SAPIENZA Università di Roma
Facoltà di Ingegneria dell'Informazione, Informatica e Statistica
Corso di Laurea in Ingegneria Informatica ed Automatica
Corso di Progettazione del Software
Esame del **10 settembre 2019**
Tempo a disposizione: 3 ore

Requisiti. L'applicazione da progettare riguarda un sistema ferroviario. Ogni vagone di un treno è collegato al precedente e al successivo. Il vagone di testa non ha un precedente e l'ultimo non ha un successivo. Di ogni vagone interessa il codice (una stringa). Un treno di cui interessa il nome (una stringa) contiene una serie di vagoni, di cui interessa solo il primo dal momento che i successivi sono semplicemente quelli che seguono. Non è di interesse risalire dal vagone al treno a cui appartiente, solo l'opposto. Alcuni vagoni sono trainanti, e di questi interessa il numero di CV di potenza (un intero). Altri sono vagoni letto e di questi interessa il numero di posti letto. Ogni treno ha un certo numero arbitrario di dipendenti ad esso assegnati. Di ogni dipendente interessa il nome e gli anno di servizio. Ogni treno ha un dipendente con il ruolo di capotreno.

Una parte specifica dell'applicazione riguarda il movimento dei vagoni. In particolare, un vagone può trovarsi in *sosta* o in *movimento*. La transizione dalla sosta al movimento viene comunicata con un evento *partenza*, mentre da movimento a sosta dall'evento *arrivo*. Un vagone in sosta che non sia attaccato a nessun altro può venire attaccato in coda a un altro vagone. Questo evento si chiama *sposta* con payload il vagone a cui attaccarsi e causa la verifica che quest'ultimo sia effettivamente un vagone di coda, ossia non ha già un successivo. La verifica avviene attraverso il seguente protocollo: il vagone da attaccare invia un evento *arrivo* al vagone a cui va attaccato; questo risponde *libero* oppure *occupato*. Solo nel primo caso lo spostamento avviene (con gli opportuni aggiornamenti del diagramma delle classi), nel secondo no.

Siamo interessati alla seguente attività principale. L'attività prende in input un treno T e un insieme V di vagoni e verifica che ciascun vagone non sia collegato ad altri e che sia presente almeno un vagone motorizzato. Se la verifica non va a buon fine, l'attività termina segnalando in output un errore. Altrimenti concorrentemente esegue le seguenti due sottoattività: *(i)* formazione e *(ii)* orario. La sottoattività di formazione del treno *(i)* forma il treno eseguendo spostamenti dei vagoni in V (i dettagli non interessano). La sottoattività chiede l'orario di orario di partenza e la tratta da percorrere e calcola l'orario di arrivo dei T (i dettagli non interessano). Quando entrambe le sottoattivita' sono terminate, viene stampato l'orario di arrivo del treno.

Domanda 1. Basandosi sui requisiti riportati sopra, effettuare l'analisi producendo lo schema concettuale in UML per l'applicazione, comprensivo di: diagramma delle classi (inclusi eventuali vincoli non esprimibili in UML); diagramma stati e transizioni per la classe *Vagone*; diagramma delle attività; specifica del diagramma stati e transizioni; segnatura dell'attività principale, sottoattività non atomiche, atomiche e segnali di input/output. Si noti che NON è richiesta la specifica delle attività. Motivare, qualora ce ne fosse bisogno, le scelte di progetto.

Domanda 2. Effettuare il progetto, illustrando i prodotti rilevanti di tale fase e motivando, qualora ce ne fosse bisogno, le scelte di progetto. In particolare definire SOLO le responsabilità sulle associazioni del diagramma delle classi (nella tabella, inserire anche il motivo di ognuna delle responsabilità).

Domanda 3. Effettuare la realizzazione, producendo un programma JAVA e motivando, qualora ce ne fosse bisogno, le scelte di progetto. In particolare realizzare in JAVA SOLO i seguenti aspetti dello schema concettuale:

- La classe `Vagone` con la classe `VagoneFired`, le classi JAVA per rappresentare le *associazioni* di cui la classe `Vagone` ha responsabilità.
- L'*attività principale* e le sue eventuali sottoattività NON atomiche.

