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 condomìni 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:

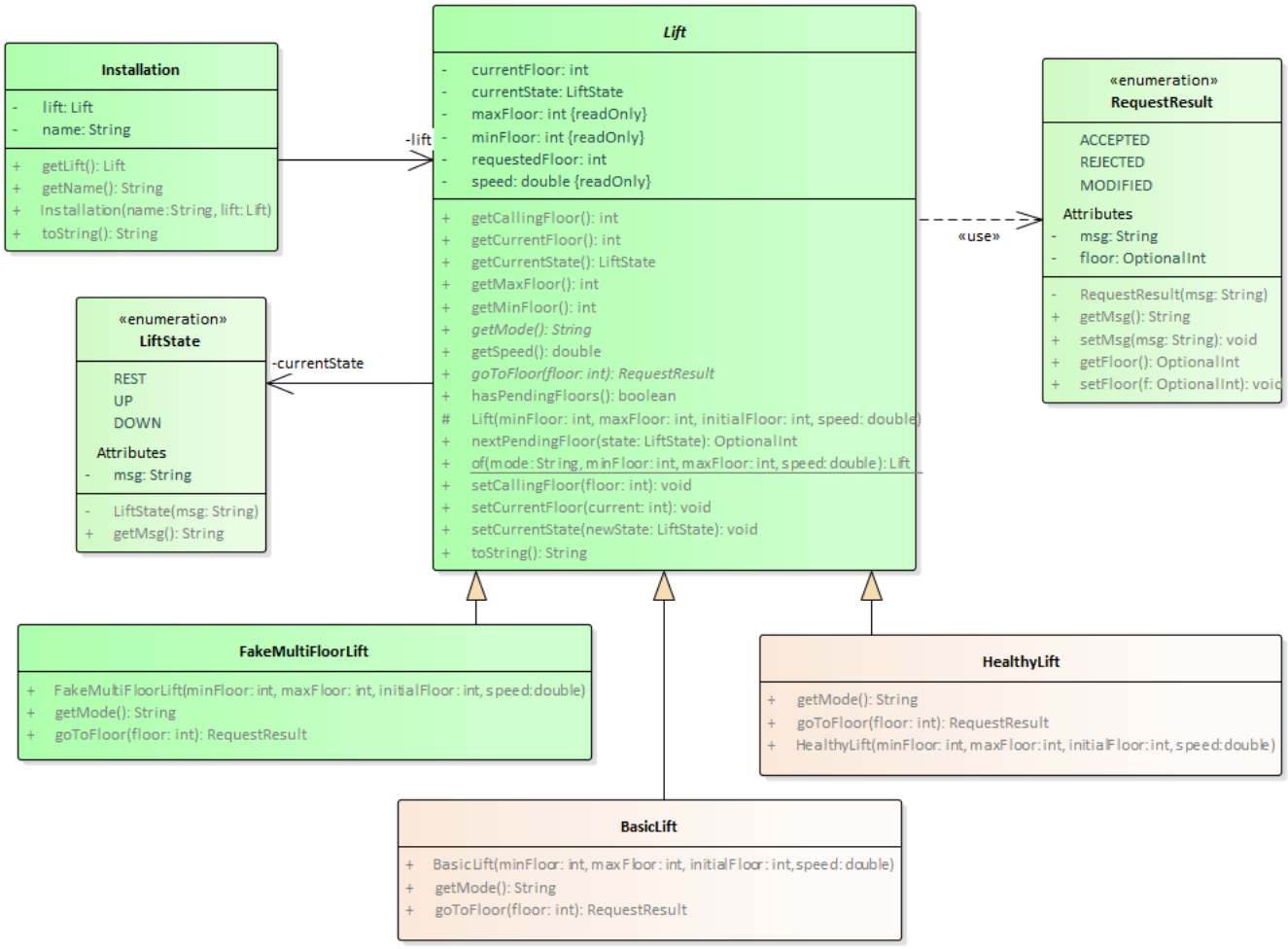
- il nome dell'edificio in cui è installato
- la modalità di funzionamento (base, multipiano, salutista)
- il numero di piani e loro numerazione (es. 6 piani da -1 a 4, 5 piani da -2 a 2, 4 piani da 0 a 3, etc.)
