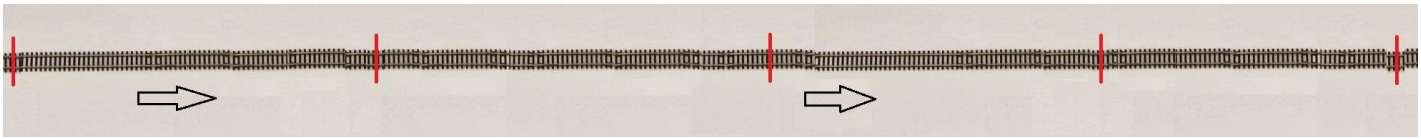
ESAME DI FONDAMENTI DI INFORMATICA T-2 del 9/1/2020
Proff. E. Denti, R. Calegari, A. Molesini – Tempo: 4 ore

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:

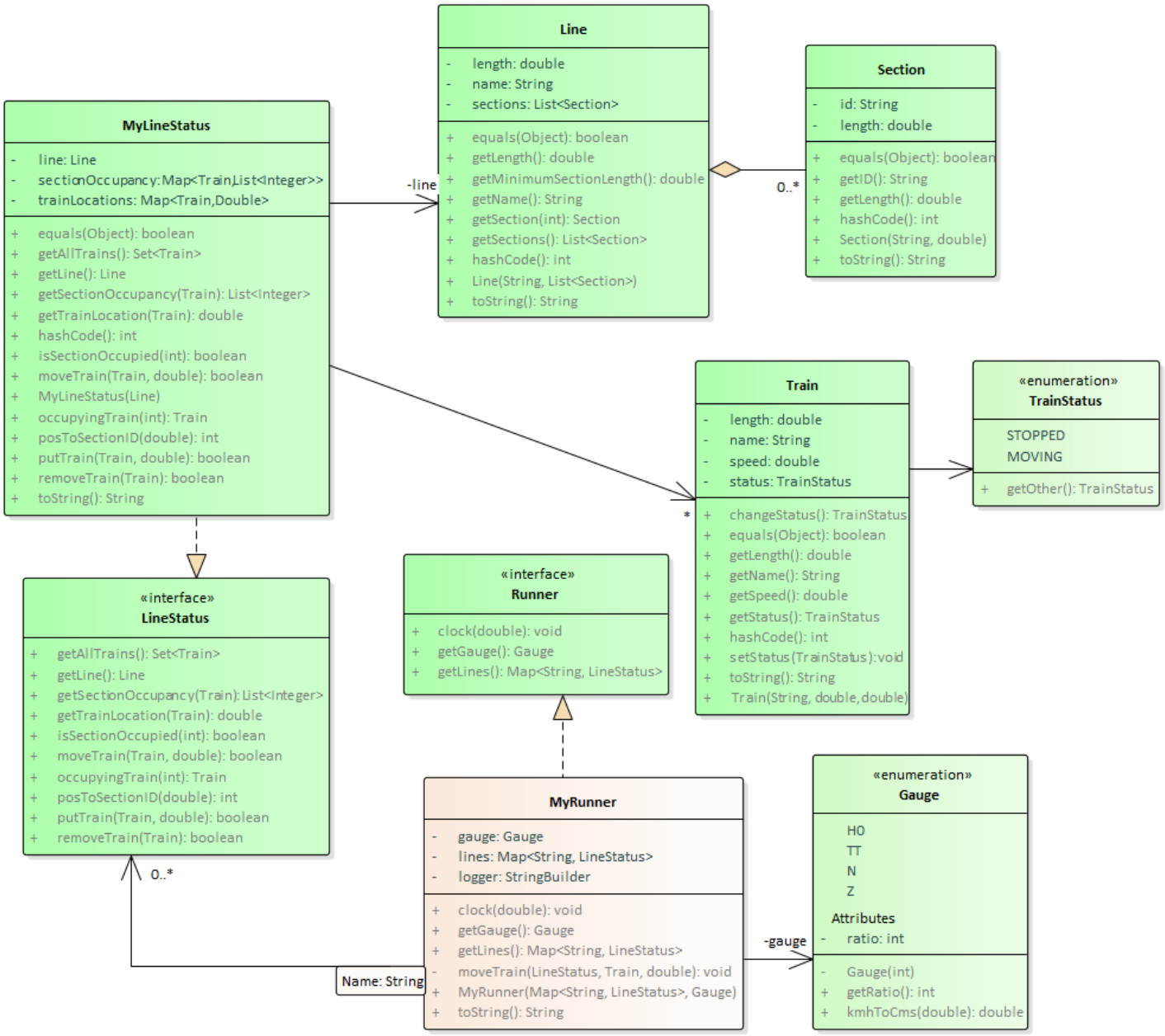
- $\mathbf{v_i}$ la velocità caratteristica del treno $\mathbf{T_i}$
- $\mathbf{pos_i}$ la posizione in un dato istante del treno $\mathbf{T_i}$

