NB: il candidato troverà nell'archivio ZIP scaricato da Esamix anche il software "Start Kit"

NOME PROGETTO ECLIPSE: CognomeNome-matricola (es. RossiMario-0000123456)

NOME ZIP DA CONSEGNARE: CognomeNome-matricola.zip (es. RossiMario-0000123456.zip)

La società *ElectricLife* ha richiesto lo sviluppo di un'applicazione che permetta ai potenziali utenti di simulare il costo della bolletta elettrica con le proprie tariffe.

DESCRIZIONE DEL DOMINIO DEL PROBLEMA.

La società offre sia tariffe *a consumo*, sia tariffe *flat* che comprendono già una certa quota di KWh mensili.

Nelle **tariffe a consumo** l'importo da pagare dipende dai KWh effettivamente consumati, mentre nelle **tariffe flat** l'importo da pagare ogni mese è fisso, purché i KWh consumati restino entro una soglia prestabilita; eventuali KWh in eccesso sono tariffati come extra, al prezzo specificato. In entrambi i casi al costo dell'energia devono essere aggiunte le **tasse** (accise e IVA), come più oltre precisato.

Più precisamente, una **tariffa a consumo** è caratterizzata da:

- nome dell'offerta commerciale
- prezzo del singolo KWh, in Euro

mentre ogni **tariffa flat** è caratterizzata da:

- nome dell'offerta commerciale
- quota di KWh inclusi nell'offerta mensile e relativo importo fisso, in Euro
- prezzo degli eventuali KWh consumati oltre la soglia, in Euro/KWh.

Le **tasse** comprendono due voci:

- le **accise**, calcolate in proporzione ai KWh fatturati (2,27 Eurocent/KWh), esclusi i primi 150 KWh mensili, esenti;
- l'**IVA** (10%), applicata sul subtotale precedente, accise incluse

Il totale della fattura va arrotondato a due cifre decimali, come previsto dalle normative.

ESEMPI

1. Tariffa flat a € 20 /mese con 150 KWh inclusi (eventuali KWh extra € 0,25/KWh): se il consumo resta entro la soglia, la fattura mensile sarà € 20 +10% IVA = **€ 22,00** [niente accise perché i primi 150 KWh sono esenti]
2. Tariffa a consumo al prezzo di € 0,14 /KWh, consumo 150KWh: la fattura sarà $(€ 0,14 * 150) + 10\% \text{ IVA} = € 21 + 10\% = € 23,10$ [niente accise perché i primi 150 KWh sono esenti]
3. Tariffa flat a € 20 /mese con 150 KWh inclusi (eventuali KWh extra € 0,25/KWh), consumo reale 184KWh: la fattura mensile sarà $(€ 20 + € 0,25 * 34 + € 0,0227 * 34) + 10\% \text{ IVA} = (€ 20 + € 9,2718) + 10\% = € 32,20$ [accise calcolate sui soli 34 KWh eccedenti la soglia dei 150 esenti]
4. Tariffa a consumo al prezzo di € 0,14 /KWh, consumo 184KWh: la fattura sarà $(€ 0,14 * 184 + € 0,0227 * 34) + 10\% \text{ IVA} = (€ 25,76 + € 0,7718) + 10\% = € 29,18$ [accise calcolate sui soli 34 KWh eccedenti la soglia]

Il file di testo [Tariffe.txt](#) contiene la descrizione delle diverse tariffe, nel formato più oltre specificato.

IMPORTANTE: a causa di modifiche nel provider delle specifiche **Locale** intervenute fra Java 8 e Java 9, **il formattatore di valute per Locale.ITALY opera diversamente in Java 9 rispetto a Java 8**. Ciò impatta anche il funzionamento dei metodi *parse* utilizzati per la conversione stringa(prezzo)/numero.

Per ottenere il comportamento classico è necessario aggiungere alle *Run Configurations* →*Arguments* la specifica:
-Djava.locale.providers=COMPAT

Essa va aggiunta sia alla *run configuration* di JUnit, sia a quella dell'Application, sia a quella dell'ApplicationMock.

NB: le run configuration vengono generate da Eclipse **dopo il primo run** di JUnit o dell'application stessa.

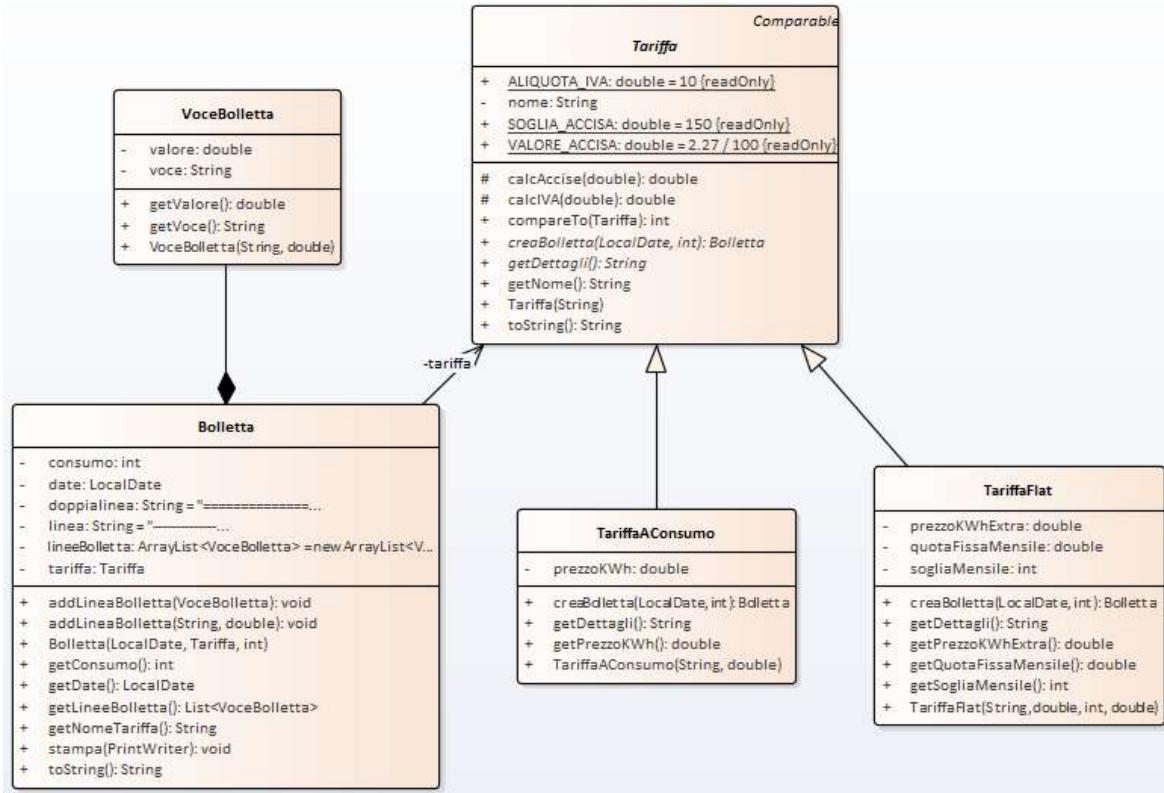
Parte 1

(punti: 18)

Dati (namespace `electriclife.model`)

(punti: 10)

Il modello dei dati deve essere organizzato secondo il diagramma UML di seguito riportato:



SEMANTICA:

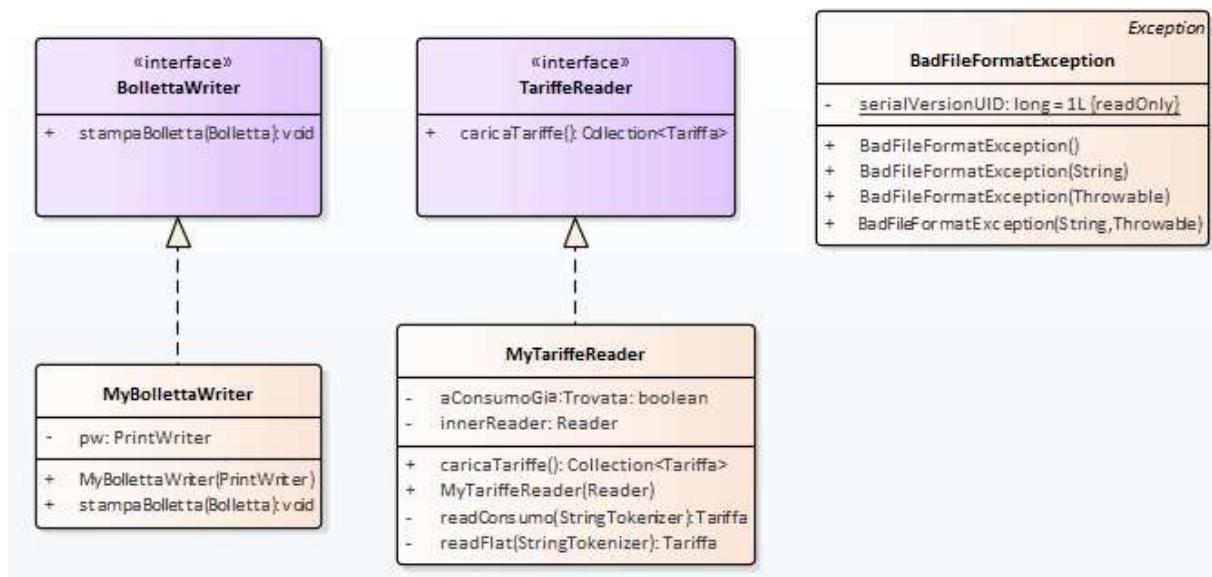
- a) la classe ***VoceBolletta*** (fornita) rappresenta una voce di costo di una bolletta, caratterizzata dalle due proprietà ***voce*** e ***valore***, recuperabili tramite opportuni accessori: essa serve a descrivere la struttura della ***Bolletta***.
 - b) la classe ***Bolletta*** (fornita) rappresenta la bolletta riferita a una certa data di emissione: è caratterizzata dal nome della tariffa applicata, dal consumo rilevato, nonché da una lista di ***VoceBolletta*** che specificano ciascuna una voce di costo. La classe offre metodi per aggiungere una linea di bolletta (***addVoceBolletta***), recuperare la lista delle linee di bolletta presenti (***getLineeBolletta***), produrre una stampa della bolletta su un apposito ***PrintWriter*** (***stampa***), oltre a una adeguata ***toString***.
 - c) la classe astratta ***Tariffa*** (fornita) rappresenta la generica tariffa caratterizzata da un nome univoco specificato all'atto della costruzione e recuperabile tramite opportuno accessorio; è anche ***Comparable*** per nome. Definisce inoltre opportune costanti (***VALORE_ACCISA***, ***SOGLIA_ACCISA***, ***ALIQUOTA_IVA***) per il calcolo delle varie voci. I due metodi concreti ***calcAccise*** / ***calcIVA*** restituiscono rispettivamente il costo delle accise / dell'IVA a partire da un consumo in KWh. Il metodo astratto ***creaBolletta***, dato il consumo mensile, crea la corrispondente ***Bolletta*** calcolando le voci di costo, le imposte e l'importo totale come da specifiche. Il metodo astratto ***getDettagli*** restituisce invece una stringa descrittiva della tariffa, che verrà riportata tale e quale in bolletta (v. esempi).
 - d) la classe ***TariffaAConsumo*** (da realizzare) specializza ***Tariffa*** nel caso delle tariffe a consumo: il costruttore riceve, oltre al nome della tariffa, il prezzo del KWh, recuperabile tramite accessorio. Il metodo ***creaBolletta*** deve inserire nella bolletta le quattro linee di bolletta necessarie, ovvero ***costo energia***, ***accise***, ***IVA***, ***totale***, mentre il metodo ***getDettagli*** restituisce una stringa del tipo "Tariffa A CONSUMO, Costo KWh € 0,14".
 - e) la classe ***TariffaFlat*** (da realizzare) specializza ***Tariffa*** nel caso delle tariffe flat: il costruttore riceve, oltre al nome, la quota fissa mensile, la soglia mensile di KWh inclusi e il prezzo degli eventuali KWh eccedenti; tutti questi valori sono recuperabili tramite opportuni accessori. Il metodo ***creaBolletta*** deve inserire nella bolletta le cinque linee di bolletta necessarie, ovvero: ***quota fissa mensile***, ***costo energia extra soglia***, ***accise***, ***IVA***, ***totale*** mentre ***getDettagli*** restituisce una stringa come "Tariffa CASETTA, € 20,00/mese per 150 KWh, poi € 0,25/KWh"

Come già anticipato, il file di testo `Tariffe.txt` contiene la descrizione delle tariffe offerte, una per riga: **per ovvi motivi può esserci una sola tariffa a consumo**, mentre le combinazioni flat possono essere molteplici. Ogni riga contiene una serie di dati separati fra loro da punti e virgola (‘;’): gli elementi della riga dipendono dal tipo di tariffa (a consumo o flat), come di seguito specificato. **Tutti i prezzi sono formattati in Euro nella forma € xx,xx.**

- Il primo token è costituito dalla tipologia della tariffa: “**FLAT**” o “**A CONSUMO**”; poi:
- per le tariffe a consumo, segue semplicemente il prezzo del KWh;
- per le tariffe flat, invece, seguono il nome della tariffa (che può contenere spazi), la soglia mensile (nel formato “**SOGLIA**” seguita dal valore intero che rappresenta i KWh inclusi), il prezzo mensile e infine il costo degli eventuali KWh extra (nel formato “**KWh EXTRA**” seguito dal relativo prezzo)

ESEMPIO DEL FILE `tariffe.txt`

```
FLAT ; CASA MINI ; SOGLIA 150; € 20,00; KWh EXTRA € 0,25
FLAT; CASA CLASSIC;SOGLIA 250 ; € 30,00; KWh EXTRA € 0,24
FLAT ;CASA BIG ; SOGLIA 350; € 40,00; KWh EXTRA € 0,22
FLAT; CASA MAXI; SOGLIA 450; € 50,00; KWh EXTRA € 0,21
A CONSUMO ; € 0,14
```



L' interfaccia **`TariffeReader`** (fornita) dichiara il metodo **`caricaTariffe`** che legge una lista di **`Tariffa`**.