DIAGRAMMA DELLE CLASSI

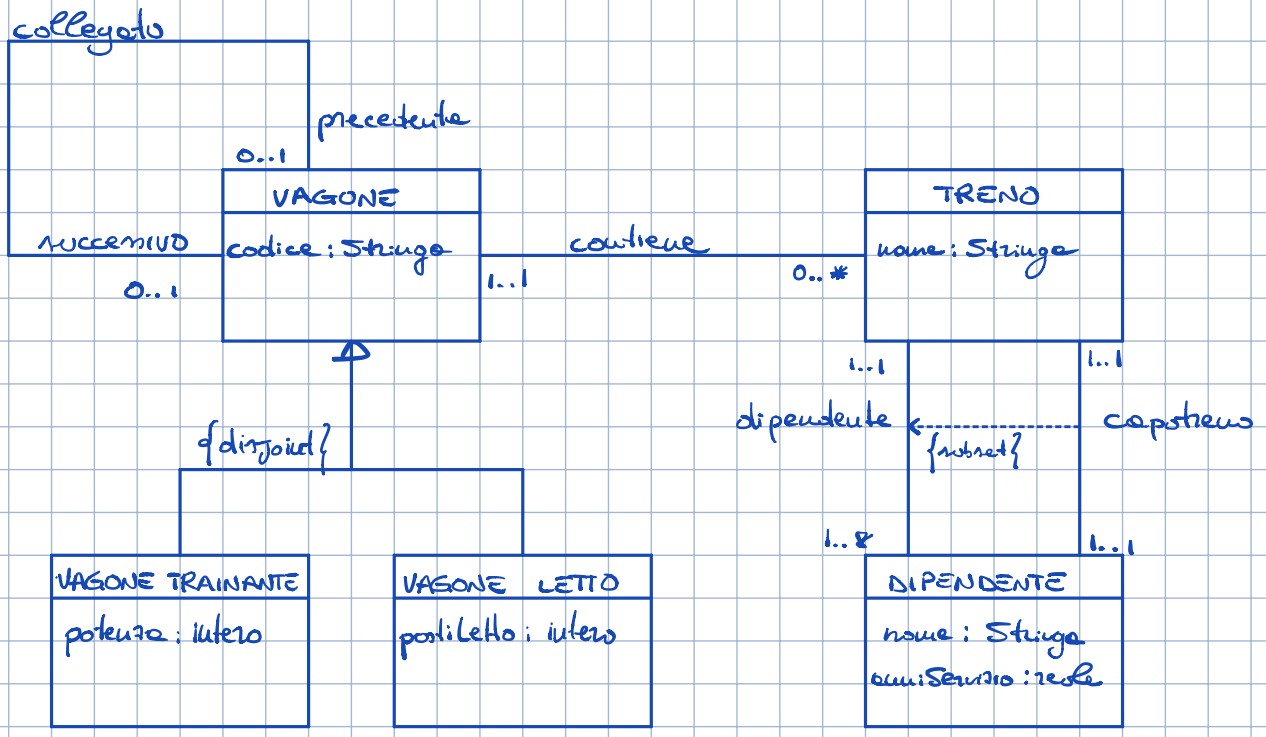
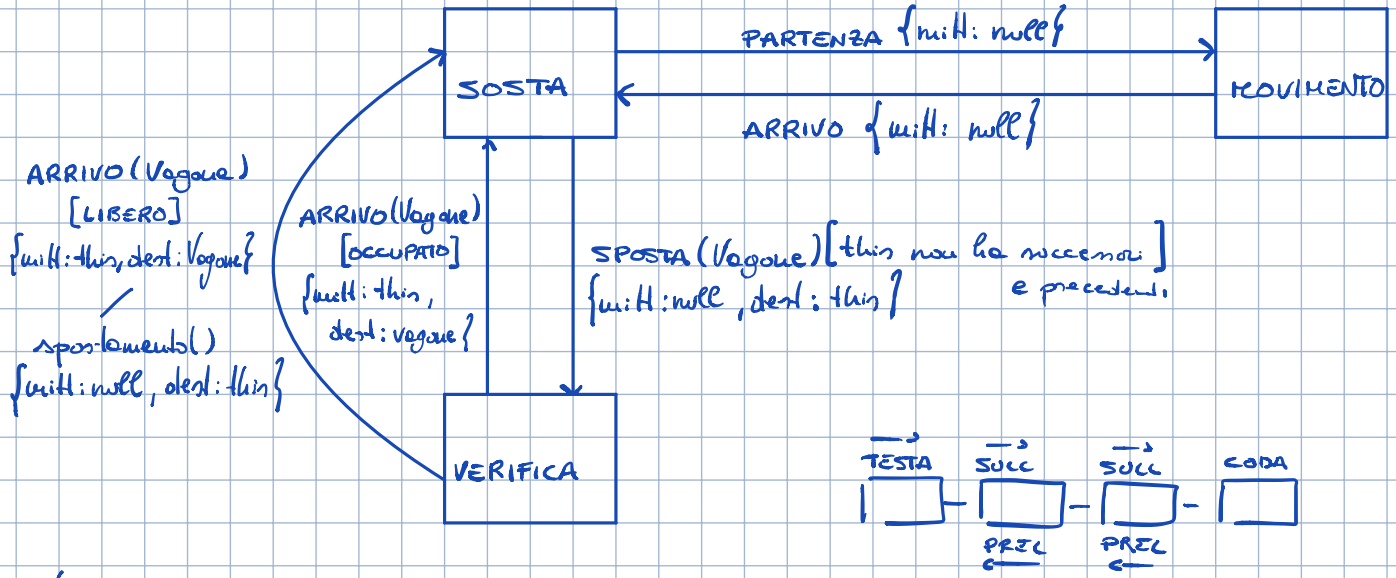