- la velocità dell'ascensore in metri/secondo (non utilizzata però nell'algoritmica di questo compito)

TEMPO STIMATO PER SVOLGERE L'INTERO COMPITO: 1h40 – 2h15

Parte 1 — punti: 19

Modello dei dati (package edlift.model) — [TEMPO STIMATO: 23-35 minuti] — *punti: 9*

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 accessor 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 accessor 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'accessor 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 mssaggio 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

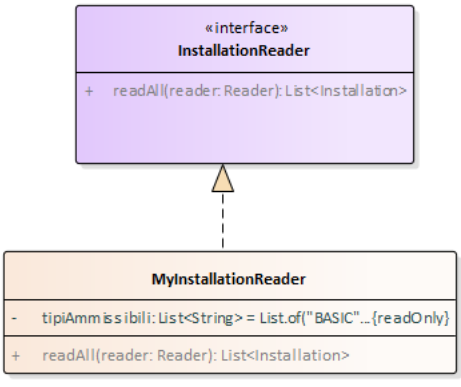
```
ASCENSORE HOTEL MIRALAGO
TIPO MULTI A 7 PIANI da -2 a 4
VELOCITA 1 m/s

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:

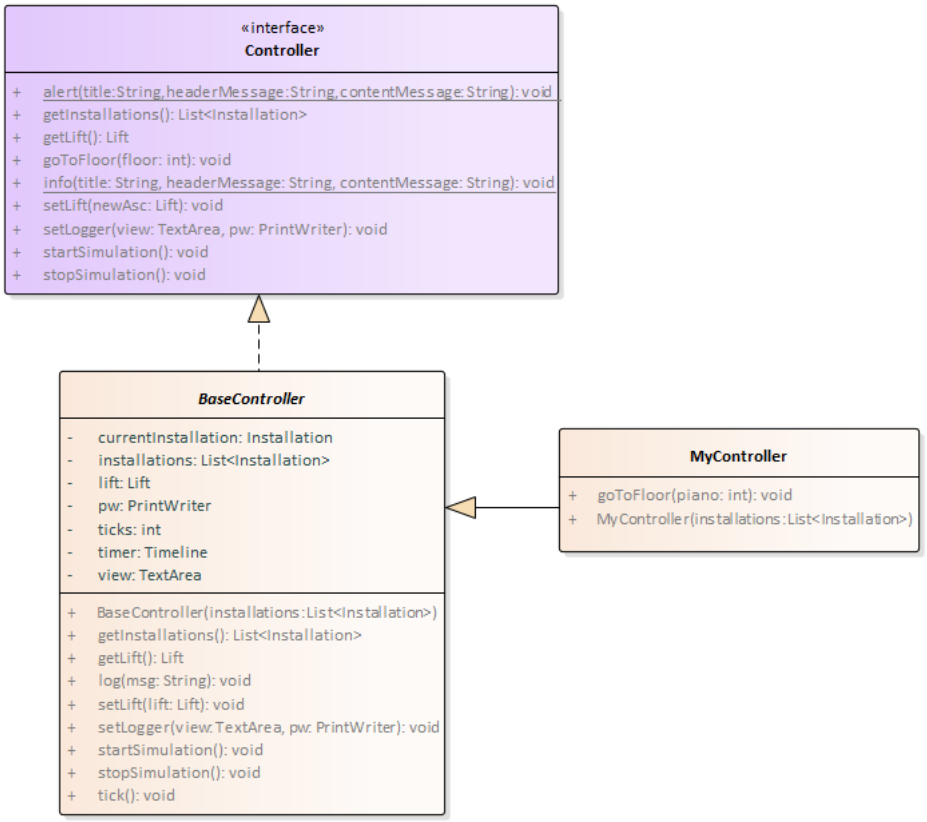
a) l'interfaccia ***InstallationReader*** (fornita) dichiara il metodo ***readAll*** che restituisce una lista di ***Installation***, lanciando, oltre alla "naturale" ***IOException***, una ***BadFileFormatException*** nel caso di errore nel formato del file;

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, ***BadFileFormatException*** 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.