La classe **`MyTariffeReader` (da realizzare)** implementa tale interfaccia: il metodo **`caricaTariffe`** deve effettuare i necessari controlli sul formato del file, inclusa la verifica che ci sia al più una sola tariffa a consumo, lanciando **`BadFileNotFoundException`** (fornita) in caso di errori di formato, o propagando **`IOException`** in caso di errori di lettura. Il costruttore riceve il **`Reader`** da cui leggere.

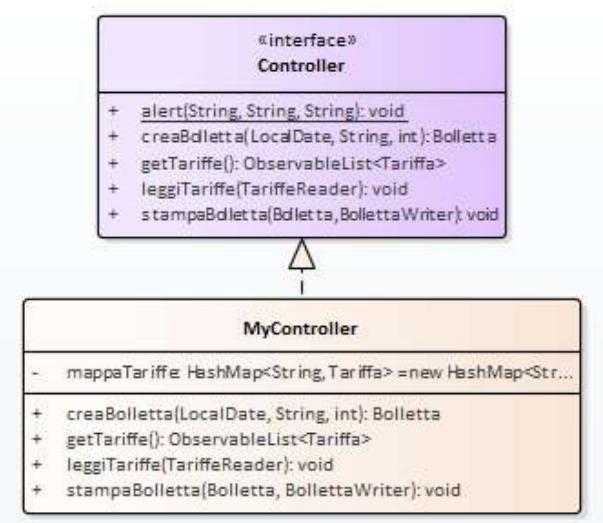
L' interfaccia **`BollettaWriter`** (fornita) e la corrispondente classe **`MyBollettaWriter`** (pure fornita) rispettivamente dichiarano/implementano il metodo **`stampaBolletta`** che stampa una **`Bolletta`**.

Parte 2 (punti: 12)

Controller (namespace `electriclife.ui.controller`) (punti 6)

L'interfaccia **`Controller`** (fornita) dichiara i metodi **`leggiTariffe`, `getTariffe`, `creaBolletta` e `stampaBolletta`** che devono:

- `leggiTariffe`**: caricare le tariffe su cui operare;



- **getTariffe**: restituire la lista osservabile ordinata di tutte le tariffe disponibili;
- **creaBolletta**: produrre la **Bolletta** a partire dai dati ricevuti (data di emissione, nome della tariffa, consumo); a tal fine recupera la tariffa per nome e delega poi ad essa la creazione effettiva della **Bolletta**;
- **stampaBolletta**: stampare la **Bolletta** tramite il **BollettaWriter** fornito.

Essa fornisce inoltre il metodo statico alert che fa comparire una finestra di dialogo all'utente, utile per segnalare errori: i tre argomenti rappresentano il titolo della finestra, l'header e il testo del messaggio (v. Figg. 5 e 6).

La **classe MyController (da realizzare)** deve implementare **Controller** memorizzando internamente al controller le tariffe caricate (si suggerisce una mappa avente per chiave il nome della tariffa); : in particolare, **leggiTariffe** deve lasciar uscire eventuali eccezioni in modo da consentirne poi l'opportuna gestione nella GUI.

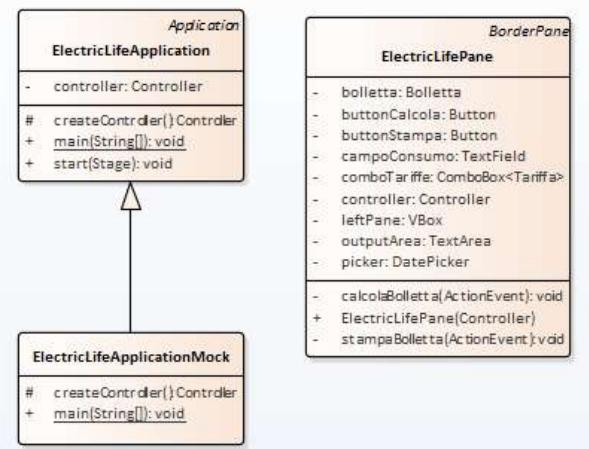
Interfaccia utente (namespace electriclife.ui)

(punti 6)

L'interfaccia utente deve essere simile (non necessariamente identica) all'esempio mostrato nelle figure seguenti.

La classe ElectricLifePane (da realizzare), che estende **BorderPane**, deve prevedere:

- una combo box per la selezione della tariffa
- un text field per inserire i dati di consumo;
- un datepicker per selezionare la data di emissione;
- un pulsante *Calcola* per effettuare il calcolo della bolletta, mostrando il risultato nella textarea;
- un pulsante *Stampa* per stampare la bolletta su file.



Formato bolletta visualizzata o stampata

=====
Electric Life - L'energia che illumina!
Bolletta del 8 maggio 2018

Tariffa A CONSUMO, Costo KWh € 0,14
Consumo KWh 184

Dettaglio importi:
Costo energia € 25,76
Corrispettivo per accise € 0,77
Corrispettivo per IVA € 2,65
Totale Bolletta € 29,18

NB: la posizione del simbolo € in alcune righe potrebbe essere posposta in Java 9: ciò non costituisce problema.

Formato bolletta visualizzata o stampata

=====
Electric Life - L'energia che illumina!
Bolletta del 8 maggio 2018

Tariffa CASA MINI, € 20,00/mese per 150 KWh, poi € 0,25/KWh
Consumo KWh 150

Dettaglio importi:
Quota fissa mensile € 20,00
Costo energia extra soglia € 0,00
Corrispettivo per accise € 0,00
Corrispettivo per IVA € 2,00
Totale Bolletta € 22,00

NB: la posizione del simbolo € in alcune righe potrebbe essere posposta in Java 9: ciò non costituisce problema.

Inizialmente, la combo è popolata con tutte le tariffe disponibili e pre-settata sulla tariffa “**A CONSUMO**”, il datepicker è impostato sulla data corrente e i campi sono vuoti (Fig. 1).

L’utente sceglie la tariffa, inserisce il consumo, sceglie la data di emissione (default: data odierna – Fig. 2) e preme il pulsante *Calcola*: in risposta, l’applicazione mostra la bolletta calcolata (Figg. 3, 4).

Premendo il pulsante *Stampa*, la stessa bolletta viene salvata sul file **Bolletta.txt**. Da notare che il pulsante *Stampa* è inizialmente disabilitato: si abilita a seguito di un nuovo calcolo e si disabilita a stampa effettuata.

In caso di problemi l’applicazione deve mostrare opportuni dialoghi, sfruttando il metodo statico **Controller.alert**: consumi negativi o mancanti (Fig. 5), mancanza del file delle tariffe o altri problemi di lettura (Fig. 6), problemi nella stampa della bolletta (Fig. 5).

SUGGERIMENTI

- per fare il parsing di stringhe formattate in valuta utilizzare i metodi **parse** del formattatore di valute
- per stampare stringhe formattate in valuta utilizzare i metodi **format** del formattatore di valute
- per stampare righe formattate a campi di larghezza fissa può essere utile il metodo **format** del **PrintWriter** (che, nella versione più ampia, accetta un **Locale** come primo argomento)

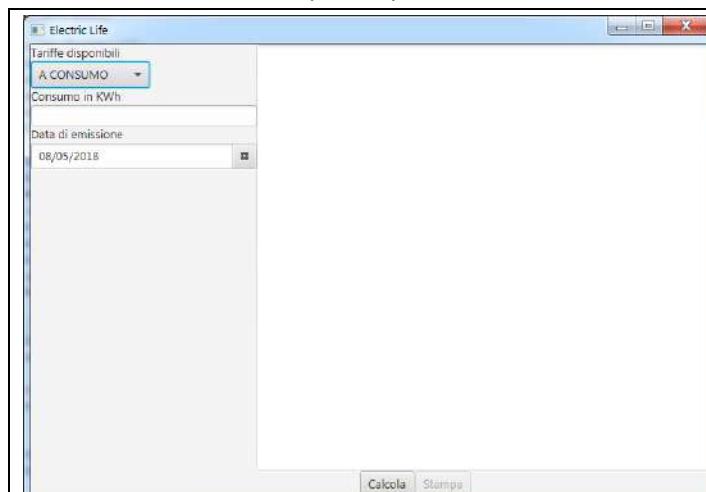


Figura 1

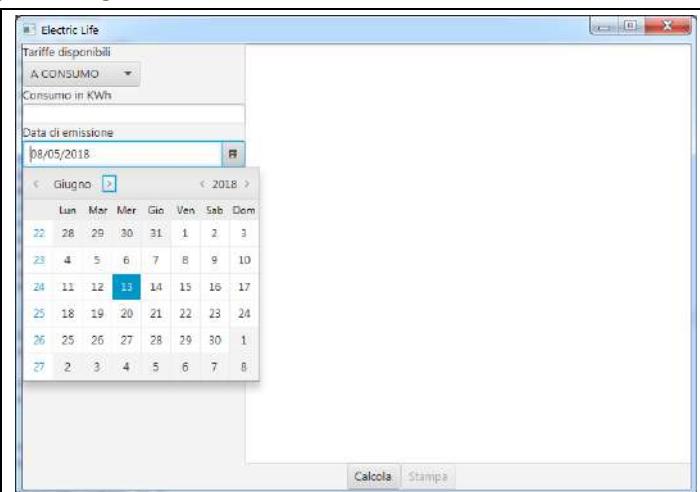


Figura 2

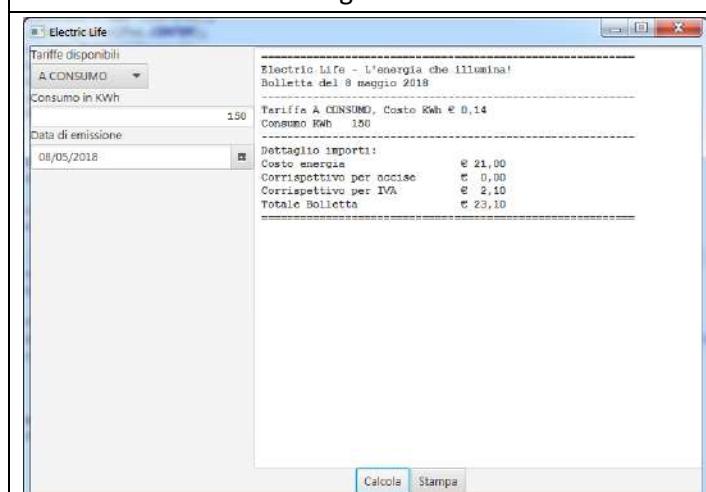


Figura 3

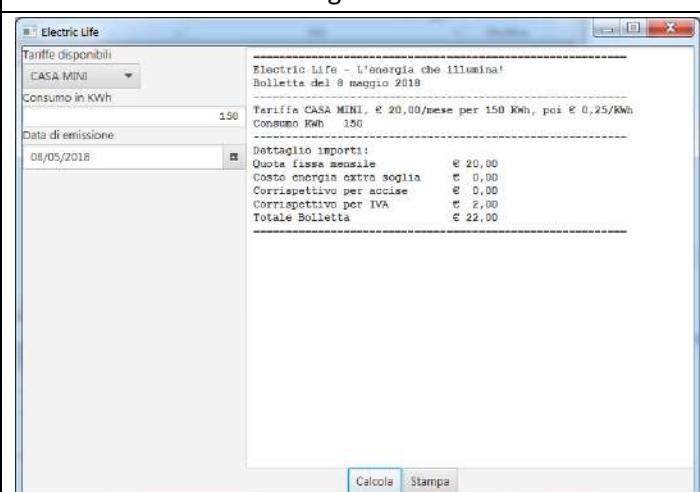


Figura 4

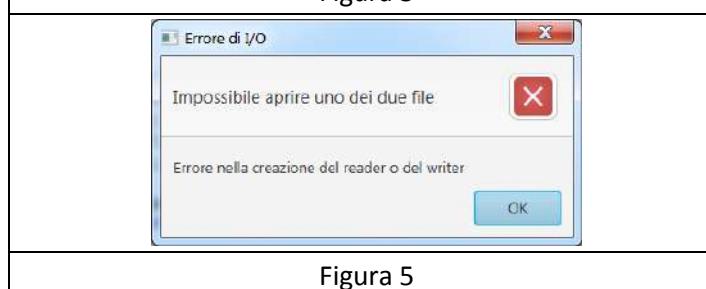


Figura 5

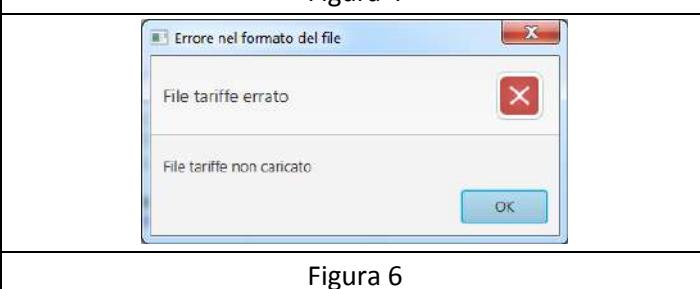


Figura 6

ESAME DI FONDAMENTI DI INFORMATICA T-2 del 7/2/2018

Proff. E. Denti – G. Zannoni

Tempo a disposizione: 4 ore MAX

NB: il candidato troverà nell'archivio ZIP scaricato da Esamix anche il software "Start Kit"

NOME PROGETTO ECLIPSE e CARTELLA : CognomeNome-matricola (es. RossiMario-0000123456)

NOME ZIP DA CONSEGNARE : CognomeNome-matricola.zip (es. RossiMario-0000123456.zip)

L'Unione Banche di Zannonia (UBZ) ha richiesto un'applicazione di ausilio per gli impiegati dei propri sportelli, che indichi visivamente quante banconote, e di che taglio, dare al cliente che effettui un prelievo in contanti.