quando l'utente preme il pulsante CLOCK, la posizione viene ricalcolata secondo la formula $\mathbf{pos_i}(t+\Delta t) = \mathbf{pos_i}(t) + \mathbf{v_i} * \Delta t$
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:

a) 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).

b) 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).

c) 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.

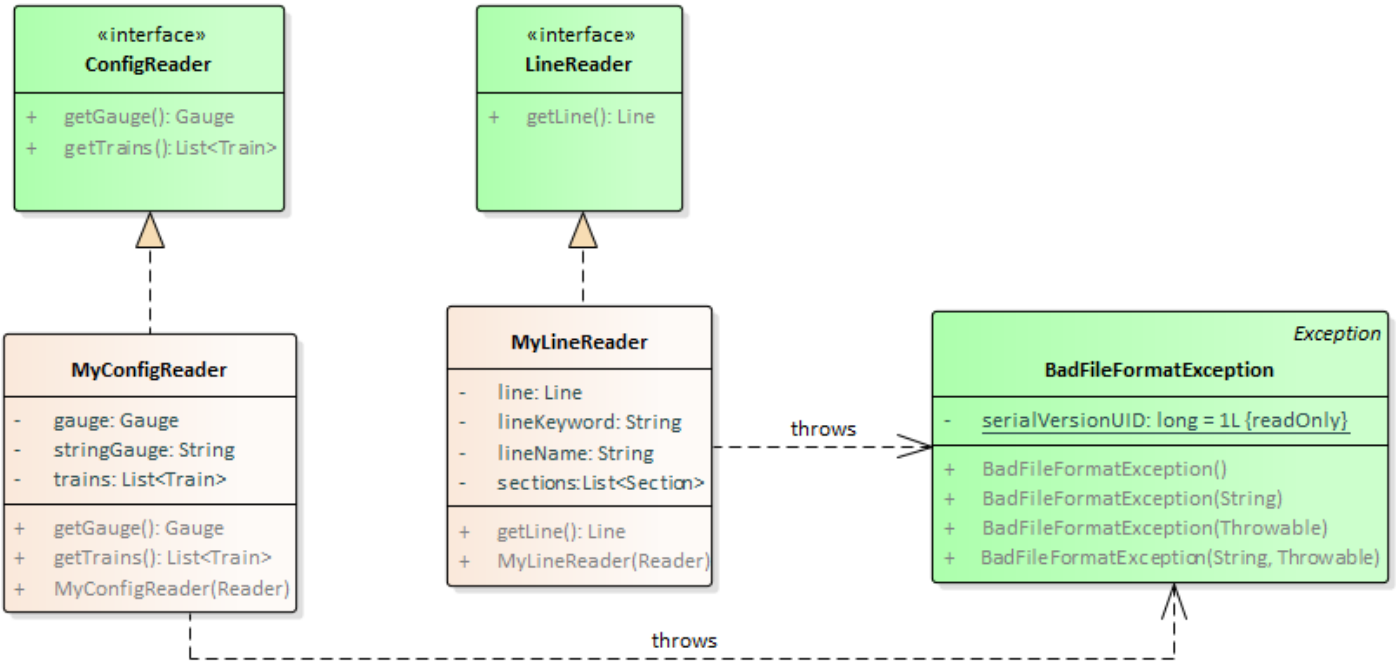
h) L'interfaccia ***Runner*** (fornita) cattura la nozione di simulatore del movimento dei treni nel plastico: incapsula una o più linee, espresse da altrettanti ***LineStatus***, e l'informazione sulla scala (un oggetto ***Gauge***). Offre metodi per:
   - recuperare le linee, sotto forma di mappa (nome, 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 restituisca 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:

a) l'interfaccia ***ConfigReader*** (fornita) dichiara i due metodi ***getGauge*** e ***getTrains***, dall'ovvio significato