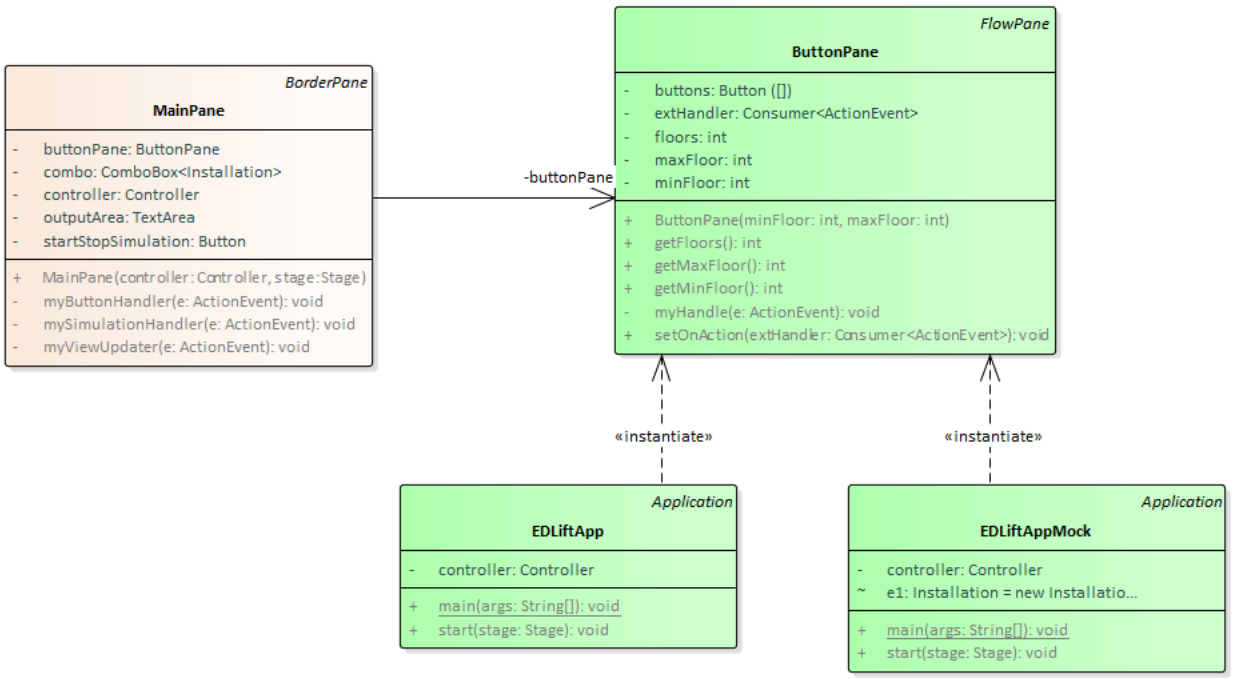
Interfaccia utente (package edlift.ui) **[TEMPO STIMATO: 40-50 minuti]** ***punti: 11***

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 bottoniera (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:

a) 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

b) La classe ***EDLiftApp*** (e la sua consorella ***-Mock***) costituisce l'entry point dell'applicazione.

c) 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 bottoniera, da collocare in basso **Quando si cambia ascensore la view dev'essere aggiornata, nel seguente modo:**
  - sostituendo la bottoniera 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 bottoniera

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?