DESCRIZIONE DEL DOMINIO DEL PROBLEMA

Quando un cliente si presenta allo sportello bancario per effettuare un prelievo, non tutte le combinazioni di banconote e monete possibili teoricamente lo sono anche realmente: ad esempio, non è realistico prelevare € 2000 in monete da € 2. Ciò che è effettivamente possibile dipende da:

1. l'importo da prelevare
2. la quantità di banconote dei vari tagli disponibili allo sportello in quel momento
3. le eventuali politiche aziendali (ad es. "non si danno più di dieci pezzi da € 10 in una singola operazione").

In generale, il cassiere sceglie sempre la combinazione che usa il massimo numero di banconote di taglio elevato, passando a usare banconote di taglio inferiore solo quando è indispensabile farlo.

ESEMPIO

Si supponga di voler prelevare € 180. Se lo sportello dispone di tutti i tagli e non ha limiti nella scelta delle banconote, le combinazioni possibili teoricamente sono molte: il cassiere però predilige quella che usa le banconote di taglio più elevato, ossia la prima fra quelle sotto elencate.

- 3x € 50 + 1x € 20 + 1x € 10 ← *prescelta dal cassiere*
- 3x € 50 + 3x € 10
- 9x € 20
- 7x € 20 + 4x € 10
- ...
- 1x € 20 + 16x € 10
- 18x € 10

Tuttavia, se lo sportello non dispone di certe banconote (ad esempio, non ha banconote da € 10) o ne ha in quantità limitata (ad esempio, soltanto 5) o, per politica aziendale, non dà allo stesso cliente più di tot banconote di piccolo taglio (ad esempio, non più di 3 pezzi da € 10), alcune combinazioni non risultano più possibili e vanno escluse.

Ad esempio, se il cassiere ha finito le banconote da € 50, dovrà scegliere solo combinazioni che usino € 20, 10, 5, 2, 1.

Nel caso di UBZ, la politica aziendale esprime appunto un tetto al numero di banconote o monete erogabili di un certo taglio: il **cassiere UBZ**, quindi, non dovrà tenere conto soltanto dei normali vincoli di cassa, come nell'esempio sopra, ma anche di questi ulteriori vincoli, che potranno portare a **escludere determinate combinazioni** – o, in casi limite, a rendere perfino impossibile il prelievo per impossibilità di comporre la cifra richiesta.

Il file **binario DotazioneSportello.dat** contiene **due oggetti** (una istanza di **Disponibilità** e una di **Politiche**) che specificano, per ogni taglio di banconote e monete da 1€ e 2€ (si trascurano le monete di valore inferiore), rispettivamente le *quantità disponibili inizialmente* all'apertura dello sportello e le *politiche aziendali*, ossia il numero massimo di banconote o monete di quel taglio che si possono dare a un cliente nel corso di una singola operazione.

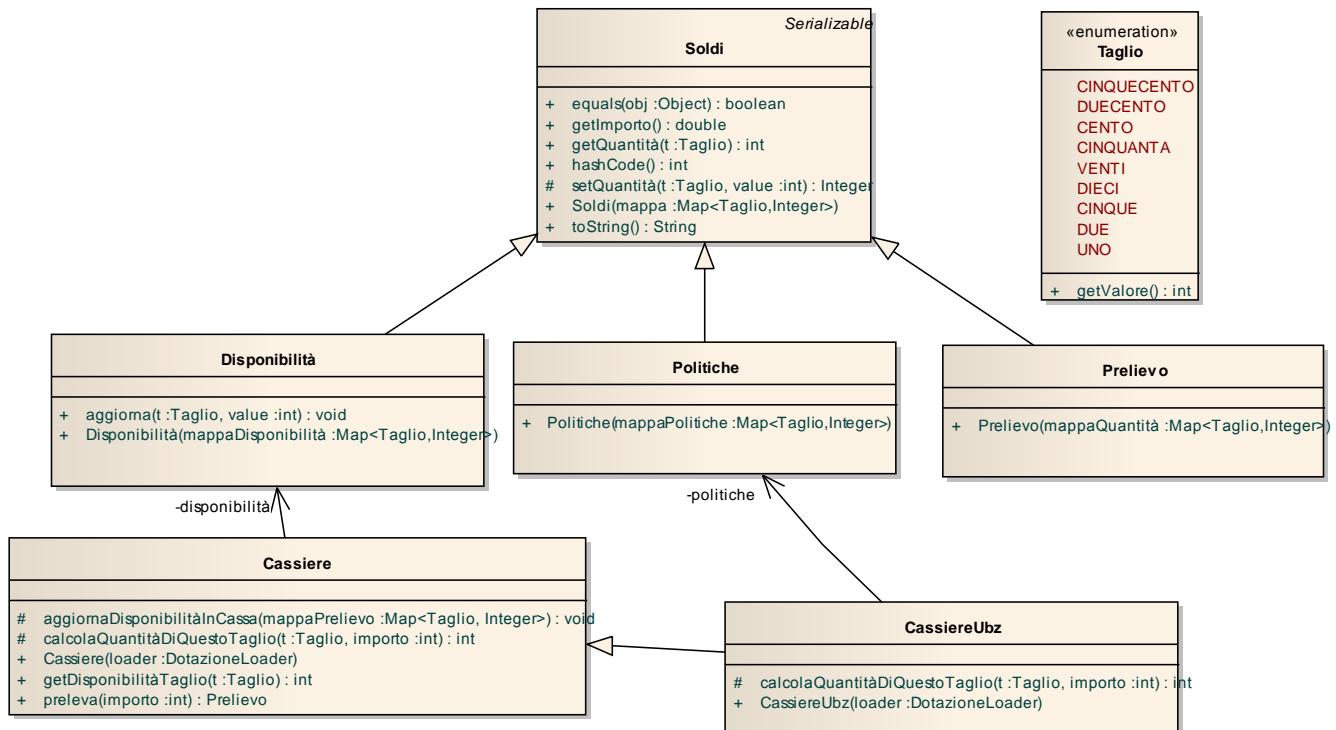
IMPORTANTE: eliminare dal progetto la parte grafica non usata.

Parte 1

(punti: 13)

Dati (package ubz.model)

(punti: 7)



SEMANTICA:

- L'enumerativo **Taglio** (fornito) elenca tutti i tagli di banconote e monete disponibili, in ordine decrescente da € 500 a € 1; ogni istanza dell'enumerativo incorpora il corrispondente valore numerico, recuperabile col metodo `getValore`.
- La classe **Soldi** (fornita) rappresenta un dato insieme di banconote e monete, memorizzate internamente sotto forma di mappa `Map<Taglio, Integer>`: il costruttore riceve un argomento del medesimo tipo. Sono forniti i metodi accessori `getQuantità(Taglio)` e `setQuantità(Taglio, int)` e `getImporto`: il primo restituisce la quantità di banconote o monete di quel taglio presenti, il secondo [protetto] la re-imposta, mentre il terzo calcola e restituisce l'importo totale corrispondente ai soldi presenti. Il metodo `toString` stampa in forma sintetica l'importo e la composizione dello stesso.
- Le tre classi **Disponibilità**, **Politiche** e **Prelievo** (fornite) specializzano **Soldi** per rappresentare rispettivamente i soldi presenti in cassa, le politiche aziendali, e un prelievo effettuato allo sportello: **Disponibilità** offre in più il metodo pubblico `aggiorna(Taglio t, int)` che consente di aggiornare la disponibilità in cassa di un certo taglio di banconote o monete.
- La classe **Cassiere** (fornita) rappresenta lo sportello bancario: costruita a partire da un `DotazioneLoader`, incorpora al suo interno la **Disponibilità**, recuperabile tramite apposito accessor. Il metodo fondamentale è `preleva`, che, dato un importo, sintetizza la combinazione di banconote e monete da dare al cliente, restituendola sotto forma di **Prelievo**. A tal fine sfrutta due metodi protetti, `calcolaQuantitàDiQuestoTaglio` e `aggiornaDisponibilitàInCassa`: il primo calcola quante banconote o monete di un dato taglio sono necessarie per comporre il prelievo, mentre il secondo toglie dalla disponibilità di cassa le banconote e le monete utilizzate per un il prelievo. Nella versione base di questa classe, il metodo `calcolaQuantitàDiQuestoTaglio` non tiene conto delle politiche aziendali: pertanto, non c'è limite al numero di banconote o monete di un dato taglio che possono essere erogate in una singola operazione (se non quello della loro disponibilità in cassa).

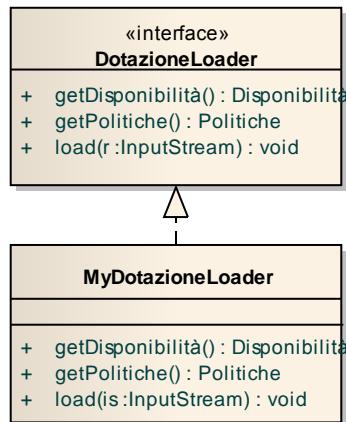
- e) la classe **CassiereUbz** (**da realizzare**) specializza **Cassiere**: costruita a partire da un **DotazioneLoader**, incorpora al suo interno le **Politiche** (in un opportuno *field*) re-implementando *calcolaQuantitàDiQuesto-Taglio* in modo da tener conto delle politiche aziendali: pertanto, la quantità erogabile di un certo taglio sarà limitata, oltre che dalla disponibilità, anche dalle politica aziendale corrispondente.

NB: chi non riuscisse a svolgere questa parte potrà fornire una implementazione vuota che si limiti a ereditare dal cassiere base, in modo da poter proseguire con la successiva parte del compito.

Persistenza (ubz.persistence)

(punti 6)

Come già anticipato, il file binario **DotazioneSportello.dat** contiene **due oggetti** (una istanza di **Disponibilità** e una di **Politiche**) che specificano, per ogni taglio di banconote e monete, rispettivamente le *quantità disponibili inizialmente all'apertura dello sportello* e le *politiche aziendali*, ossia il numero massimo di banconote o monete di quel taglio che si possono dare a un dato cliente nel corso di una singola operazione.



SEMANTICA:

- a) L'interfaccia **DotazioneLoader** (fornita) dichiara tre metodi:
- *loadMaps* che carica dal file binario la disponibilità e le politiche aziendali;
 - *getDisponibilità* e *getPolitiche* restituiscono rispettivamente l'istanza di **Disponibilità** o **Politiche** caricate.
- b) La classe **MyDotazioneLoader** (**da realizzare**) implementa **DotazioneLoader**: il metodo *loadMaps* riceve in ingresso l'**InputStream** da cui leggere i dati, ed emette:
- una **IllegalArgumentException** nel caso l'inputstream ricevuto sia nullo;
 - l'opportuna **BadFormatException** con messaggio d'errore appropriato in caso di problemi nel formato del file (mancanza di entrambe le mappe, di una sola mappa, o presenza nel file binario di oggetti estranei di tipo diverso dalla mappa)
 - una **IOException** in caso di altri problemi di I/O.

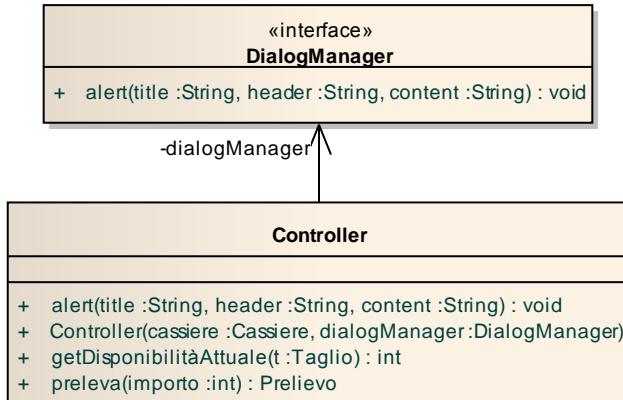
PROSEGUE ALLA PAGINA SUCCESSIVA

Parte 2

(punti: 17)

Controller (ubz.ui.controller)

Il Controller è organizzato secondo il diagramma UML seguente:



SEMANTICA:

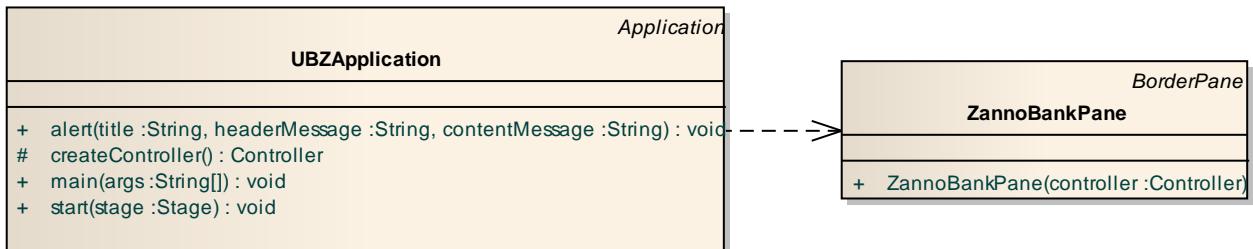
La classe **Controller** (fornita) dichiara l’interfaccia del controller (metodo `getCassiere`) e implementa i metodi:

- `alert`, utile per mostrare avvisi all’utente tramite un **DialogManager**: quest’ultimo è passato in fase di costruzione unitamente al **Cassiere**;
- `getDisponibilitàAttuale`, che restituisce il numero di banconote disponibili per il taglio specificato: usa cassiere per ottenere tale informazione;
- `preleva`, che ridirige il controllo al cassiere per effettuare l’operazione di prelievo.