b) 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 ***BadFileFormatException*** (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
```

c) l'interfaccia ***LineReader*** (fornita) dichiara il solo metodo ***getLine,*** che legge dal file i dati di una **singola** linea

d) 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 ***BadFileFormatException*** (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***

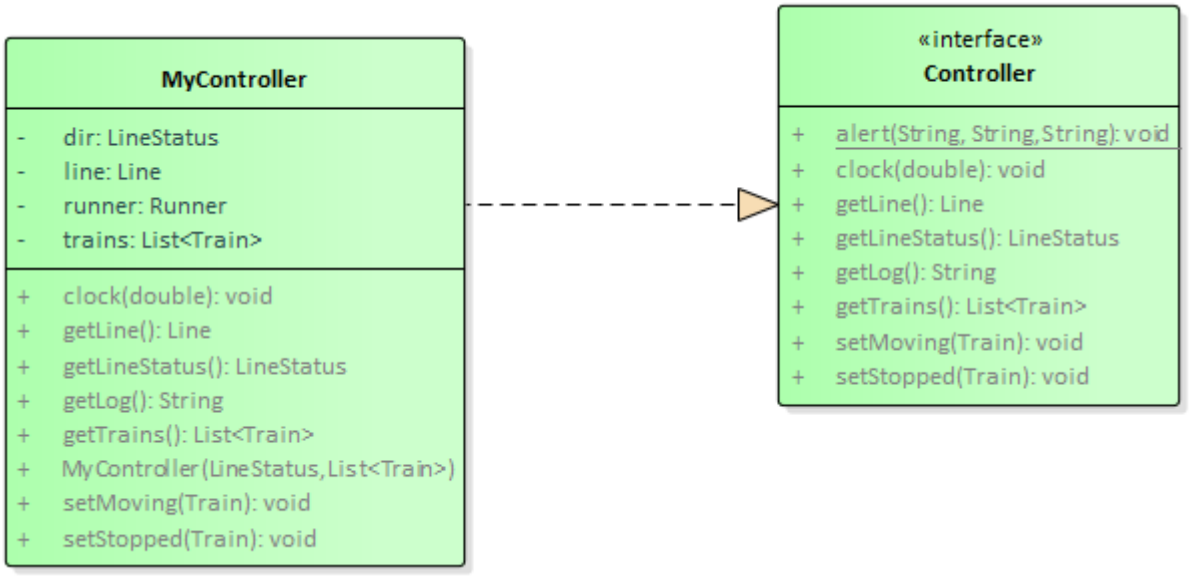
FORMATO DEL FILE DI LINEA: la prima riga contiene il nome della linea, preceduto dalla parola "Line" (scritta con qualunque sequenza di caratteri maiuscoli e/o minuscoli). Le righe successive descrivono ciascuna una sezione, tramite