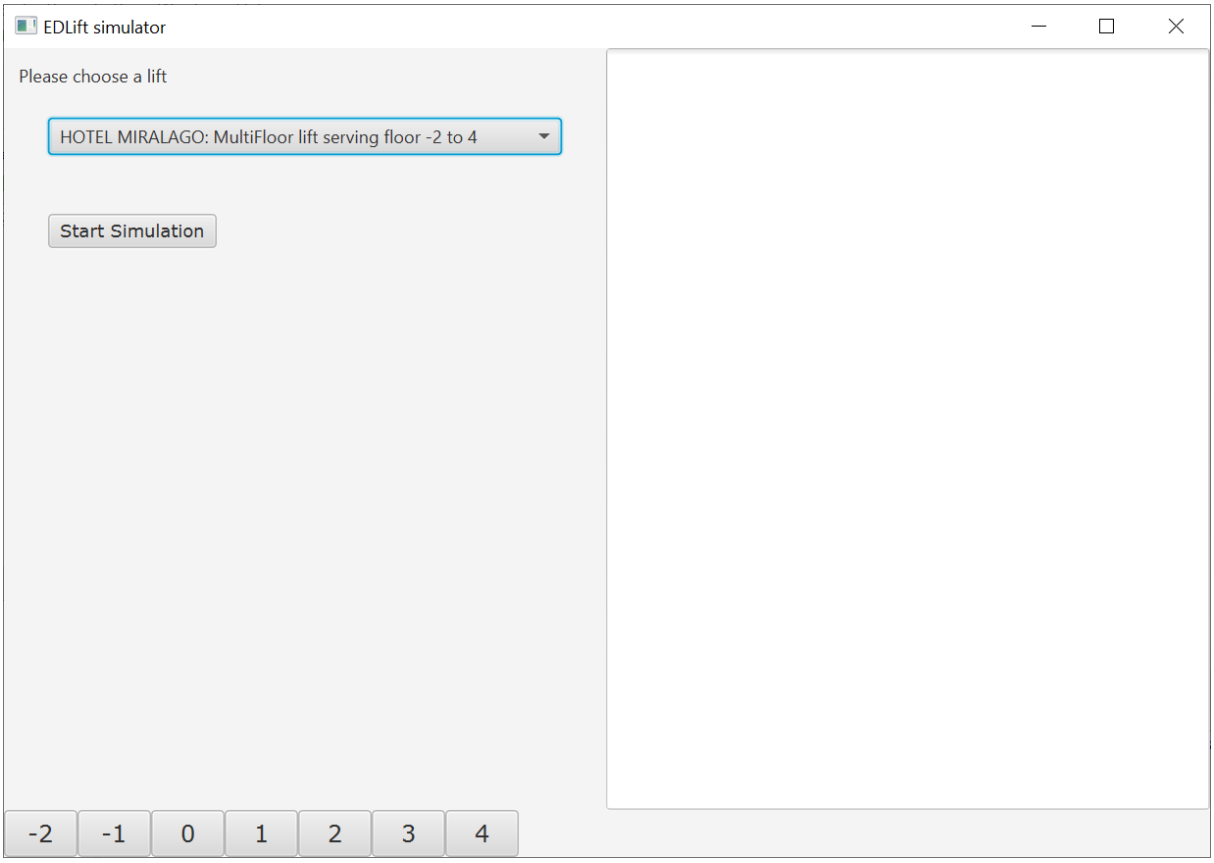


Figura 1