Interfaccia Utente JavaFX (ubz.ui.javafx) per studenti A.A. 2016/17

(punti 17)

L’architettura segue il modello sotto illustrato:



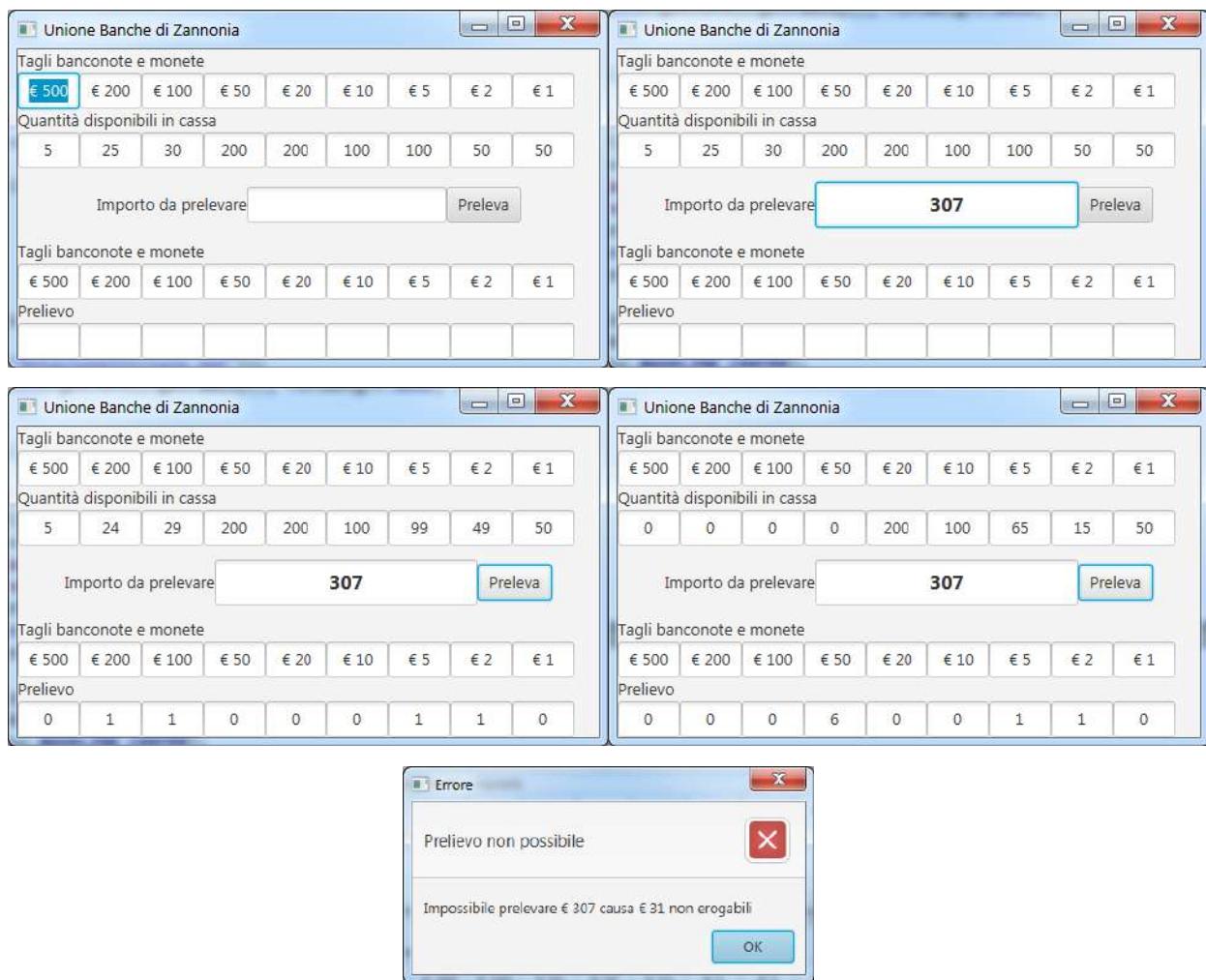
La classe **UBZApplication** (fornita) costituisce l’applicazione JavaFX che si occupa di aprire il file, il controller e incorporare lo **ZannoBankPane** (da realizzare). Per consentire di collaudare la GUI anche in assenza della parte di persistenza, è possibile avviare l’applicazione mediante la classe **UBZApplicationMock**.

L’interfaccia utente deve essere simile (non necessariamente identica) all’esempio mostrato nella figura seguente.

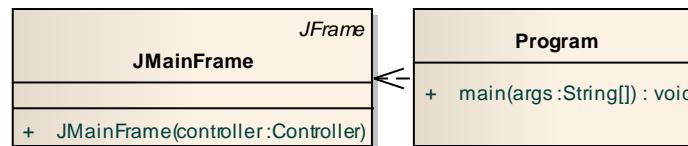
In alto, una griglia di campi di testo (non editabili) elenca i possibili tagli di banconote e monete, seguiti da un’analoga griglia con le disponibilità di cassa (Fig. 1). Al centro, un campo di testo (con testo centrato e font bold 16 punti) consente di inserire l’importo da prelevare (Fig. 2): il successivo tasto *Preleva* attiva il prelievo. In risposta a tale evento (Fig. 3), le due griglie di campi di testo sottostanti dettagliano la composizione del prelievo (ossia quante banconote e monete di quali tagli vengono erogate): la disponibilità di cassa (in alto) cala di conseguenza.

Di prelievo in prelievo, naturalmente, la cassa cala sempre più: le banconote di valore più elevato iniziano via via a mancare e il prelievo viene proposto con banconote di minor valore (Fig. 4), fino al punto in cui il prelievo stesso non può più avvenire (Fig. 5) – tipicamente, perché si violerebbero le politiche aziendali.

La classe **ZannoBankPane** deve estendere **BorderPane**; per le griglie di campi di testo si suggerisce l'uso di opportuni **TilePane**. [NB: poiché non è possibile includere lo stesso componente grafico due volte in un Frame, la griglia dei tagli di banconote e monete deve essere duplicata per poter essere mostrata sia in alto sia in basso].



SWING: VEDERE ALLA PAGINA SUCCESSIVA

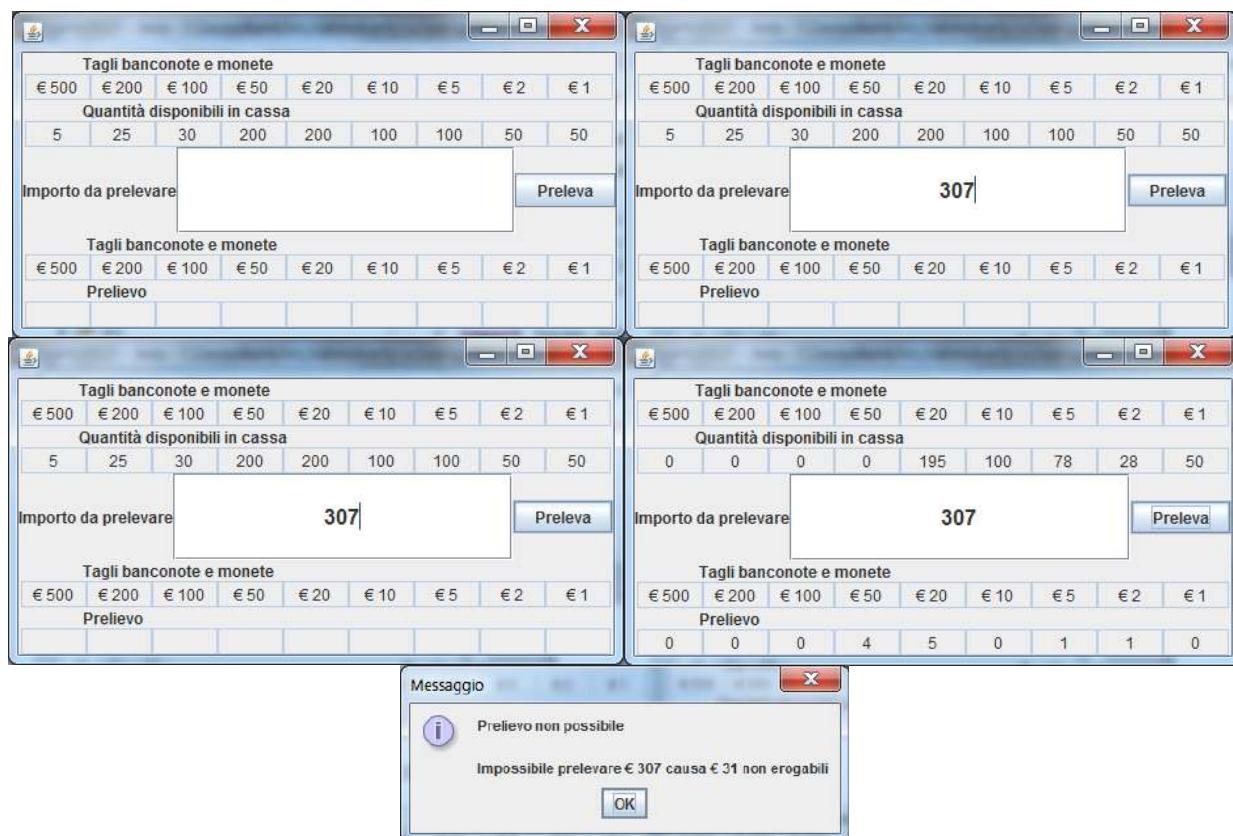


L'interfaccia utente deve essere simile (non necessariamente identica) all'esempio mostrato nella figura seguente.

In alto, una griglia di campi di testo (non editabili) elenca i possibili tagli di banconote e monete, seguiti da un'analogia griglia con le disponibilità di cassa (Fig. 1). Al centro, un campo di testo (con testo centrato e font bold 16 punti) consente di inserire l'importo da prelevare (Fig. 2): il successivo tasto *Preleva* attiva il prelievo. In risposta a tale evento (Fig. 3), le due griglie di campi di testo sottostanti dettagliano la composizione del prelievo (ossia quante banconote e monete di quali tagli vengono erogate): la disponibilità di cassa (in alto) cala di conseguenza.

Di prelievo in prelievo, naturalmente, la cassa cala: le banconote di valore più elevato iniziano via via a mancare e il prelievo viene proposto con banconote di minor valore (Fig. 4), fino al punto in cui il prelievo non può più avvenire (Fig. 5) – tipicamente, perché si violerebbero le politiche aziendali.

[NB: poiché non è possibile includere lo stesso componente grafico due volte in un Frame, la griglia dei tagli di banconote e monete deve essere duplicata per poter essere mostrata sia in alto sia in basso].



La classe **Program** (fornita) contiene il *main* di partenza dell'intera applicazione, che si occupa di aprire il file e creare tutto il necessario. Per consentire di collaudare la GUI anche in assenza della parte di persistenza, è possibile avviare l'applicazione mediante la classe **GUITest**.

La classe **JMainFrame** (**da realizzare**) deve organizzare l'interfaccia come sopra illustrato: il pannello principale deve adottare **BorderLayout**. Per le griglie di campi di testo si suggerisce l'uso di opportuni sotto-pannelli gestiti da **GridLayout**. [NB: poiché non è possibile includere lo stesso componente grafico due volte in un JFrame, la griglia dei tagli di banconote e monete deve essere duplicata per poter essere mostrata sia in alto sia in basso].

ESAME DI FONDAMENTI DI INFORMATICA T-2 del 17/01/2018

Proff. E. Denti – G. Zannoni

Tempo a disposizione: 4 ore MAX

NB: il candidato troverà nell'archivio ZIP scaricato da Esamix anche il software "Start Kit"

NOME PROGETTO ECLIPSE e CARTELLA: CognomeNome-matricola (es. RossiMario-0000123456)

NOME ZIP DA CONSEGNARE : CognomeNome-matricola.zip (es. RossiMario-0000123456.zip)

Un'azienda di Quality Assurance deve, per lavoro, effettuare decine di misure di precisione, da cui estrarre valori statistici utili per le relazioni tecniche. A tal fine ha commissionato un'applicazione che elabori le misure (con i relativi dettagli) estraendone le statistiche richieste, personalizzabili con vari filtri, e li mostri graficamente.