una serie di dati separati da spazi: nell'ordine si trovano la parola "Section", l'identificativo della sezione stessa (senza spazi) e infine la sua lunghezza (un numero reali, 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)

GUI (namespace minirail.ui) *(punti: 8)*

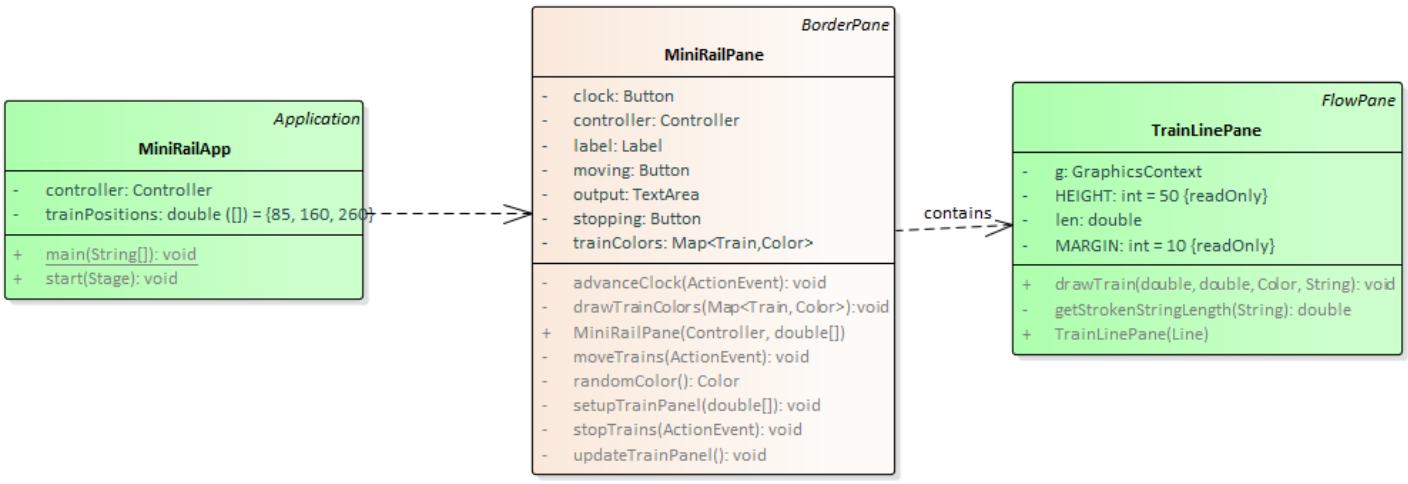
In caso di malfunzionamento dei reader, l'app è già preconfigurata per utilizzare valori di default anziché leggerli da file.

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 (convenzionalmente 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.



a) La classe ***MiniRailApp*** (fornita) contiene il main di partenza dell'applicazione.