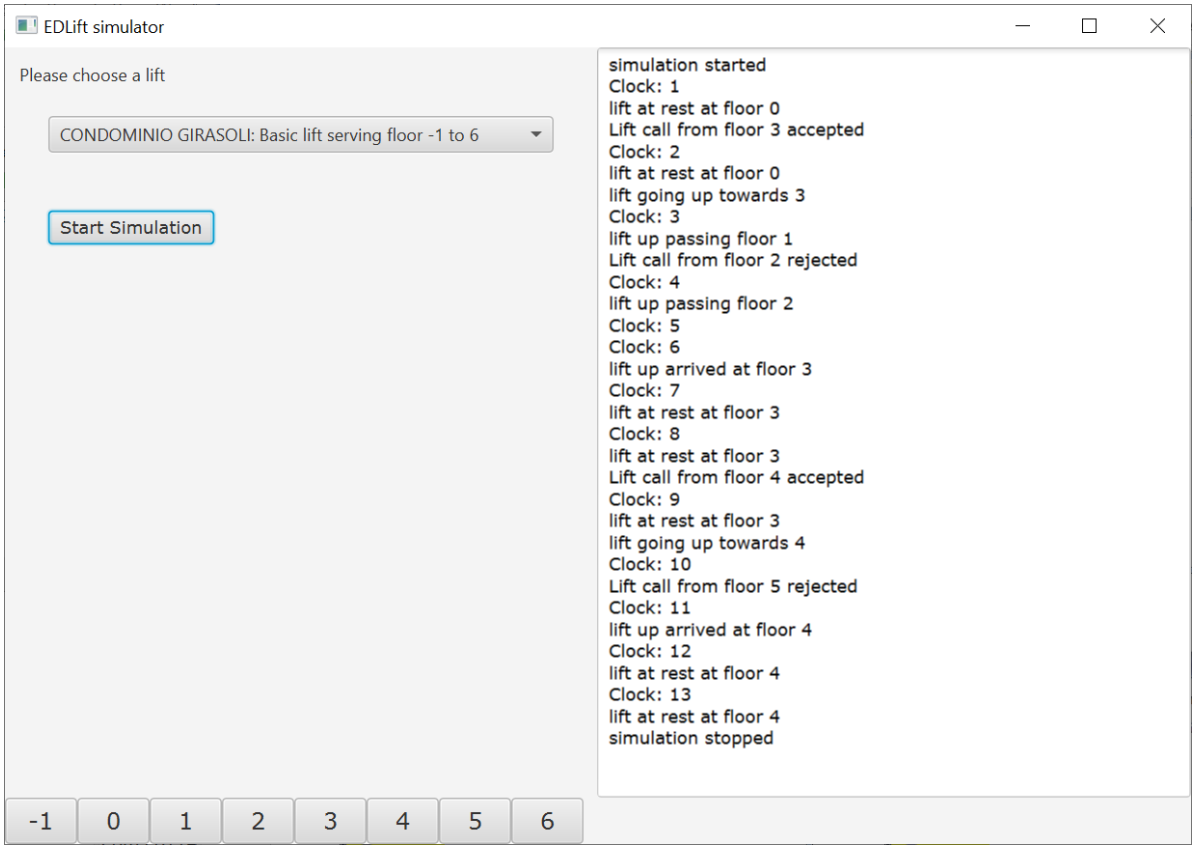


Figura 2

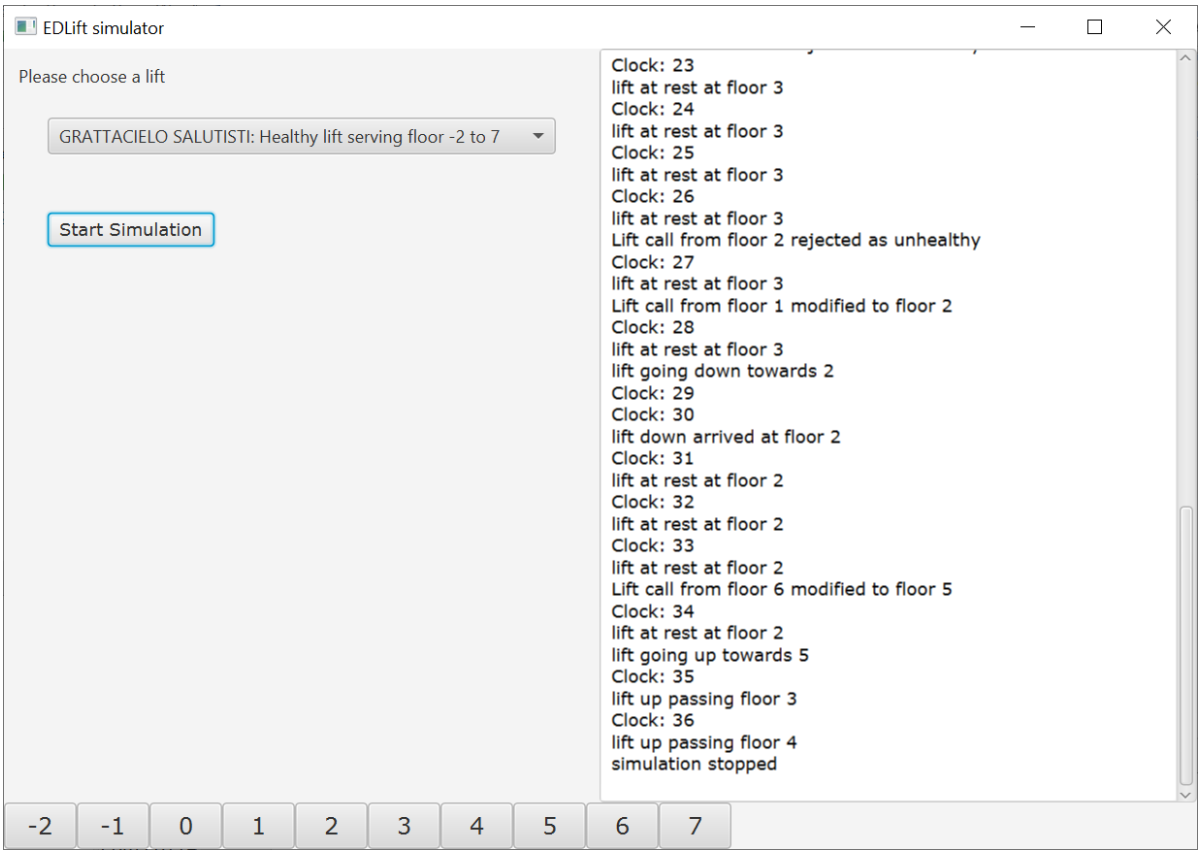