DESCRIZIONE DEL DOMINIO DEL PROBLEMA

In generale, per verificare la qualità e la corrispondenza di un prodotto alle specifiche si effettuano una serie di misure, elaborandone poi le risposte in modo da estrarne uno o più parametri significativi (*indicatori*), che devono rientrare entro limiti prestabiliti (*tolleranze*). Esempi: percentuale di oggetti che rientrano entro l'1% nelle specifiche di peso o di lunghezza, percentuale di oggetti scartati perché troppo grandi, scoloriti, etc.

Nel caso in questione interessa soltanto il peso del prodotto, che deve rientrare entro i seguenti limiti:

- fino a 50g di peso: tolleranza del 9% del peso [quindi, fra 0 e 4.5g]
- da 50 a 100g: tolleranza fissa di 4.5g
- da 100 a 200g: tolleranza del 4.5% del peso [quindi, fra 4.5g e 9g]
- da 200 a 300g: tolleranza fissa di 9g
- da 300 a 500g: tolleranza del 3% del peso [quindi, fra 9g e 15g]
- da 500 a 1000g: tolleranza fissa di 15g
- oltre 1000g di peso: tolleranza del 1.5% del peso [quindi, da 15g in su, in proporzione al peso]

Si tratta chiaramente di una funzione continua a tratti, che consente (giustamente) tolleranze un po' maggiori sulle piccole confezioni, dove sarebbe difficile dosare il prodotto al grammo, e via via la riduce per le confezioni più grandi.

Ai fini della tutela del consumatore, però, interessano solo le confezioni **sotto-dosate**: se una confezione contiene più prodotto del previsto, infatti, l'azienda ci rimette ma il consumatore non ha di che lamentarsi.

ESEMPIO

Si supponga di aver misurato il peso di decine di confezioni di pasta o biscotti, uscite da una catena di produzione: nominalmente dovrebbero essere da 100g, ma di fatto potrebbero contenere 99.80g, 98.85g, 101.16g, di prodotto.

Quelle di peso inferiore non devono superare una certa percentuale: altrimenti, l'azienda non soddisfa i requisiti di qualità richiesti.

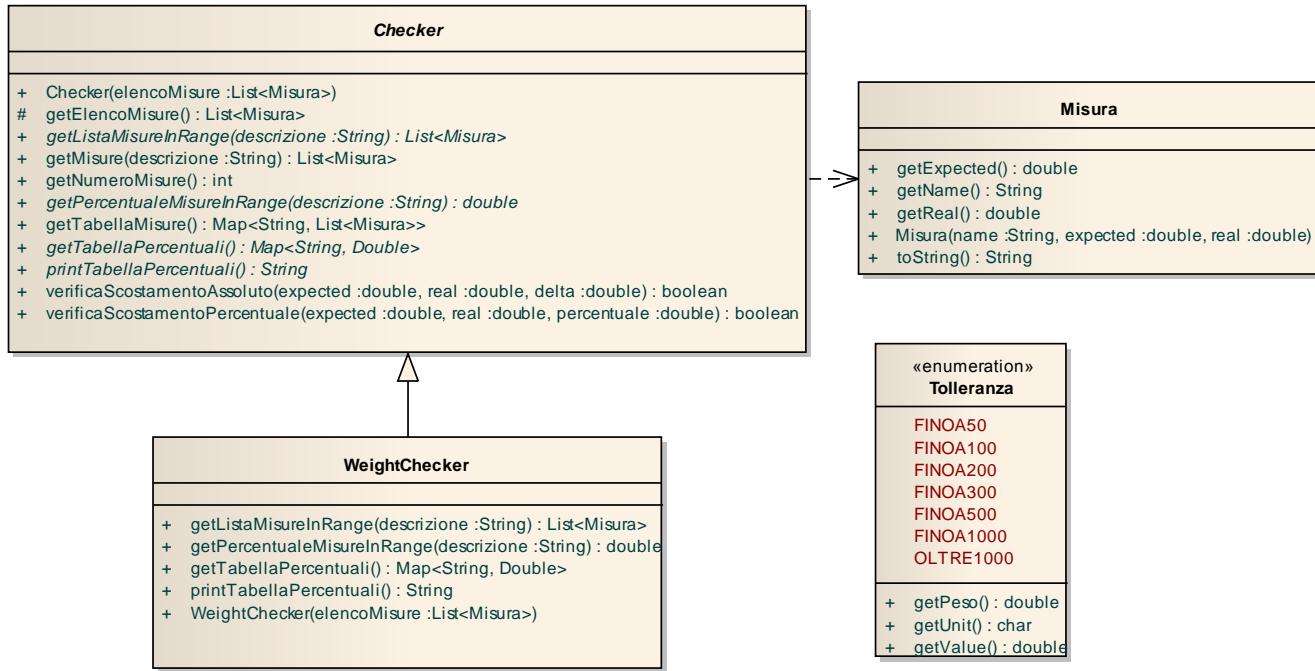
Il file di testo **Misure.txt** contiene i risultati delle misure (nel formato dettagliato più oltre), una per riga.

Parte 1

(punti: 18)

Dati (package qa.model)

(punti: 14)



SEMANTICA:

- L'enumerativo **Tolleranza** (fornito) definisce le sette costanti che catturano i casi della funzione continua a tratti: ogni valore dell'enumerativo incorpora il peso limite e la tolleranza corrispondente (in percentuale o in grammi, secondo i casi); sono forniti tre metodi accessor per recuperare peso, valore e unità di misura.
- La classe **Misura** (fornita) rappresenta il risultato di una misura, caratterizzata dal nome del prodotto misurato e, rispettivamente, dal valore atteso e da quello effettivamente misurato;
- la classe astratta **Checker** (fornita) incapsula i dati e l'algoritmo per verificarne la conformità. Il costruttore riceve un elenco di misure e lo memorizza internamente, popolando anche una mappa ausiliaria avente per chiave i nomi dei prodotti e per valori la lista di misure relative a tale prodotto. Sono forniti opportuni accessori per recuperare l'elenco di tutte le misure (*getElencoMisure*), il numero totale di misure presenti (*getNumeroMisure*), l'elenco delle sole misure relative a un certo prodotto (*getMisure*) e la mappa ausiliaria suddetta (*getTabellaMisure*).

Sono inoltre forniti i due metodi *verificaScostamentoPercentuale* e *verificaScostamentoAssoluto* che confrontano un valore atteso rispetto a quello effettivo e restituiscono **true** se il valore effettivo è entro il range previsto, applicando la semantica descritta nel dominio del problema (OVVERO: sono fuori range solo le confezioni **sotto-dosate** che eccedono la tolleranza prevista per il loro peso). I due metodi si differenziano solo per il criterio di confronto, che è percentuale nel primo caso e assoluto nel secondo.

Ad esempio, se una confezione da 500g ne pesa in realtà 490, *verificaScostamentoPercentuale(500,490,3%)* restituisce **true** (3% di 500g = 15g), mentre *verificaScostamentoAssoluto(500,490,5)* restituisce **false**, perché i 10g mancanti non rientrano nella fascia fissa di 5g specificata.

Al contrario, se una confezione da 250 g ne pesa in realtà 252, entrambi i metodi restituiscono **true** perché il prodotto in eccesso non è mai un problema per il consumatore.

Rimangono astratti i quattro metodi *getTabellaPercentuali*, *printTabellaPercentuali*, *getListaMisureInRange* e *getPercentualeMisureInRange*.

- la classe **WeightChecker** (da realizzare) specializza **Checker** per le misure di peso secondo i criteri di tolleranza espressi dall'enumerativo **Tolleranza**. Il costruttore riceve lo stesso elenco misure della classe base, a cui delega il popolamento delle strutture dati. I quattro metodi da concretizzare devono:

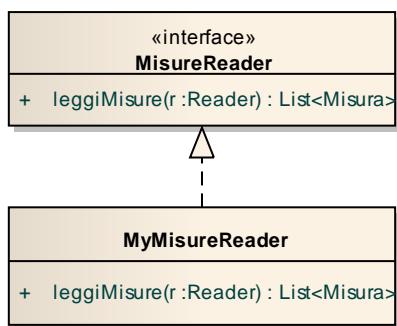
- `getTabellaPercentuali`: restituire una mappa <String, Double> che, per ogni prodotto, riporta la percentuale di confezioni che soddisfano il criterio di qualità;
- `printTabellaPercentuali`: restituire una rappresentazione in forma di stringa di tale mappa, *in cui le percentuali sono correttamente formattate come tali, con due cifre decimali*;
- `getListeMisureInRange(String descrizione)`: restituire la lista delle misure in range del prodotto specificato. [Suggerimento: fattorizzare la logica di verifica in un metodo ausiliario privato]
- `getPercentualeMisureInRange(String descrizione)` : restituisce la percentuale di misure in range del prodotto specificato.

Persistenza (qa.persistence)

(punti 4)

Il file di testo `Misure.txt` contiene una misura per riga: nell'ordine troviamo la descrizione del prodotto (chiave univoca), il peso nominale e quello reale, separati da virgole

```
Spicchi di luna, 250, 247
Spicchi di luna, 250, 245
Spicchi di luna, 250, 255
Rigatoncini, 500, 490
Rigatoncini, 500, 508
Rigatoncini, 500, 483
...
```



SEMANTICA:

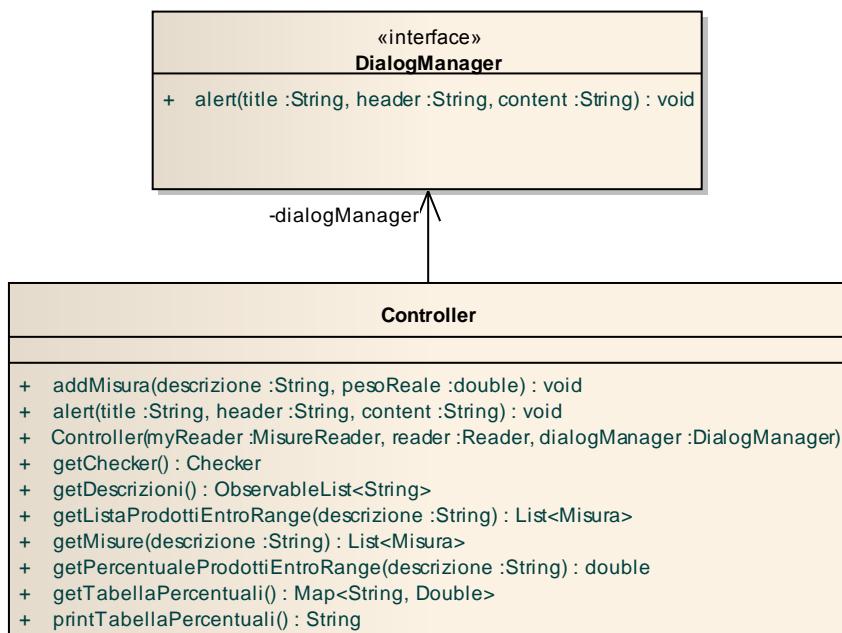
- L'interfaccia **MisureReader** (fornita) dichiara il metodo `leggiMisure` che restituisce una lista di **Misura** lette dal *reader* ricevuto come argomento; in caso di problemi di I/O, il metodo propaga l'opportuna **IOException**, mentre eventuali problemi nel formato del file sono incapsulati in **BadFormatException**.
- La classe **MyMisureReader** (**da realizzare**) implementa **MisureReader** secondo tali specifiche.

Parte 2

(punti: 12)

Controller (qa.ui.controller)

Il Controller è fornito già pronto ed è organizzato secondo il diagramma UML in figura.



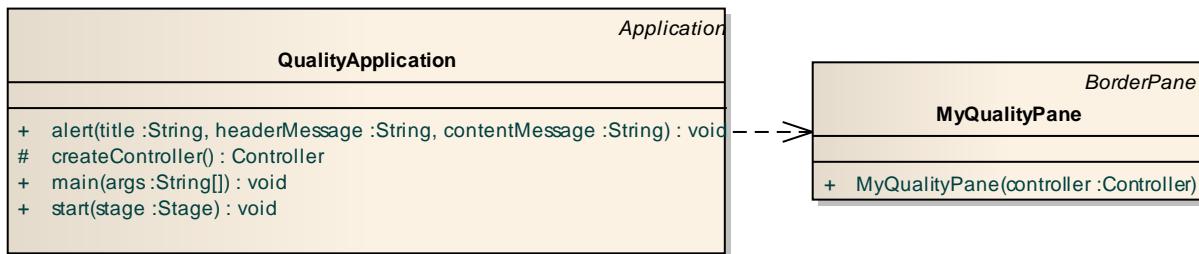
SEMANTICA:

La classe **Controller** (fornita) fornisce una serie di metodi che richiamano gli analoghi metodi del checker; in più, il metodo ausiliario `alert` può mostrare avvisi all'utente.

Il metodo `addMisura` [utile solo per la versione Swing completa] consente di aggiungere una misura alla tabella interna del checker, specificandone semplicemente descrizione e peso attuale (il peso atteso, uguale per tutti i prodotti dello stesso tipo, viene recuperato automaticamente).