DIAGRAMMA STATI-TRANSIZIONI



Variabile ausiliare

Vagone : vagone

Nota: occupato = vagone non è un vagone di coda

libero = vagone è un vagone di coda

DIAGRAMMA ATTIVITA':

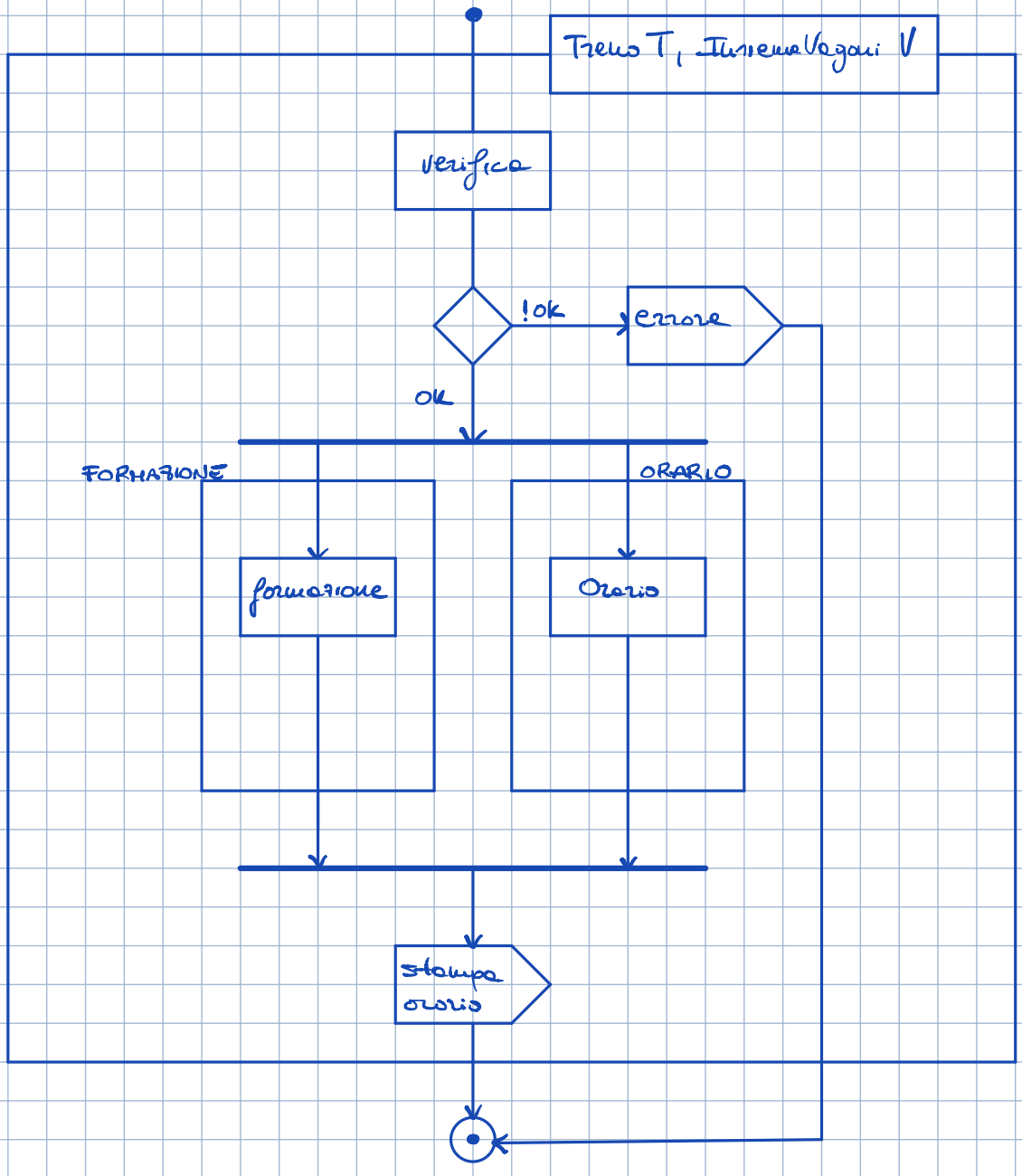
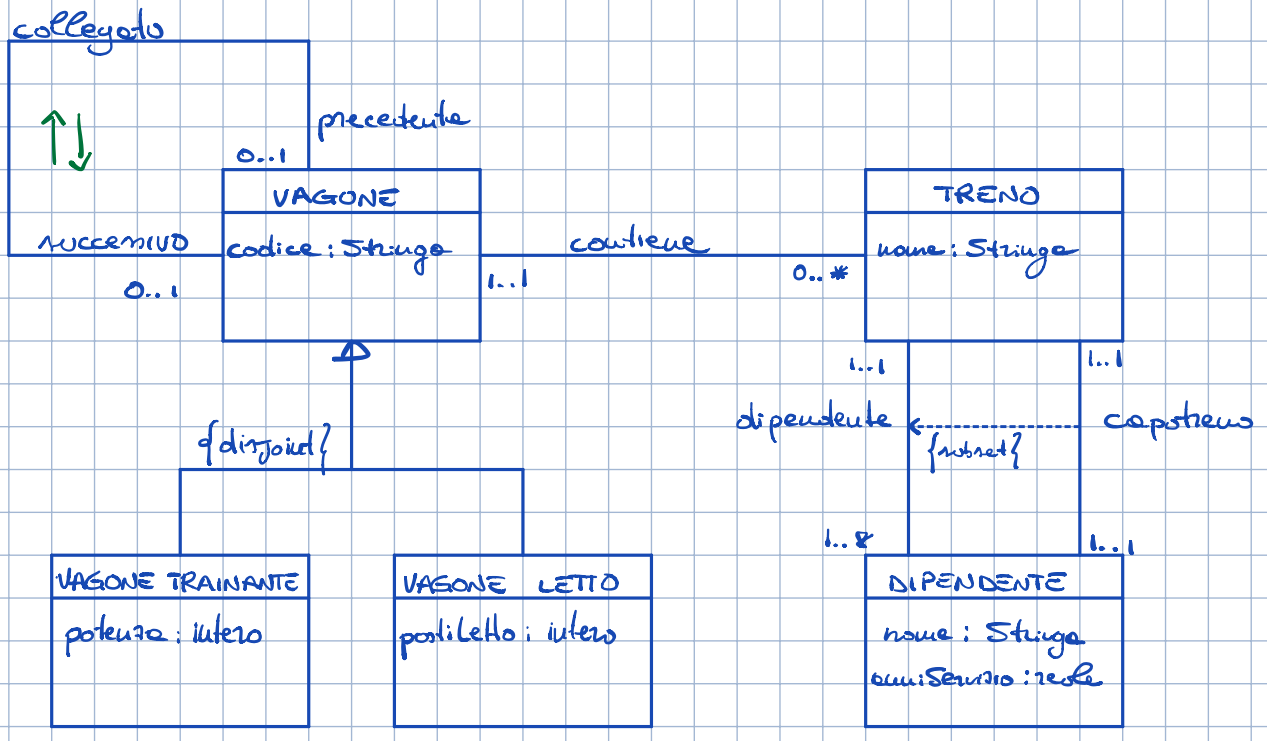