b) 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.*

c) 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
    1. far avanzare il clock del controller di ½ secondo
    2. emettere sull'area di testo il log del controller

3. 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"?

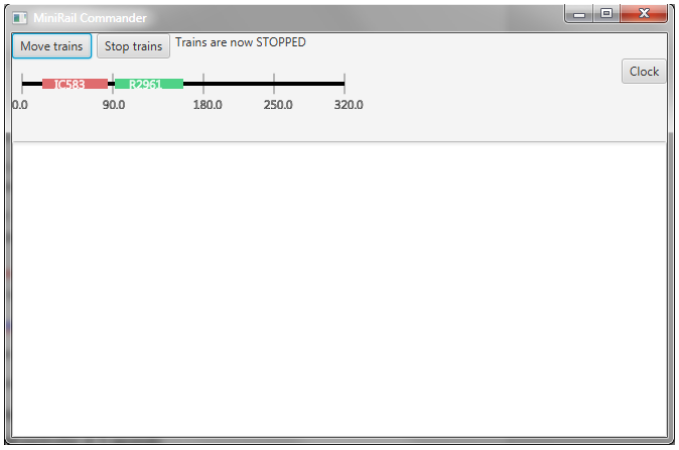


Figura 1

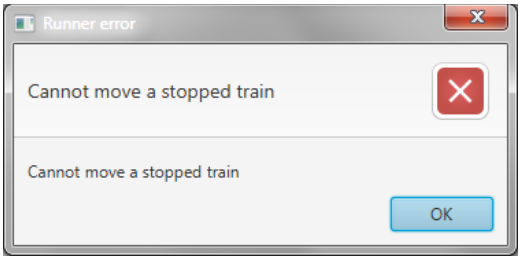


Figura 2

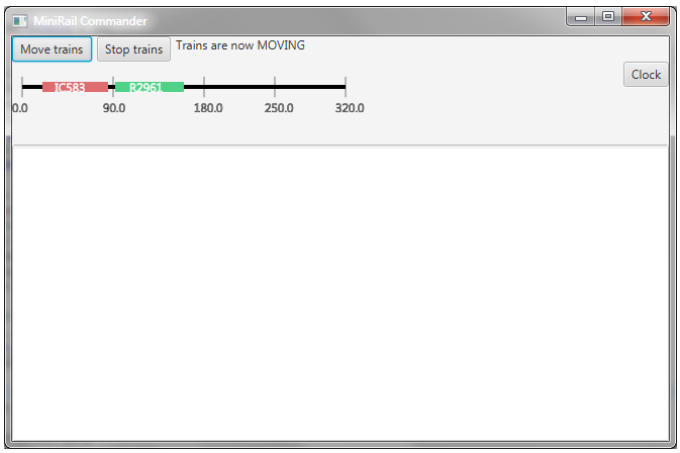


Figura 3

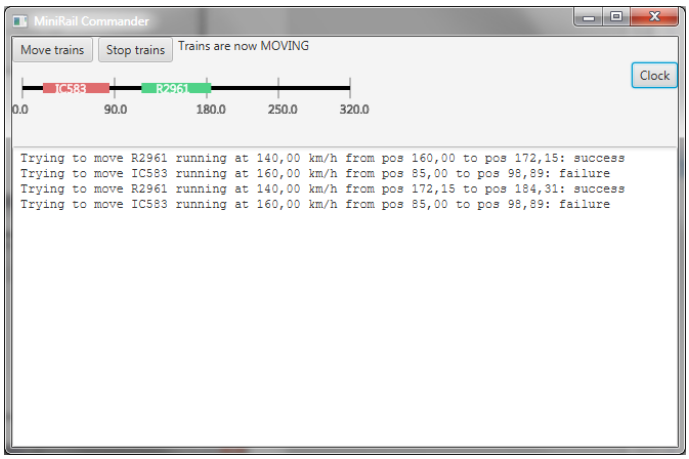


Figura 4

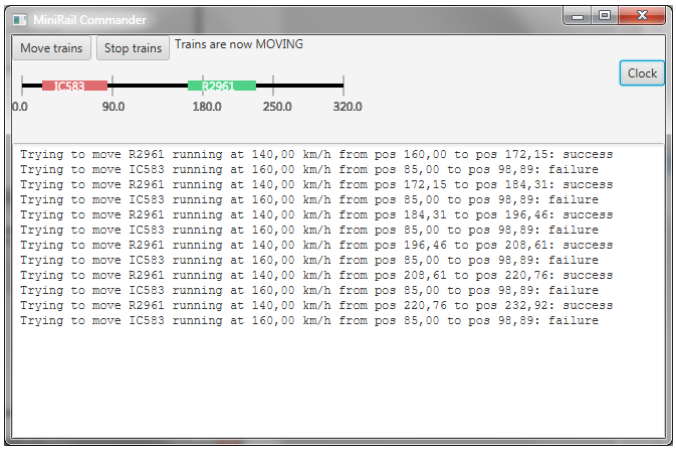


Figura 5

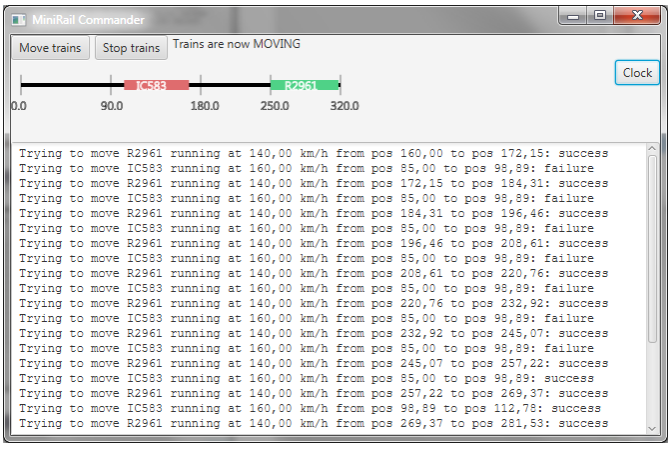


Figura 6

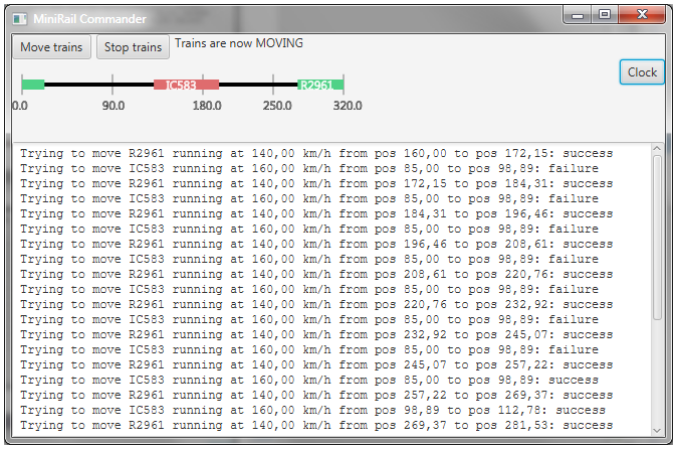


Figura 7

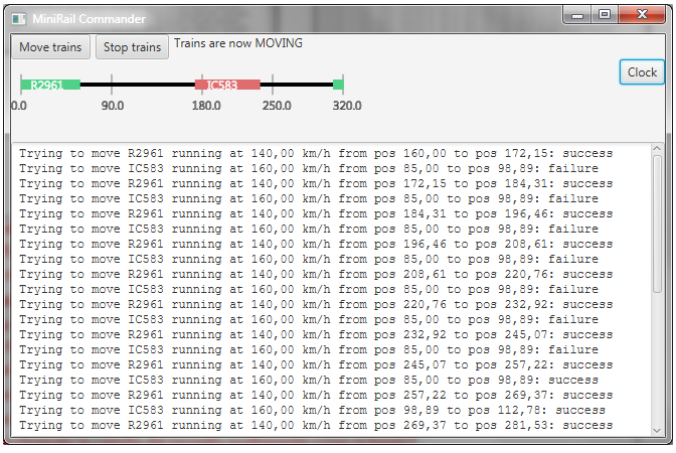


Figura 8