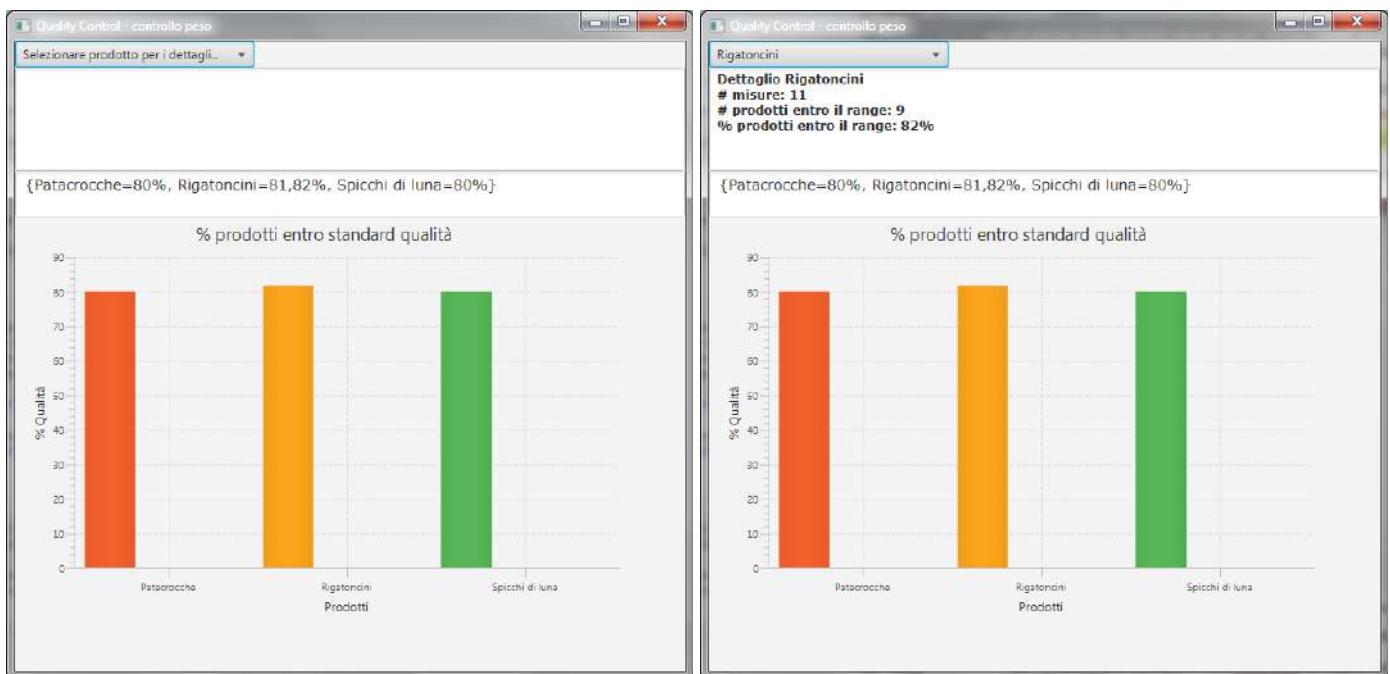
L'interfaccia utente deve essere simile (non necessariamente identica) all'esempio mostrato nelle figure seguenti.

L'architettura segue il modello sotto illustrato:



La classe **QualityApplication** (fornita) costituisce l'applicazione JavaFX che si occupa di aprire il file, il controller e incorporare l'apposita istanza di **MyQualityPane** (**da realizzare**). Per consentire di collaudare la GUI anche in assenza della parte di persistenza, è possibile avviare l'applicazione mediante la classe **QualityApplicationMock**.

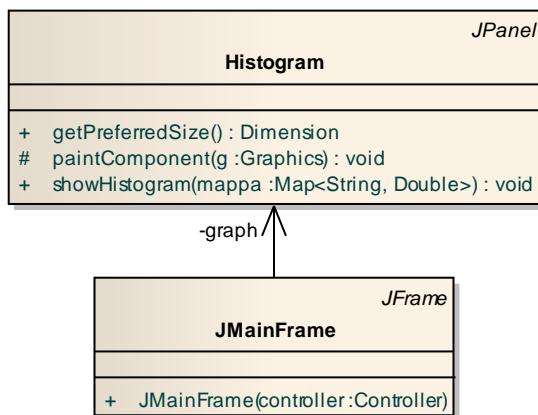
L'interfaccia utente deve essere simile (non necessariamente identica) all'esempio mostrato nella figura seguente.



La classe **MyQualityPane** (**da realizzare**) deve estendere **BorderPane**.

- 1) In alto, una **Combo** e due **TextArea** di opportuna dimensione consentono rispettivamente di scegliere quale prodotto approfondire, mostrarne il dettaglio, e visualizzare la tabella percentuali relativa al grafico mostrato sotto - *che non cambia nel tempo*.
- 2) Al centro un **BarChart** (senza legenda) mostra la distribuzione percentuale dei prodotti che superano il test di qualità, in forma grafica.
- 3) Quando l'utente seleziona un diverso elemento dalla combo, il dettaglio corrispondente nella prima textarea viene aggiornato.

In questo compito è offerta allo studente la possibilità di realizzare un'interfaccia grafica semplificata (7 punti) o completa (12 punti)



L'interfaccia utente semplificata deve essere simile all'esempio mostrato nella figura seguente:

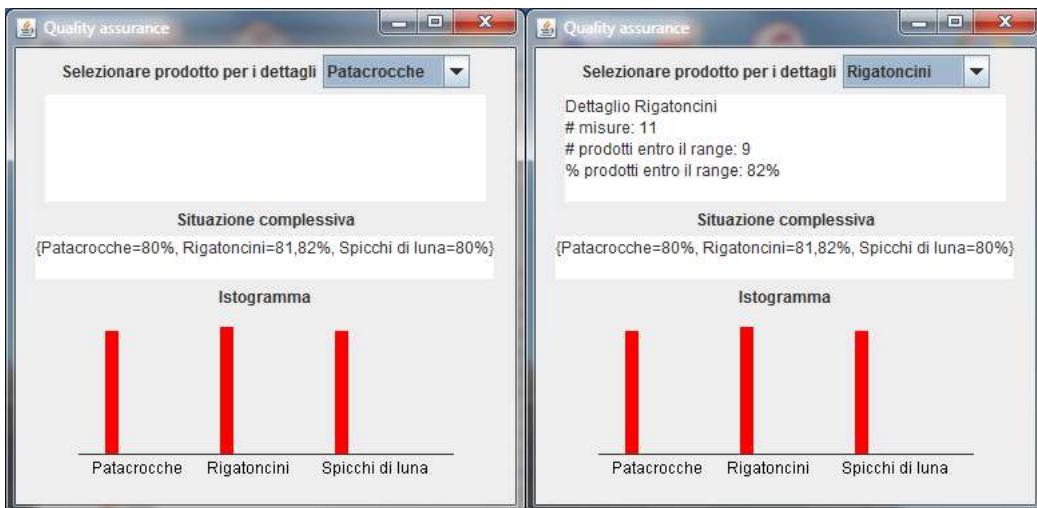


Fig. 1

Fig. 2

- La classe **Program** (fornita) contiene il *main* di partenza dell'intera applicazione, che si occupa di aprire il file e creare tutto il necessario. Per consentire di collaudare la GUI anche in assenza della parte di persistenza, è possibile avviare l'applicazione mediante la classe **GUITest**.
- La classe **Histogram** (fornita) implementa un pannello-istogramma, in grado di mostrare un grafico a barre; prevede il solo metodo pubblico `showHistogram` che riceve i dati da mostrare sotto forma di `mappa<String,Double>`.

La classe **JMainFrame (da realizzare)** deve organizzare l'interfaccia come sopra illustrato (Fig. 1), ovvero:

- 1) in alto, una label seguita dalla combo con tutte le descrizioni dei prodotti;
- 2) subito sotto, un'area di testo (non editabile) per mostrare i dettagli del prodotto;
- 3) più sotto, un'altra label seguita da una seconda area di testo (anch'essa non editabile) che mostri la situazione complessiva;
- 4) Infine, in basso, una label e di seguito l'istogramma (fisso) con la situazione complessiva.

A ogni selezione della combo (Fig. 2), l'applicazione deve reagire mostrando i dettagli del prodotto: in questa versione semplificata, la situazione complessiva e il grafico rimangono *immutati*.

L'interfaccia utente **completa** (valore: 12 punti) dev'essere invece come nell'esempio della figura seguente.

Rispetto alla GUI semplice, questa versione (Fig. 3) ha in più una riga di controlli al centro, costituita da un bottone (“*Nuova misura*”) e due campi di testo, che consentono di *aggiungere dinamicamente una nuova misura* non presente nel file, causando il corrispondente *l'aggiornamento dei dettagli, delle percentuali e del grafico*.

Inizialmente (Fig. 3), il bottone è abilitato mentre i due campi di testo sono disabilitati.

Premendo il pulsante (Fig. 4), il campo di testo più a destra si abilita, mentre nel primo viene ricoppiata l'attuale selezione della combo (nell'esempio, le Patacrocche) e il pulsante viene disabilitato. Ciò consente all'utente di inserire il peso (attuale) della nuova misura nel textfield attivo (Fig. 5), confermandola poi con INVIO sulla tastiera.

Tale conferma causa (Fig. 6) l'aggiunta immediata della nuova misura nel model (tramite il metodo *addMisura* del controller) e il conseguente ricalcolo dei dettagli, delle percentuali e del grafico. Contemporaneamente, il pulsante viene ri-abilitato, il campo di testo centrale svuotato e il campo di testo di destra ri-disabilitato, riportando così questa riga alla situazione iniziale.

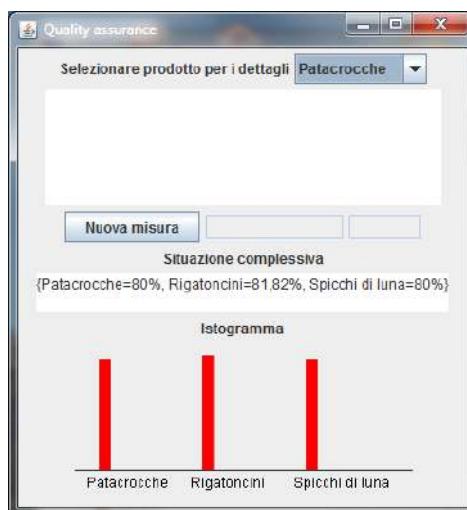


Fig. 3

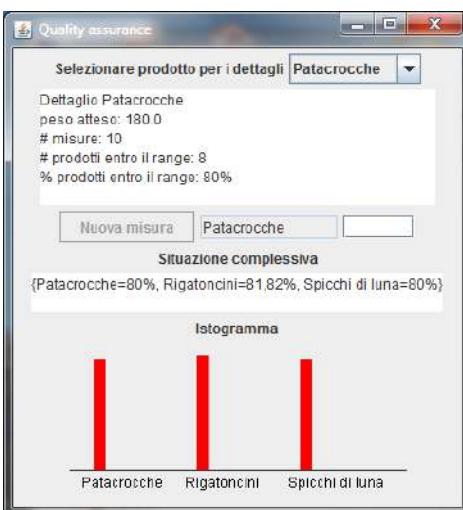


Fig. 4

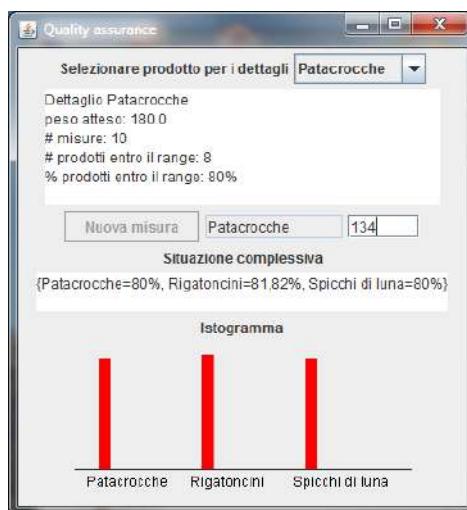


Fig. 5

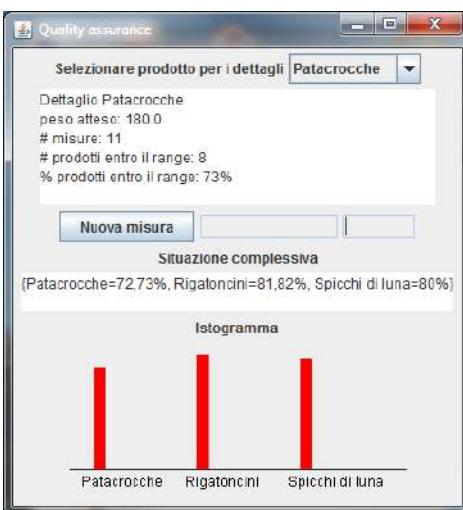


Fig. 6

ESAME DI FONDAMENTI DI INFORMATICA T-2 del 5/09/2017

Proff. E. Denti – G. Zannoni

Tempo a disposizione: 4 ore MAX

NB: il candidato troverà nell'archivio ZIP scaricato da Esamix anche il software "Start Kit"

NOME PROGETTO ECLIPSE e CARTELLA: CognomeNome-matricola (es. RossiMario-0000123456)

NOME ZIP DA CONSEGNARE : CognomeNome-matricola.zip (es. RossiMario-0000123456.zip)

La nota azienda “ED Creams & Dreams” leader nella produzione del gelato artigianale nella prospera EDLandia desidera offrire alle sue gelaterie l’applicazione “EDIceCream” che permetta la gestione di ciascun negozio. “ED Creams & Dreams” è suddivisa in un centro di produzione e svariate gelaterie che si occupano della vendita al dettaglio.

