ESAME 10 SETTEMBRE 2019

InizioSpecificaStati Vagone:

Stati: {SOSTA, MOVIMENTO, ATTESA}

Variabili di stato ausiliarie:
vagone: Vagone

Stato iniziale:
statoCorrente = SOSTA
vagone = _

FineSpecificaStati

InizioSpecificaTransizioni Richiesta:

Transizione: SOSTA -> MOVIMENTO
Evento: partenza {dest == this}
Condizione: _
Azione:
Pre: _
Post: _

Transizione: MOVIMENTO -> SOSTA
Evento: arrivo
Condizione: _
Azione:
Pre: _
Post: _
Transizione: SOSTA -> ATTESA
Evento: sposta {dest = this, payLoad = vagone} / arrivo {dest = vagone, mitt = this}
Condizione: _
Azione:
Pre: _
Post: nuovoEvento = new arrivo(this,vagone));
Transizione: ATTESA -> SOSTA
Evento: risposta{dest = this, mitt = vagone, payLoad = esitoVerifica} / eventuale SideEffect
Condizione: e.getMitt() = this.vagone;
Azione:
Pre: _
Post: if(esitoVerifica == true) {this.prec = vagone && vagone.succ = this;} else return;

FineSpecificaTransizioni

SpecificaAttività solo segnature
AttivitàPrincipale(Treno t, HashSet<Vagone> vv):() //complessa
Formazione(Treno t, HashSet<Vagone> vv):() //complessa, compone il treno

Orario(Treno t):(Ora arrivo) //complessa, il tipo di orario va definito nella seconda parte
Verifica(HashSet<Vagone> vv):(boolean ok)//atomica
Spostamento(Treno t, HashSet<Vagone> vv):() //atomica, compone il treno
RichiestaOrario(Treno t):(Ora partenza) //atomica
RichiestaTratta(Treno t):(String tratta) //atomica, immagino che la tratta sia indicata con una stringa
Calcolo(OrarioPartenza p, String tratta):(Ora arrivo) //atomica
Errore():() //IO, stampa messaggio di errore
StampaArrivo(Ora arrivo):() //IO, stampa ora di arrivo

Tabella responsabilità
1 molteplicità, 2 operazioni, 3 requisiti

collegati	vagone(precedente)	1,2
	vagone(successivo)	1,2
primo	vagone	(forse 1 ma 3 dice no) no
	treno	1
lavoraIn	dipendente	no
	treno	1
capotreno	dipendente	no
	treno	1

Classe Vagone

```
public class Vagone implements Listener{ //uso protected al posto di private per permettere l'ereditarietà
        protected String codice;
        protected TipoLinkCollegati successivo; //molteplicità 0..1
        protected TipoLinkCollegati precedente;//molteplicità 0..1

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

        public getCodice(){
                return codice;
        }

        //gestione successivo con molteplicità doppia (la gestione di precedente è identica ma non viene riportata)
        public TipoLinkCollegati getLinkSuccessivo(){
                return successivo;
```

```
	}
	public void inserisciLinkSuccessivo(TipoLinkCollegati l){
		if(l!=null && l.getPrecedente() == this && successivo == null){ //forse si può omettere l'ultimo controllo e permettere di sovrascivere il successivo
			managerCollegati.inserisciSuccessivo(l);
		}
		throw new EccezionePrecondizioni();
	}

	public void rimuoviLinkSuccessivo(){ //il link da eliminare non va come argomento perchè ce n'è solo uno
		if(successivo!=null){
			managerCollegati.eliminaSuccessivo();
		}
		throw new EccezionePrecondizioni();
	}

	public void inserisciPerManagerSuccessivo(ManagerCollegati m){
		successivo = m.getLink();
	}

	public void eliminaPerManagerSuccesivo(MangerCollegati m){
		successivo = null;
	}

	public boolean haSuccessivo(){
		return successivo!=null;
	}

	//gestione dello stato
	public static enum Stato {SOSTA, MOVIMENTO, ATTESA}

	Stato statoCorrente = Stato.SOSTA;
	Vagone vagone = null;

	public Stato getStato{
		return statoCorrente;
	}

	public void fired(Evento e){
		TaskExecutor.getInstance().perform(new VagoneFired(this,e));
	}
}
```

Classe TipoLinkCollegati

```
public class TipoLinkCollegati{
	private Vagone precedente;
```

```
	private Vagone Successivo;

	public TipoLinkCollegati(Vagone p, Vagone s){
		if(precedente == null || successivo == null) throw new
EccezionePrecondizioni();
		precedente = p;
		successivo = s;
	}

	public Vagone getPrecente(){
		return precedente;
	}

	public Vagone getSuccessivo(){
		return successivo;
	}

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

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

ClasseManagerCollegati

