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 capilinea 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:

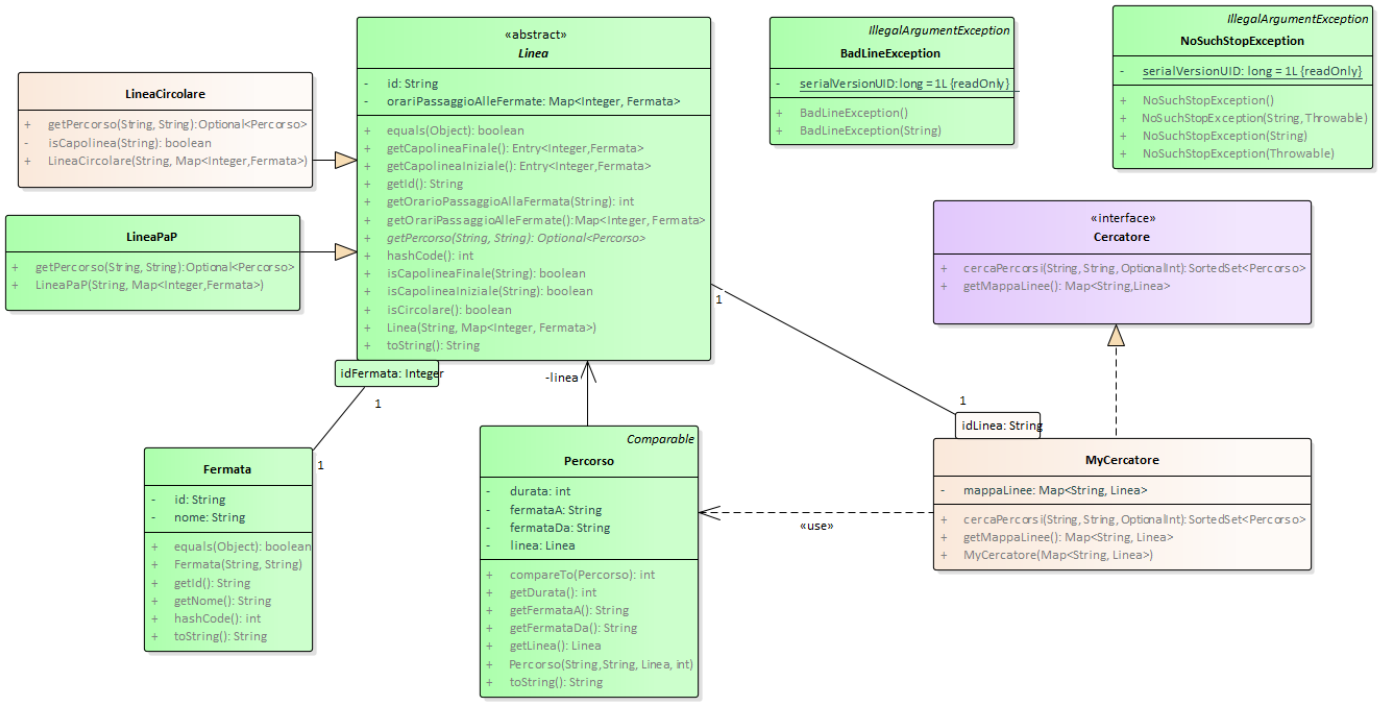
- per le linee da punto a punto (PaP), la differenza (>0) tra gli orari di passaggio alle due fermate Y e X
- per le linee circolari, a seconda della posizione relativa delle due fermate rispetto al capolinea, o come sopra, o (se la differenza è negativa) come somma dei due sotto-tragitti da X al capolinea 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.

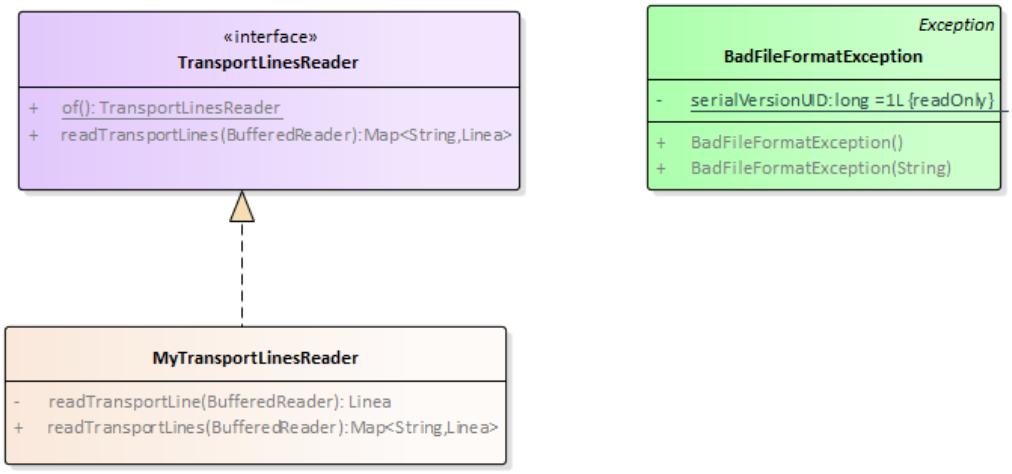
Persistenza (package bussy.persistence) *(punti 8)*

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 virgole), separati da virgole (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 ***BadFileFormatException*** 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).