L’applicazione “EDIceCream” dovrà permettere di:

- leggere da file le scorte di gelato ricevute ogni giorno dal centro di produzione;
- gestire le scorte ricevute;
- definire ed inserire ogni gelato venduto
- visualizzare a video lo stato corrente della gelateria (gelati venduti, incasso, ecc...);
- stampare su file i resi giornalieri

DESCRIZIONE DEL DOMINIO DEL PROBLEMA

“ED Creams & Dreams” desidera offrire alle sue gelaterie una applicazione per migliorare la gestione di ogni singolo punto vendita. Ogni mattina ciascun punto vendita riceve la fornitura giornaliera di gelato dal centro di produzione. Tale fornitura è memorizzata nel file [IceCreamQuantity.txt](#), dove per ogni gusto di gelato viene indicato il peso in grammi che è stato consegnato. Inoltre, al fine di mantenere la coerenza tra i diversi punti vendita, “ED Creams & Dreams” fornisce il file [Kinds.txt](#) che dettaglia le diverse tipologie di gelato, ad esempio:

- Cono piccolo: prezzo 2,50 euro, gusti massimi 2, peso totale del gelato 60gr

L’applicazione quindi deve permettere al venditore di comporre ogni singolo gelato partendo dalla specifica del tipo di gelato (*kind*) e successivamente scegliere i diversi gusti senza violare il numero massimo di gusti possibili per la specifica tipologia. Prima della vendita di ogni gelato occorre verificare che siano disponibili tutti i gusti, ovvero che per ogni gusto sia presente il quantitativo in grammi specifico per la tipologia di gelato, ad esempio:

Cono piccolo crema e cioccolato → va verificato che siano disponibili 30 gr sia di crema, sia di cioccolato

Cono piccolo cioccolato → va verificato che siano disponibili 60 gr di cioccolato

N.B. il gelato può avere meno gusti di quelli indicati nella tipologia!

Se non è possibile erogare il gelato deve comparire un messaggio a video. Dopo la vendita del gelato è necessario aggiornare la fornitura di gelato scalando i quantitativi venduti per ogni gusto. Ad ogni vendita “EDIceCream” deve provvedere ad aggiornare le informazioni a video relative al numero totale dei gelati venduti, i quantitativi per ogni singolo gusto e il totale incassato. A fine giornata inoltre è necessario stampare sul file “[StockGiornaliero.txt](#)” i quantitativi resi per ogni singolo gusto.

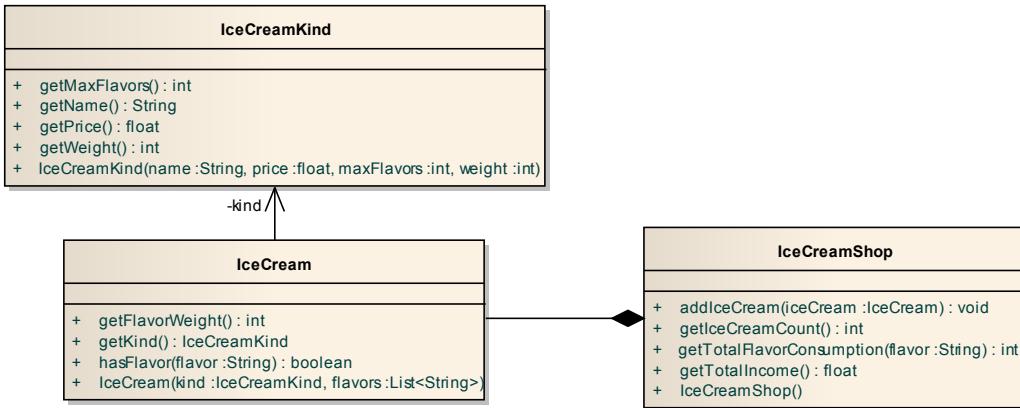
Parte 1

(punti: 15)

Dati (namespace [edicecream.model](#))

(punti: 5)

Il modello dei dati deve essere organizzato secondo il diagramma UML più sotto riportato.



SEMANTICA:

- La classe ***IceCream*** (fornita nello Start kit) rappresenta i gelati venduti: il metodo ***getKind*** permette di recuperare il tipo di gelato (***IceCreamKind***), il metodo ***hasFlavor*** permette di verificare se il gelato è composto da uno specifico gusto, ed infine il metodo ***getFlavorWeight*** restituisce il peso in grammi di ogni gusto
- La classe ***IceCreamKind*** (fornita nello Start kit) rappresenta le diverse tipologie di gelato in termini di nome, prezzo, massimo numero di gusti e peso totale del gelato. Sono messi a disposizione i necessari metodi accessor per recuperare questi valori
- La classe ***IceCreamShop*** (da realizzare) rappresenta la collezione dei gelati venduti e offre le seguenti funzionalità:
 - ***addIceCream(IceCream icecream)***: inserisce un nuovo gelato;
 - ***getTotalIncome()***: restituisce il totale incassato;
 - ***getTotalFlavorConsumption(String flavor)***: restituisce un intero che rappresenta il consumo totale in grammi per il gusto che viene passato in ingresso;
 - ***getIceCreamCount()***: restituisce il numero totale di gelati venduti.

Lo Start Kit contiene anche i test (da includere nel progetto) per verificare il funzionamento di queste classi.

Persistenza (*edcream.persistence*)

(punti 10)

Come già anticipato, il file di testo ***IceCreamQuantity.txt*** contiene l'elenco di tutti i gusti di gelato consegnati con l'indicazione del corrispondente quantitativo.

Il file ***IceCreamQuantity.txt*** contiene una riga per ogni gusto, nella forma:

<i>NomeGusto - Quantità</i>

dove

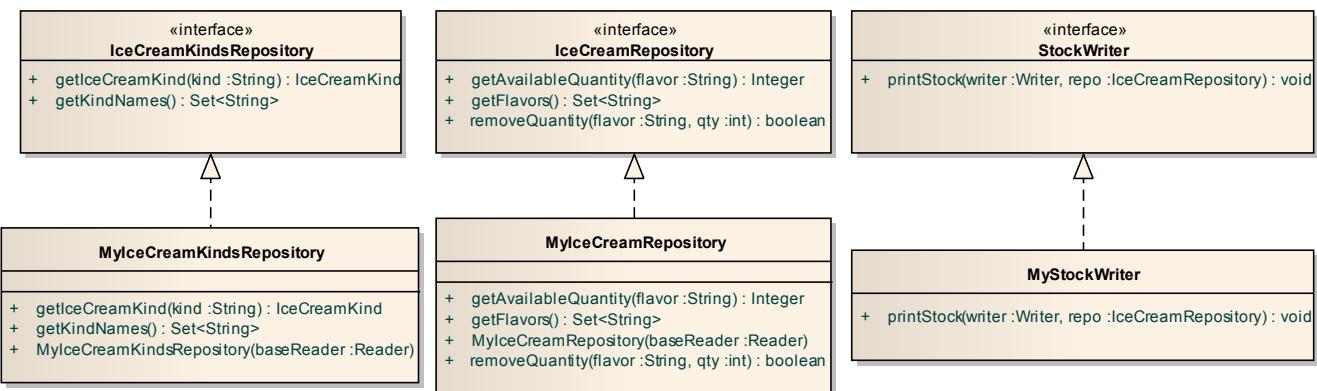
- *NomeGusto* rappresenta il nome del gusto del gelato: può contenere spazi in caso di nomi composti
- Zero o più spazi
- Un carattere ‘-’
- Zero o più spazi
- *Quantità* rappresenta il peso totale in grammi del gelato

Il file ***kinds.txt*** contiene una riga per ogni tipologia di gelato, nella forma:

<i>NomeTipologia - Prezzo - NumGusti - Peso</i>

dove

- *NomeTipologia* è il nome del tipo di gelato: può contenere spazi in caso di nomi composti
- Zero o più spazi
- Un carattere ‘-’
- *Prezzo* è un numero reale che rappresenta il prezzo del gelato in formato **italiano** (quindi con la virgola a separare la parte decimale e il ‘.’ come separatore delle migliaia)
- Zero o più spazi
- Un carattere ‘-’
- *NumGusti* è un valore intero che indica il numero massimo di gusti per la tipologia del gelato
- Zero o più spazi
- Un carattere ‘-’
- *Peso* è un valore intero che rappresenta il peso massimo in grammi del gelato



SEMANTICA:

- L'interfaccia ***IceCreamRepository*** (fornita nello Start kit) dichiara i metodi messi a disposizione dal repository che gestisce le scorte di gelato.
- La classe ***MyIceCreamRepository*** (da realizzare) implementa ***IceCreamRepository***:
 - riceve in ingresso nel costruttore il Reader da cui leggere i dati relativi alle scorte di gelato (se il parametro è *null* lancia una **IllegalArgumentException**). Il costruttore recupera tutti i diversi gusti ed il loro peso e li memorizza in una opportuna struttura dati che permetta **una ricerca veloce dei gusti per nome**. In caso di problemi nella lettura del file, si lascia uscire la **IOException** se i problemi sono relativi al file, mentre viene lanciata una **BadFormatException** (fornita nello Start kit) se i problemi sono relativi ai dati;
 - getFlavors*** restituisce un **Set<String>** che contiene i *nomi* di tutti i gusti;
 - getAvailableQuantity(String flavor)*** restituisce un intero che rappresenta la quantità in grammi ancora disponibili del gusto che viene passato in ingresso;
 - removeQuantity(String flavor, int quantity)*** riceve in ingresso il nome del gusto e il quantitativo di grammi da rimuovere ed aggiorna le scorte .
- L'interfaccia ***IceCreamKindsRepository*** (fornita nello Start kit) dichiara i metodi messi a disposizione dal repository che gestisce le diverse tipologie di gelato.
- La classe ***MyIceCreamKindsRepository*** (da realizzare) implementa ***IceCreamKindsRepository***:
 - riceve in ingresso nel costruttore il Reader da cui leggere i dati relativi alle diverse tipologie di gelato (se il parametro è *null* lancia una **IllegalArgumentException**). Il costruttore recupera tutti i diversi gusti ed il loro peso e li memorizza in una opportuna struttura dati che permetta **una ricerca veloce delle tipologie per nome**. In caso di problemi nella lettura del file, si lascia uscire la **IOException** se i problemi sono

relativi al file, mentre viene lanciata una **BadFormatException** (fornita nello Start kit) se i problemi sono relativi ai dati.

- **getKindNames** restituisce un **Set<String>** contenente i nomi delle diverse tipologie di gelato
 - **getIceCreamKind(String name)** restituisce un oggetto di tipo **IceCreamKind** che rappresenta la tipologia di gelato relativa al nome passato come argomento
- e) L'interfaccia **StockWriter** (fornita nello Start kit) dichiara i metodi messi a disposizione per la scrittura del file **StockGiornaliero.txt**
- f) La classe **MyStockWriter** (fornita nello Start kit) implementa **StockWriter** e mette a disposizione il metodo **printStock(Writer write, IceCreamRepository repo)** che riceve in ingresso il Writer su cui scrivere i dati e il gestore delle scorte di gelato rimaste ed esegue la scrittura su file.

Lo Start Kit contiene anche i test (da includere nel progetto) per verificare il funzionamento di queste classi.

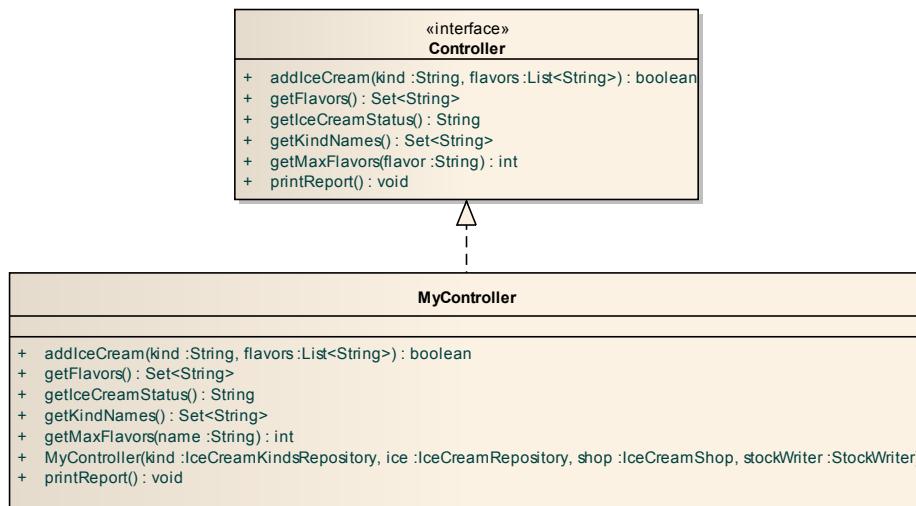
Parte 2

(punti: 15)

Controller (edicecream.controller)

(punti 7)

Il Controller deve essere organizzato secondo il diagramma UML più sotto riportato.