Figura 3

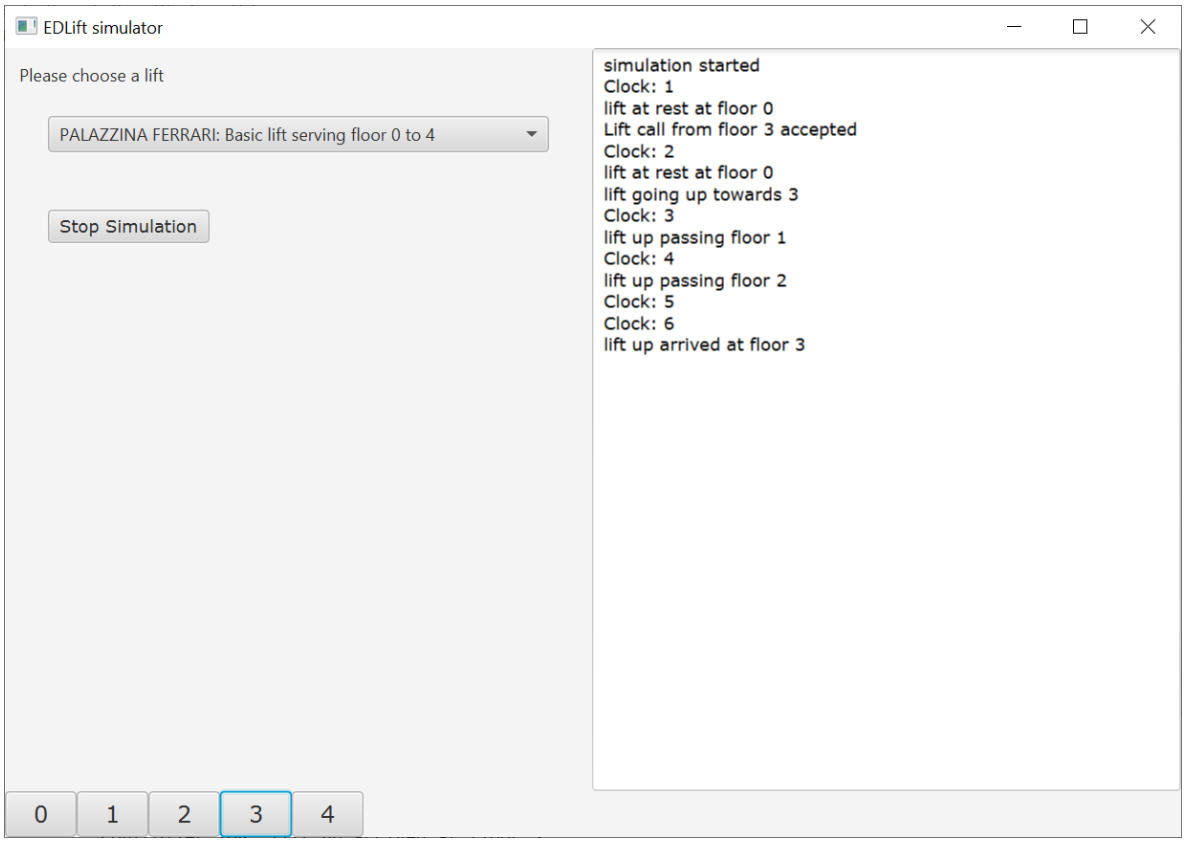


Figura 4