TABELLA RESPONSABILITÀ:

collegato	precedente	SI 1 2
	successivo	SI 1 2
contiene	vagone	NO
	treno	SI 1
dipendente	dipendente	SI 1
	treno	SI 1
capotreno	capotreno	SI 1
	treno	SI 1



Una parte specifica dell'applicazione riguarda il movimento dei vagoni. In particolare, un vagone può trovarsi in *sosta* o in *movimento*. La transizione dalla sosta al movimento viene comunicata con un evento *partenza*, mentre da movimento a sosta dall'evento *arrivo*. Un vagone in sosta che non sia attaccato a nessun altro può venire attaccato in coda a un altro vagone. Questo evento si chiama *sposta* con payload il vagone a cui attaccarsi e causa la verifica che quest'ultimo sia effettivamente un vagone di coda, ossia non ha già un successivo. La verifica avviene attraverso il seguente protocollo: il vagone da attaccare invia un evento *arrivo* al vagone a cui va attaccato; questo risponde *libero* oppure *occupato*. Solo nel primo caso lo spostamento avviene (con gli opportuni aggiornamenti del diagramma delle classi), nel secondo no.

Siamo interessati alla seguente attività principale. L'attività prende in input un treno T e un insieme V di vagoni e verifica che ciascun vagone non sia collegato ad altri e che sia presente almeno un vagone motorizzato. Se la verifica non va a buon fine, l'attività termina segnalando in output un errore. Altrimenti concorrentemente esegue le seguenti due sottoattività: *(i)* formazione e *(ii)* orario. La sottoattività di formazione del treno *(i)* forma il treno eseguendo spostamenti dei vagoni in V (i dettagli non interessano). La sottoattività chiede l'orario di orario di partenza e la tratta da percorrere e calcola l'orario di arrivo dei T (i dettagli non interessano). Quando entrambe le sottoattivita' sono terminate, viene stampato l'orario di arrivo del treno.

Specifica stati e transizioni

InizioSpecificaStatiClasse Vagone

Stato: {SOSTA, MOVIMENTO, VERIFICA}
Variabili ausiliarie:
Vagone: vagone;
Stato iniziale:
statoCorrente = SOSTA;
vagone = --;

FineSpecifica

InizioSpecificaTransizioniClasse Vagone

Transizione: SOSTA -> MOVIMENTO
partenza() {mitt: null. dest:this}
Evento: partenza() {mitt: null. dest:this}
Condizione: --
Azione:
pre: --
post: --

Transizione MOVIMENTO -> SOSTA
arrivo() {mitt: null. dest:this}
Evento: arrivo() {mitt: null. dest:this}
Condizione: --
Azione:
pre: --
post: --

Transizione: SOSTA -> VERIFICA
sposta(vagone) {mitt: null, dest: this}
Evento: sposta(vagone) {mitt: null, dest: this}
Condizione: this non ha successori e precedenti;
Azione:
pre: --
post: --

Transizione VERIFICA -> SOSTA
arrivo(vagone)[occupato] {mitt: this, dest: vagone}
Evento: arrivo(vagone) {mitt: this, dest: vagone}
Condizione: vagone ha successori;
Azione:
pre: --
post: --

Transizione VERIFICA -> SOSTA

arrivo(vagone)[libero]{mitt: this, dest: vagone} / spostamento() {mitt: null, dest: this}

Evento: arrivo(vagone) {mitt: this, dest: vagone}

Condizione: vagone non ha successori;

Azione:

pre: --

post: spostamento() {mitt: null, dest: this}

FineSpecifica

Segnatura Attività

AttivitaPrincipale(Treno t, Insieme<Vagoni> v):()	//attività complessa
Formazione(Treno t, Insieme<Vagoni> v):()	//attività complessa
Orario(Treno t):(Stringa orario)	//attività complessa
Verifica(Insieme<Vagoni> v):(boolean ok)	//attività atomica
Formazione(Treno t, Insieme<Vagoni> v):()	//attività atomica
Orario(OrarioDiPartenza o, TrattaDaPercorrere t):(Stringa orarioArrivo)	//attività atomica
Errore():()	//segnaleIO
StampaOrario(Stringa orarioArrivo):()	//segnaleIO

Realizzazione

Da realizzare:

- Vagone
- VagoneFired
- TipoLinkCollegato
- ManagerCollegato
- AttivitaPrincipale
- Formazione
- Orario

Classe Vagone

```
public class Vagone implements Listener {
	private final String codice;
	private TipoLinkSuccessivo successivo;
	private TipoLinkPrecedente precedente;

	public Vagone(String c) {
		codice = c;
	}

	public String getCodice() {return codice;}

	//gestione successive

	public int quantiSuccessivo(){
		if(successivo == null) return 0;
		else return 1;
	}

	public TipoLinkSuccessivo getTipoLinkSuccessivo(){
		return successivo;
	}

	public void inserisciTipoLinkSuccessivo(TipoLinkSuccessivo t){
		if(t!= null && t.getPrecedente().equals(this)){
			ManagerSuccessivo.inserisci(t);
		}
	}

	public void eliminaTipoLinkSuccessivo(TipoLinkSuccessivo t){
		if(t!= null && t.getPrecedente().equals(this)){
			ManagerSuccessivo.elimina(t);
		}
	}

	public void inserisciPerManagerSuccessivo(ManagerSuccessivo m){
		if(m!=null){
			successivo = m.getLink();
		}
	}

	public void eliminaPerManagerSuccessivo(ManagerSuccessivo m){
		if(m!=null){
			successivo = null;
		}
	}
```

```
	//gestione precedente

	public int quantiPrecedente(){
		if(precedente == null) return 0;
		else return 1;
	}

	public TipoLinkPrecedente getTipoLinkPrecedente(){
		return precedente;
	}

	public void inserisciTipoLinkPrecedente(TipoLinkPrecedente t){
		if(t!= null && t.getSuccessivo().equals(this)){
			ManagerPrecedente.inserisci(t);
		}
	}

	public void eliminaTipoLinkPrecedente(TipoLinkPrecedente t){
		if(t!= null && t.getSuccessivo().equals(this)){
			ManagerPrecedente.elimina(t);
		}
	}

	public void inserisciPerManagerPrecedente(ManagerPrecedente m){
		if(m!=null){
			precedente = m.getLink();
		}
	}

	public void eliminaPerManagerPrecedente(ManagerPrecedente m){
		if(m!=null){
			precedente = null;
		}
	}

	//gestione stato

	public static enum Stato = {SOSTA, MOVIMENTO, VERIFICA;}
	Vagone vagone;
	Stato statoCorrente = Stato.SOSTA;
	public void getStato(){
		return statoCorrente;
	}
	public void fired(Evento e){
		TaskExecutor.getInstance().perform(new VagoneFired(this, e));
	}
}
```

Classe VagoneFired (Si trova nello stesso package della classe Vagone)

```
class VagoneFired implements Task{
        private boolean eseguita = false;
        private Vagone v;
        private Evento e;

        public VagoneFired(Vagone v, Evento e){
                this.v = v;
                this.e = e;
        }

        public synchronized void run(){
                if(eseguita || (e.getMittente() != v && e.getMittente() != null)) { return; }
                eseguita = true;
                switch(v.getStato()){

                        case SOSTA:
                                if(e.getClass().equals(Partenza.class)){
                                        Partenza ii = (Partenza)e;
                                        v.statoCorrente = Stato.MOVIMENTO;
                                }
                                else if(e.getClass().equals(Sposta.class)){
                                        Sposta ii = (Sposta)e;
                                        if(ii.getDestinatario().equals(v) &&
                                        (v.getTipoLinkSuccessore().getSuccessore() == null)&&
                                        (v.getTipoLinkPrecedente().getPrecedente() == null)){
                                                v.vagone = ii.getPayload();
                                                v.statoCorrente = Stato.VERIFICA;
                                        }
                                }

                                break;

                        case MOVIMENTO:
                                if(e.getClass().equals(Arrivo.class)){
                                        Arrivo ii = (Arrivo)e;
                                        v.statoCorrente = Stato.SOSTA;
                                }

                                break;

                        case VERIFICA:
                                if(e.getClass.equals(Arrivo.class)){
                                        Arrivo ii = (Arrivo)e;
                                        if(ii.getMittente.equals(v) &&
                                          ii.getDestinatario().equals(v.vagone)){
                                                if(v.vagone.getTipoLinkSuccessivo().getSuccessivo() == null){
                                                        Envirioment.nuovoEvento(new Spostamento(null, v));
                                                        v.statoCorrente = Stato.SOSTA;
                                                }
```

```
                                        else if(v.vagone.getTipoLinkSuccessivo().getSuccessivo()!=null){
                                                v.statoCorrente = Stato.SOSTA;
                                        }
                                }
                        }

                default:
                        throw new RuntimeException("Stato non riconosciuto");
        }
    }

    public synchronized boolean estEseguita(){
        return eseguita;
    }
}
```