Parte 2 (punti: 7)

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**.

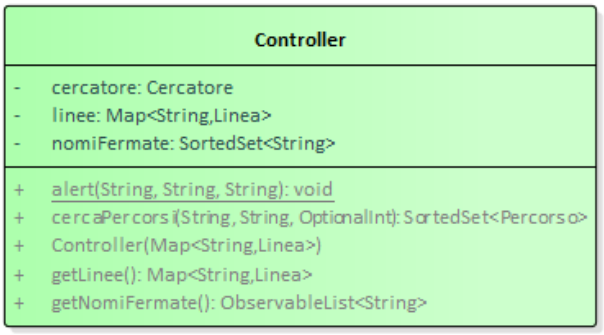
In alto, due combo di stringhe sono popolate con i nomi di tutte le fermate, in ordine alfabetico (Figg. 1,2,3): inizialmente non è selezionata alcuna fermata. Avvicinandosi alla combo, appositi tooltip (non mostrati) invitano a scegliere una fermata. Dopo aver scelto le due fermate (Figg. 4 e seguenti), 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)*

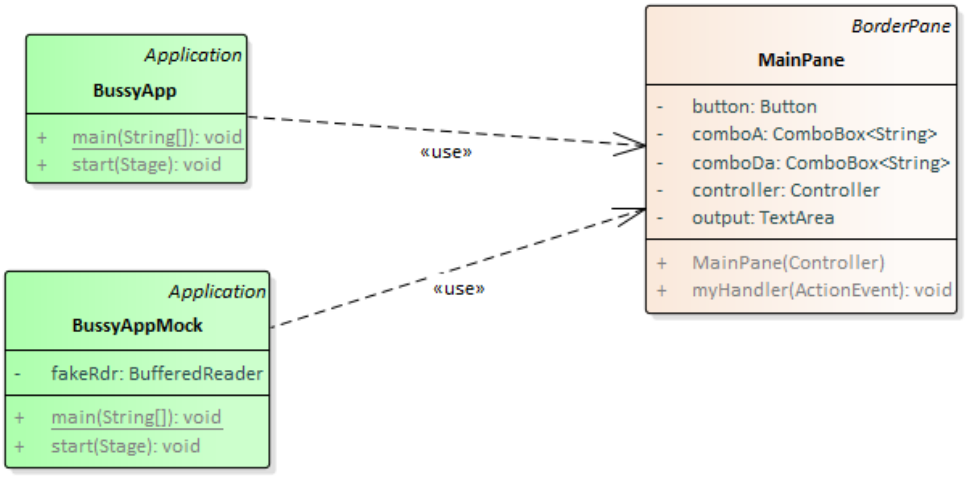
Il ***Controller*** è fornito già implementato:

- 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

a) 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

b) 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"?