```
public final class ManagerCollegati{
	private final TipoLinkCollegati link;

	private ManagerCollegati(TipoLinkCollegati l){
		if(l==null) throw new EccezionePrecondizioni();
		link = l;
	}

	public getLink(){
		return link;
	}

	public static void inserisci(TipoLinkCollegati l){
		if(l==null || l.getPrecedente() != null || l.getSuccessivo() != null) throw new
EccezionePrecondizioni();
```

```
                ManagerCollegati m = new ManagerCollegati(l);
                l.getPrecedente().inserisciPerManagerPrecedente(m);
                l.getSuccessivo().inserisciPerManagerSuccessivo(m);
        }

        public static void elimina(TipoLinkCollegati l){
                if(l==null) throw new EccezionePrecondizioni();
                ManagerCollegati m = new ManagerCollegati(l);
                l.getPrecedente().eliminaPerManagerPrecedente(m);
                l.getSuccessivo().eliminaPerManagerSuccessivo(m);
        }
}
```

Classe VagoneFired

```
class VagoneFired implements Runnable{ //nello stesso package di Vagone
        private Vagone v;
        private Evento e;
        private boolean eseguita = false;

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

        public synchronized void run(){
                if(eseguita || !(e.getDest()==v || e.getDest()==null)) return;
                eseguita = true;
                switch(v.getStato()){
                        case SOSTA:
                                if(e.getClass().equals(partenza.class)){
                                        v.statoCorrente = Stato.MOVIMENTO;
                                }
                                else if(e.getClass().equals(sposta.class)){
                                        v.vagone = e.getPayload();
                                        Environmente.aggiungiEvento(new
arrivo(v,e.getPayload())); //dove il payload è il vagone a cui attaccarsi
                                        v.statoCorrente = Stato.ATTESA;
                                }
                                break;
                        case MOVIMENTO:
                                if(e.getClass().equals(arrivo.class)){
                                        v.statoCorrente = Stato.SOSTA;
                                }
                                break;
                        case ATTESA:
                                if(e.getClass().equals(risposta.class)){
                                        if(e.getPayload == “libero”){
```

```
                                                TipoLinkCollegati l = new
TipoLinkCollegati(v.vagone,v);
                                                v.inserisciPrecedente(l);
                                                v.vagone.inserisciSuccessivo(l);
                                        }
                                        v.statoCorrente = Stato.SOSTA;
                                        v.vagone = null;
                                }
                                break;
                        default:
                                throw new RuntimeException("Stato non riconosciuto");

                }
        }

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

Classe AttivitàPrincipale

```
public class AttivitàPrincipale implements Runnable(){
        private Treno t;
        private HashSet<Vagone> vv;
        private boolean eseguita = false;

        public AttivitàPrinciplale(Treno t, HashSet<Vagone> vv){
                this.t = t;
                this.vv = vv;
        }

        public synchronized void run(){
                Verifica v = new Verifica(vv);
                TaskExecutor.getInstance().perform(v);
                boolean esito = v.getRis();
                if(!esito){
                        TaskExecutor.getInstance().perform(new Errore());
                }
                Formazione f = new Formazione(t,vv);
                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();
                        return;
                }
                StampaArrivo s = new StampaArrivo(o.getRis());
                TaskExecutor.getInstance().permorm(s);
                return;
        }

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

Classe Formazione

```
public class Formazione implements Runnable(){
        private Treno t;
        private HashSet<Vagone> vv;
        private boolean eseguita = false;

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

        public synchronized void run(){
                SottoFormazione s = new SottoFormazione(t,vv);
                TaskExecutor.getInsance().perform(s);  //il treno formato viene salvato in t
                return;
        }

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

Classe Orario

```
public class Orario implements Runnable(){
        private Treno t;
        private Orario arrivo = null;
        private boolean eseguita = false;

        public Orario(Treno t){
                this.t = t;
        }
```

```
        public synchronized void run(){
                RichiestaOrario r1 = new RichiestaOrario(t);
                TaskExecutor.getInsance().perform(r1);
                Orario partenza = r1.getRis();
                RichiestaTratta r2 = new RichiestaTratta(t);
                TaskExecutor.getInsance().perform(r2);
                String tratta = r2.getRis();
                Calcolo c = new calcolo(partenza,tratta);
                TaskExecutor.getInsance().perform(c);
                this.arrivo = c.getRis();
        }

        public synchronized boolean estEseguita(){
                return eseguita;
        }

        public Orario getRis(){
                if(!eseguita) throw new RuntimeException("Risultato non ancora calcolato");
                return arrivo;
        }
}
```