SEMANTICA:

La classe **MyController (da realizzare)** implementa l'interfaccia **Controller** (fornita nello Start Kit), in particolare:

- il costruttore deve prevedere quattro argomenti di tipo **IceCreamKindsRepository**, **IceCreamRepository**, **IceCreamShop** e **StockWriter**
- il metodo **getFlavors** restituisce la lista ordinata di tutti i gusti di gelato a disposizione;
- il metodo **getKindNames** restituisce la lista ordinata di tutte le diverse tipologie di gelato;
- il metodo **addIceCream(String kind, List<String> flavor)** riceve in ingresso il nome della tipologia del gelato che si sta inserendo e la lista dei gusti richiesti, restituisce **true** nel caso sia possibile creare il gelato, **false** in caso contrario. È possibile inserire un nuovo gelato solo se sono soddisfatte due condizioni:
 - il numero dei gusti richiesti è inferiore o uguale al numero dei gusti massimi indicati nella tipologia;
 - per ogni gusto è disponibile nelle scorte il quantitativo di gelato necessario (N.B. **IceCreamKind** fornisce il peso massimo in grammi del gelato, non il peso per ogni gusto: quest'ultimo valore viene calcolato dalla classe **IceCream**);

- il metodo ***printReport()*** che gestisce la stampa dei resi mediante l'uso di ***StockWriter*** e aprendo/chiudendo il file necessario. Tale metodo deve gestire eventuali errori di I/O;
- il metodo ***getMaxFlavors(String name)*** che restituisce il numero massimo di gusti relativi al nome della tipologia di gelato passato in ingresso;
- il metodo ***getIceCreamStatus()*** che restituisce una stringa contenente lo stato attuale dei gelati. La stringa deve essere della forma:

```

Totale gelati venduti: num
Venduti x gr di gusto1
Venduti y gr di gusto2
Totale incasso: xyz euro

```

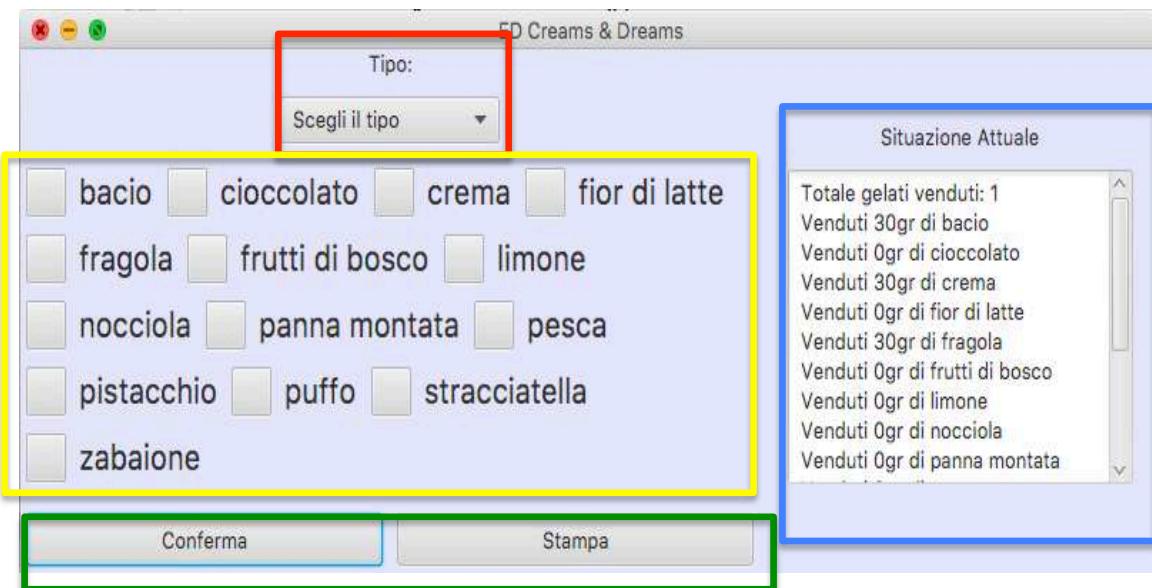
Interfaccia Utente

(punti 8)

Studenti A.A. 2016/17 – versione con JavaFX (package: *edicecream.ui.javafx*)

La classe ***EDIceCreamApplication*** (fornita) costituisce l'applicazione JavaFX che si occupa di aprire i file, creare il controller e incorporare l'***IceCreamPane*** (da realizzare). Per consentire di collaudare la GUI anche in assenza della parte di persistenza, è possibile avviare l'applicazione mediante la classe ***EDIceCreamApplicationMock***.

L'interfaccia utente deve essere simile (non necessariamente identica) all'esempio mostrato nella figura seguente:



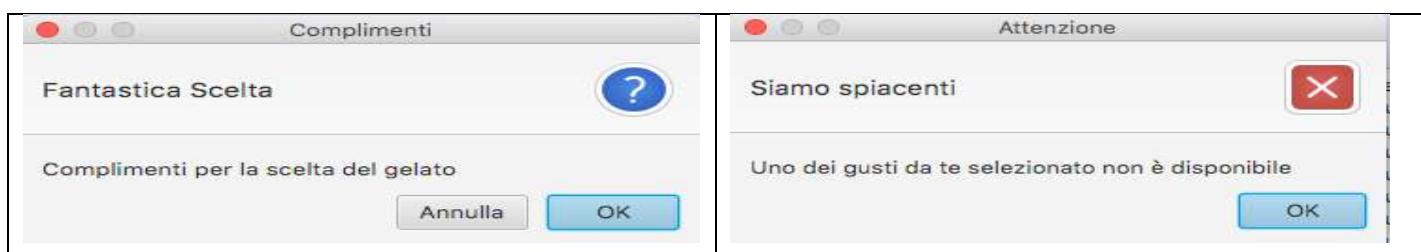
La classe ***IceCreamPane*** (da realizzare) deve estendere ***BorderPane***.

La GUI è organizzata in quattro sezioni:

- 1) Sezione inserimento tipologia gelato (parte sinistra in alto, evidenziata in rosso):

- Una Label

- una **ComboBox** pre-popolata con i diversi tipi di gelato. Attenzione che quando cambia la tipologia di gelato va cambiato il valore del massimo numero di gusti selezionabili nel **FlavorPanel** (vedi dopo). All'avvio e dopo l'inserimento di ogni gelato la **ComboBox** deve presentare un *hint* con la scritta "Scegli il tipo" (usare il metodo **setPromptText** per settare il valore all'avvio).
- 2) Sezione scelta dei gusti (parte sinistra centrale, evidenziata in giallo):
- Un **FlavorPane** (fornito nello Start kit) che permette la selezione dei gusti del gelato. Se si tenta di inserire un numero di gusti maggiore rispetto a quello massimo viene visualizzato un messaggio di errore. Tale classe fornisce le seguenti funzionalità:
 - Costruttore: riceve in ingresso un insieme di stringhe che rappresentano i diversi gusti del gelato. Il costruttore imposta a 0 il massimo numero di gusti selezionabili in modo tale che non sia possibile selezionare gusti finché non è stato selezionato il tipo di gelato.
 - **setMaxSelected** permette di impostare il massimo numero di gusti selezionabili.
 - **getSelected** restituisce una *List* dei gusti selezionati
 - **reset** permette di resettare il pannello deselezionando tutte le **checkbox**
- 3) Sezione risultato (parte destra, evidenziata in blu)
- **Label** che specifica il contenuto della **TextArea**
 - **TextArea** che si aggiorna ad ogni inserimento di gelato.
- 4) Sezione pulsanti (parte sinistra in basso, evidenziata in verde):
- Un **Button** per la conferma del gelato da inserire. La pressione di tale bottone deve mostrare a video una **Alert** con il risultato dell'inserimento del gelato come indicato nelle figure sottostanti e deve aggiornare il testo della **TextArea** che rappresenta lo status attuale della gelateria: totale gelati venduti, grammi venduti per ogni gusto e totale incasso.

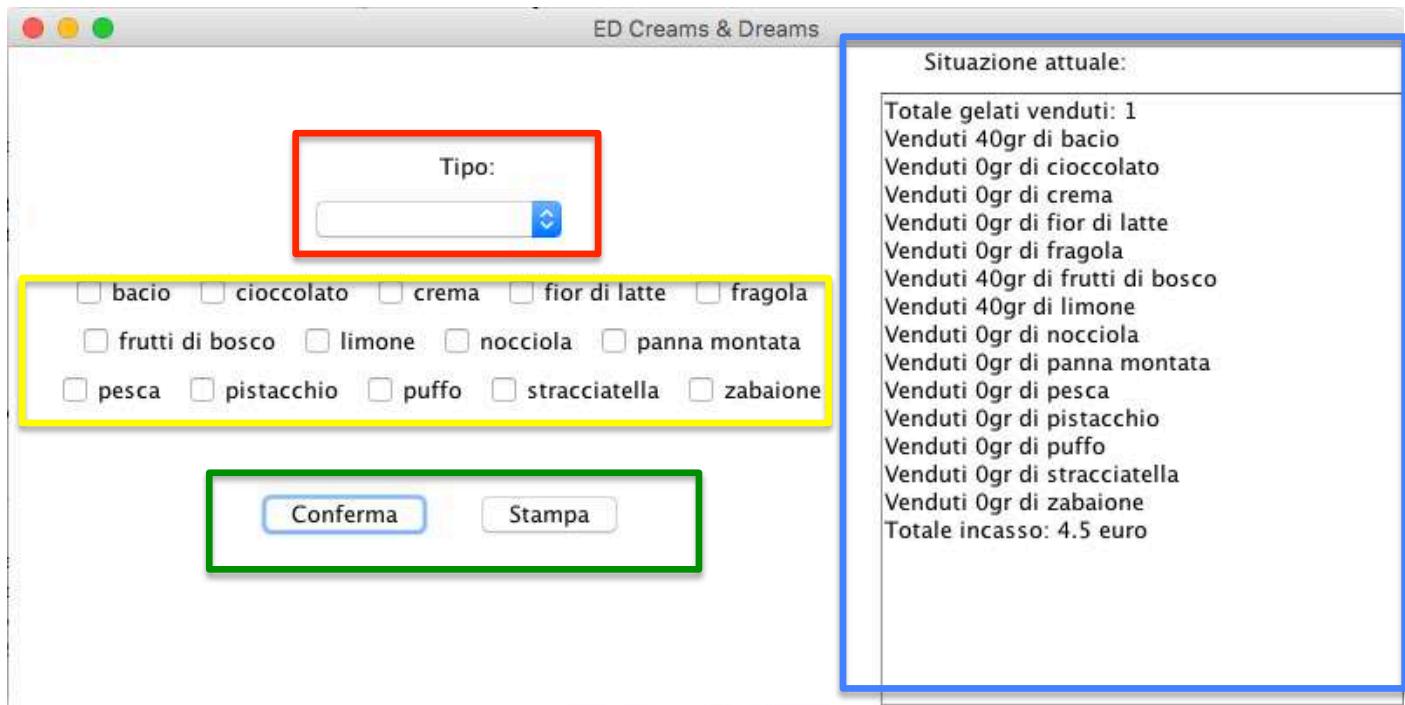


- Un **Button** stampa la cui pressione deve permettere di stampare su file il reso giornaliero.

Studenti A.A. 2015/16 (e precedenti) – versione con Swing (package: zannofantasyfootball.ui.swing)

La classe **EDIceCreamMain** (fornita) costituisce il main per Swing che si occupa di aprire i file, creare il controller e incorporare il **IceCreamFrame** (da realizzare). Per consentire di collaudare la GUI anche in assenza della parte di persistenza, è possibile avviare l'applicazione mediante la classe **GUITest**.

L'interfaccia utente deve essere simile (non necessariamente identica) all'esempio mostrato nella figura seguente:



La classe ***IceCreamFrame*** (da realizzare) deve estendere ***JFrame***.

La GUI è organizzata in quattro sezioni:

- 5) Sezione inserimento tipologia gelato (parte sinistra in alto, evidenziata in rosso):
 - Una ***JLabel***
 - una ***JComboBox*** pre-popolata con i diversi tipi di gusti. Attenzione che quando cambia la tipologia di gelato va cambiato il valore del massimo numero di gusti selezionabili nel ***FlavorPanel*** (vedi dopo).
- 6) Sezione scelta dei gusti (parte sinistra centrale, evidenziata in giallo):
 - Un ***FlavorPanel*** (fornito nello Start kit) che permette la selezione dei gusti del gelato. Se si tenta di inserire un numero di gusti maggiore rispetto a quello massimo viene visualizzato un messaggio di errore. Tale classe fornisce le seguenti funzionalità:
 - Costruttore riceve in ingresso un insieme di stringhe che rappresentano i diversi gusti del gelato. Il costruttore imposta a 0 il massimo numero di gusti selezionabili in modo tale che non sia possibile selezionare gusti finché non è stato selezionato il tipo di gelato.
 - ***setMaxSelected*** permette di impostare il massimo numero di gusti selezionabili
 - ***getSelected*** restituisce un ***List*** dei gusti selezionati
 - ***reset*** permette di resettare il pannello deselezionando tutte le checkbox
- 7) Sezione risultato (parte destra, evidenziata in blu)
 - ***JLabel*** che specifica il contenuto della ***JTextArea***
 - ***JTextArea*** che si aggiorna ad ogni inserimento di gelato.
- 8) Sezione pulsanti (parte sinistra in basso, evidenziata in verde):
 - Un ***JButton*** per la conferma del gelato da inserire. La pressione di tale bottone deve mostrare a video un JOptionPane con il risultato dell'inserimento del gelato come indicato nelle figure sottostanti e deve aggiornare il testo della ***JTextArea*** che rappresenta lo status attuale della gelateria: totale gelati venduti, grammi venduti per ogni gusto e totale incasso.



- Un **JButton** stampa la cui pressione deve permettere di stampare su file il reso giornaliero