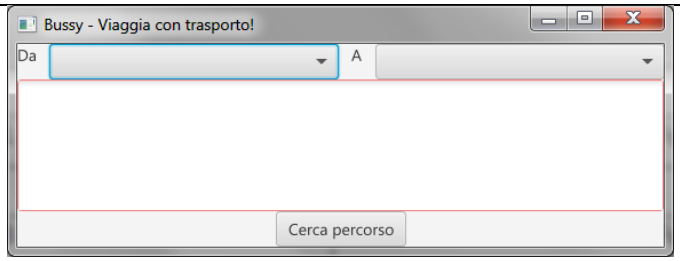


Fig. 1

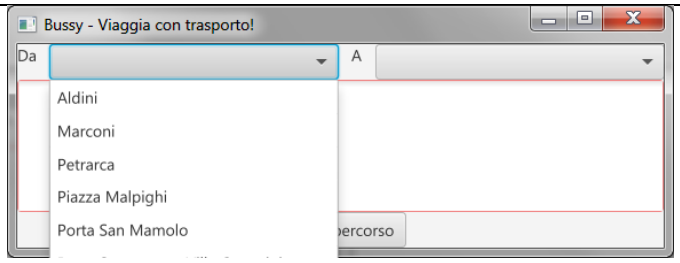


Fig. 2

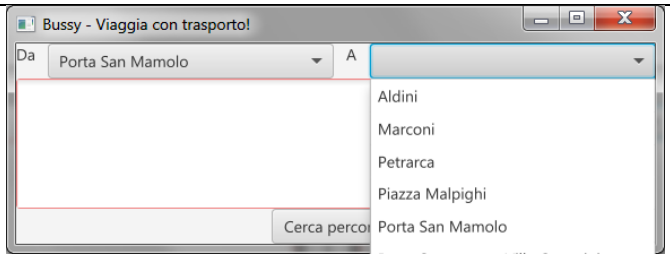


Fig. 3

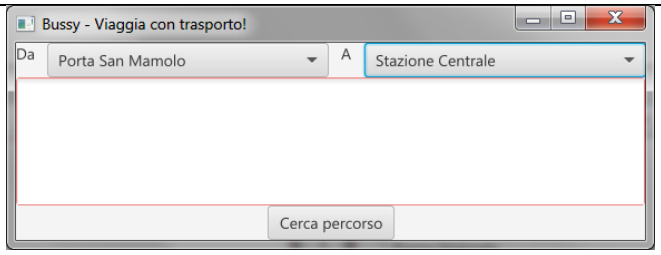


Fig. 4

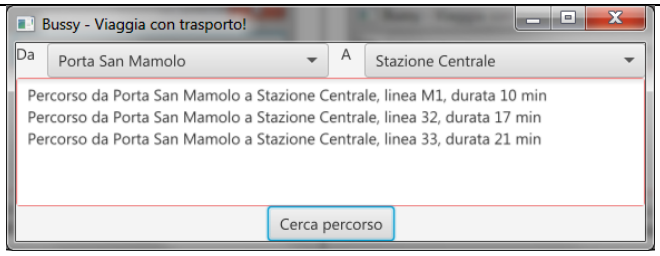


Fig. 5

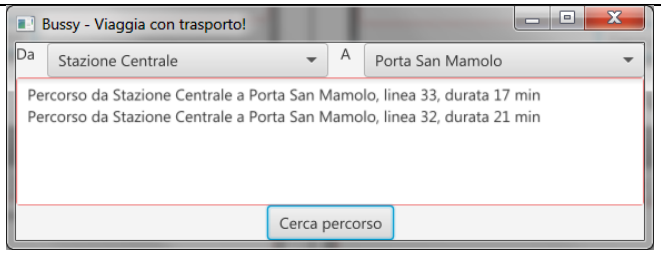


Fig. 6

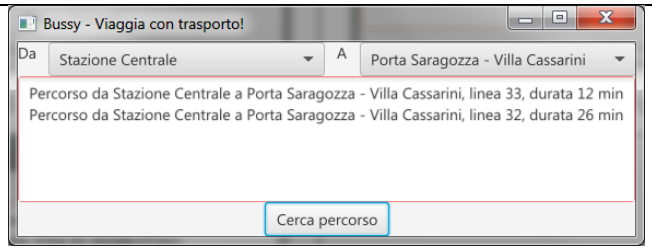


Fig. 7

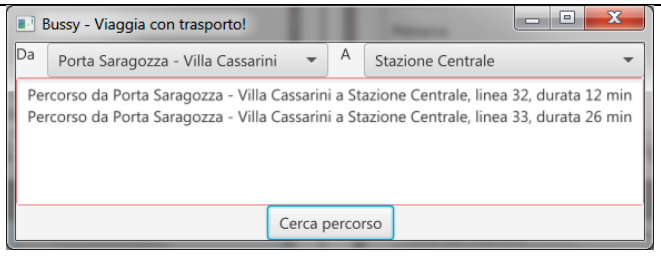


Fig. 8

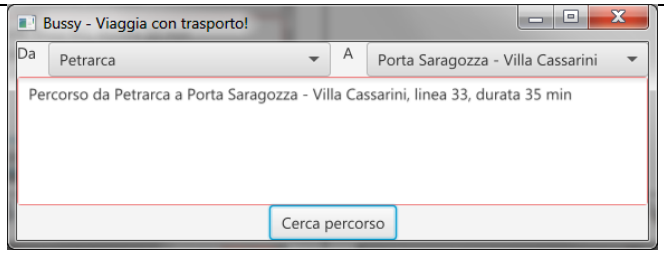


Fig. 9

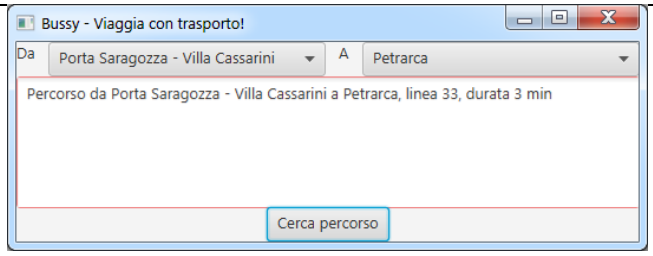


Fig. 10

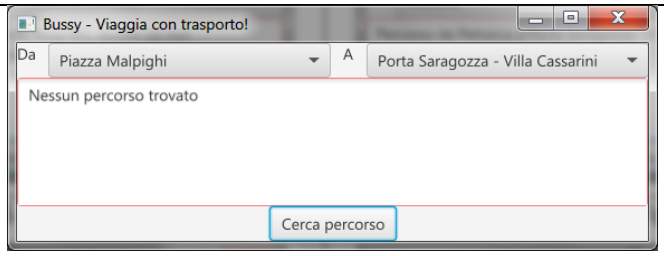


Fig. 11