Classe TipoLinkSuccessivo

```
public class TipoLinkSuccessivo {
	private Vagone successivo;
	private Vagone precedente;

	public TipoLinkSuccessivo(Vagone s, Vagone p){
		successivo = s;
		precedente = p;
	}

	public Vagone getSuccessivo(){ return successivo; }
	public Vagone getPrecedente(){ return precedente; }

	public boolean equals(Object o){
		if(o!=null && getClass().equals(o.getClass())){
			TipoLinkSuccessivo t = (TipoLinkSuccessivo) o;
			return t.getSuccessivo = successivo && t.getPrecedente = precedente;
		}
		else return false; }

	public int hashCode(){ return sucessivo.hashCode()+precedente.hashCode(); }

}
```

Classe ManagerSuccessivo

```
public class ManagerSuccessivo{
	private final TipoLinkSuccessivo link;
	private ManagerSuccessivo(TipoLinkSuccessivo t){
		link = t;
	}

	public void inserisci(TipoLinkSuccessivo t){
		if(t!=null){
			MaganerSuccessivo m = new ManagerSuccessivo(t);
			t.getSuccessivo().inserisciPerManagerSuccessivo(m);
			t.getPrecedente().inserisciPerMaganerSuccessivo(m);
		}
	}

	public void elimina(TipoLinkSuccessivo t){
		if(t!=null){
			MaganerSuccessivo m = new ManagerSuccessivo(t);
			t.getSuccessivo().eliminaPerManagerSuccessivo(m);
			t.getPrecedente().eliminaPerMaganerSuccessivo(m);
		}
	}

	public TipoLinkSuccessivo getLink() {return link};

}
```

Classe AttivitaPrincipale

```
public class AttivitaPrincipale implements Runnable{
	private boolean eseguita = false;
	private Treno t;
	private HashSet<Vagoni> v;

	public AttivitaPrincipale(Treno t, HashSet<Vagoni> v){
		this.t = t;
		this.v = v;
	}

	public synchronized void run(){
		if(eseguita) return;
		eseguita = true;
		while(eseguita){
			Verifica ver = new Verifica(v)
			TaskExecutor.getInstance().perform(ver);
			boolean ok = ver.getResult();
			if(!ok){
				SegnaliIO.errore();
			}
			else{
				Formazione f = new Formazione(t, v);
				Orario o = new Orario(t);
				Thread t1 = new Thread(f);
				Thread t2 = new Thread(o);
				t1.start();
				t2.start();
				try{
					t1.join();
					t2.join();
				} catch (InterruptedException e){
					e.printStackTrace();
				}
				SegnaliIO.stampaOrario(o.getResult());
			}
		}
	}

	public synchronized boolean estEseguita(){
		return eseguita;
	}
}
```

Classe Formazione

```
public class Formazione implements Runnable{
	boolean eseguita;
	private Treno t;
	private HashSet<Vagoni> v;

	public Formazione(Treno t, HashSet<Vagoni> v){
		this.t = t;
		this.v = v;
	}

	public synchronized void run {
		if(eseguita) return;
		eseguita = true;
		Formazione f = new Formazione(t,v);
		TaskExecutor.getInstance().perform(f);
	}

	public synchronized boolean eseguita(){
		return eseguita;
	}
}
```

Classe Orario

```
public class Orario implements Runnable{
	boolean eseguita;
	private Treno t;
	private String orarioDiPartenza;

	public Formazione(Treno t){
		this.t = t;
	}

	public synchronized void run {
		if(eseguita) return;
		eseguita = true;
		Orario o = new Orario(t);
		TaskExecutor.getInstance().perform(o);
		orarioDiPartenza = o.getResult();
	}

	public synchronized String getResult(){
		return orarioDiPartenza;
	}

	public synchronized boolean eseguita(){
		return eseguita;
	}
}
```