Progettazione del Software - 2021-01-29

```
Specifica stati:
   Stato = { PREPARAZIONE, CONSEGNA, COMPLETATO }
   variabili ausiliarie :
         Rider rider;
   Stato iniziale :
      Stato statocorrente = Stato.PREPARAZIONE;
       rider  = - -;
FineSpecifica
```

```
Segnatura Attività Complesse :

      AttivitaPrincipale (insiemeRider insR , insiemeOrdini insO):( );
      ConsegnaOrdini ( insiemeRider insR, insiemeOrdini insO ):();
      Analisi(insiemeRider insR , insiemeOrdini insO):(String report);

Segnatura Attività Atomiche:

      Verifica( insiemeRider insR, insiemeOrdini insO ):( boolean ok );

Segnatura Attività Input/Output :

     RichiediAggiornamento() : ();
      Report ( String report) : ();

FineSegnatura.
```

Realizzazione:

```
//CLASSE ORDINE
package Ordine;

public class Ordine implements Listener{
   private final int MIN_MOLT =1;
   Private final int MAX_MOLT = 1;
   private String codice;
   private HashSet<TipoLinkContiene> linkContiene;
   private Rider linkConsegnato;
   private TipoLinkEffettua linkEffettua;


   public Ordine ( String c  ) {
      this.codice = c;
      linkPizza = new HashSet<TipoLinkContiene>();
   }


   public String getCodice(){ return codice;}

   public void setCodice(String s){ codice = s;}


   public Rider getLinkConsegnato(){
      return linkConsegnato;
   }

   public void setLinkConsegnato(Rider l){
            linkConsegnato = l;
   }


  public TipoLinkEffettua getLinkEffettua(){
      if(linkEffettua == null)
          throw new RunTimeException("violata molteplicità 1..1");
      else return linkEffettua;
```

```
}

public int quantiEffettua() {
      if (linkEffettua == null) return 0;
     else return 1;
}

public void inserisciTipoLinkContiene(TipoLinkContiene l){
       if(l!=null)
              linkContiene.add(l);
}

public void eliminaTipoLinkContiene(TipoLinkContiene l){
       if(l!=null)
              linkContiene.remove(l);
}


public Set<TipoLinkContiene> getLinkContiene(){
   if ( linkContiene.size() < MIN_MOLT)
          throw new RunTimeException(" Violata la molteplicità 1..* ");
   else (Set<TipoLinkContiene>) linkContiene.clone();
}

public int quantiConsegnato() {return linkContiene.size()}

public void inserisciTipoLinkEffettua(TipoLinkEffettua l){
   if(l!=null && l.getOrdine() == this)
          ManagerEffettua.inserisci(l);
}

public void eliminaTipoLinkEffettua(TipoLinkEffettua l){
   if(l!=null && l.getOrdine() == this)
          ManagerEffettua.elimina(l);
}

public void inserisciPerManagerEffettua(ManagerEffettua m){
   if(m!=null)
          linkEffettua = m.getLink();
}
```

```
  public void eliminaPerManagerEffettua(ManagerEffettua m){
    if(m!=null)
        linkEffettua = null;
  }




  public static enum Stato = {  PREPARAZIONE , CONSEGNA, COMPLETATO };
  Stato statocorrente = Stato.PREPARAZIONE;

  Rider rider; //IMPORTANTE

  public Stato getStato(){return statocorrente;}

  public void fired(Evento e){
      TaskExecutor.getIstance().perform( new OrdineFired(this,e));
  }
}



//SOTTOCLASSE ORDINEPRIORITARIO
public class OrdinePrioritario extends Ordine{
  private double sovraprezzo;

  public OrdinePrioritario( String c, double s){
    super(c);
    sovraprezzo = s;
  }


  public double getSovraprezzo() { return sovraprezzo;}
  public void setSovraprezzo( double s) { this.sovraprezzo = s;}


}
```

```
//ASSOCIAZIONE EFFETTUA

public class TipoLinkEffettua{
      private Cliente cliente;
      private Ordine ordine;

      public TipoLinkEffettua ( Cliente c, Ordine o){
             if (c == null || o == null) throw new RuntimeException("link non valido");
            this.cliente = c;
            this.ordine  = o;
      }
      public Cliente getCliente(){ return cliente;}
      public Ordine getOrdine(){ return ordine ;}


      public boolean equals(Object o){
            if(o!=null && this.getClass().equals(o.getClass())){
                  TipoLinkEffettua l = (TipoLinkEffettua) o;
                  return ordine == l.ordine && cliente == l.cliente;
            }
            else return false;
      }
   public int hashCode() { return cliente.hashCode() + ordine.hashCode();}


}


public class ManagerEffettua {
   private TipoLinkEffettua link;
   private ManagerEffettua( TipoLinkEffettua l ) { this.link = l; }
   public TipoLinkEffettua getLink() {return link;}


   public void static inserisci ( TipoLinkEffettua l ) {
         if(l!=null){
            ManagerEffettua m = new ManagerEffettua( l);
            l.getOrdine().inserisciPerManagerEffettua(m);
            l.getCliente().inserisciPerManagerEffettua(m);
         }
   }
```

```
    public void static elimina( TipoLinkEffettua l ) {
        if(l!=null){
            ManagerEffettua m = new ManagerEffettua(l);
            l.getOrdine().eliminaPerManagerEffettua(m);
            l.getCliente().eliminaPerManagerEffettua(m);
        }
    }
}


//ASSOCIAZIOINE CONTIENE

public class TipoLinkContiene{
        private Pizza pizza;
        private Ordine ordine;
        private int quantita;


        public TipoLinkContiente( Pizza p, Ordine o, int q){
            if (c == null || o == null) throw new RuntimeException("link non valido");
            ordine = o;
            pizza  = p;
            quantita = q;
        }

        public int getQuantita() { return quantita;}
        public Ordine getOrdine(){ return ordine;}
        public Pizza getPizza(){ return pizza;}


        public boolean equals(Object o){
            if(o!=null && this.getClass().equals(o.getClass())){
                TipoLinkContiene l = (TipoLinkContiene) o;
                return ordine == l.ordine && pizza== l.pizza;
            }
            else return false;
        }
    public int hashCode() { return pizza.hashCode() + ordine.hashCode();}
}
```

//ORDINE FIRED

```
package Ordine;

class OrdineFired implements Task{
	private boolean eseguita = false;
	private Ordine ordine ;
	private Evento e;

	public OrdineFired(Ordine o, Evento e){
		ordine  = o;
		this.e = e;
	}


	public synchronized void esegui(){
		if ( eseguita == true || e.getDestinatario()!=null && e.getDestinatario()!=ordine )
			return;
		eseguita = true;

		switch(ordine.getStato()){
			case PREPARAZIONE:
			  if ( e.getClass().equals(Partenza.class) ){
				Partenza p = (Partenza) e;
				ordine.rider = p.getPayload();
				Environment.nuovoEvento(
					new Notifica(ordine , ordine.getLinkEffettua().getCliente()));
				ordine.statocorrente = Stato.CONSEGNA;
			  }
			  break;
			case CONSEGNA:
				if ( e.getClass().equals(Consegnato.class)){
				  Consegnato c = (Consegnato) e;
				  ordine.setLinkConsegnato(ordine.rider);
				  ordine.statocorrente = Stato.COMPLETATO;
				}
				break;
			case COMPLETATO:
				break;
```

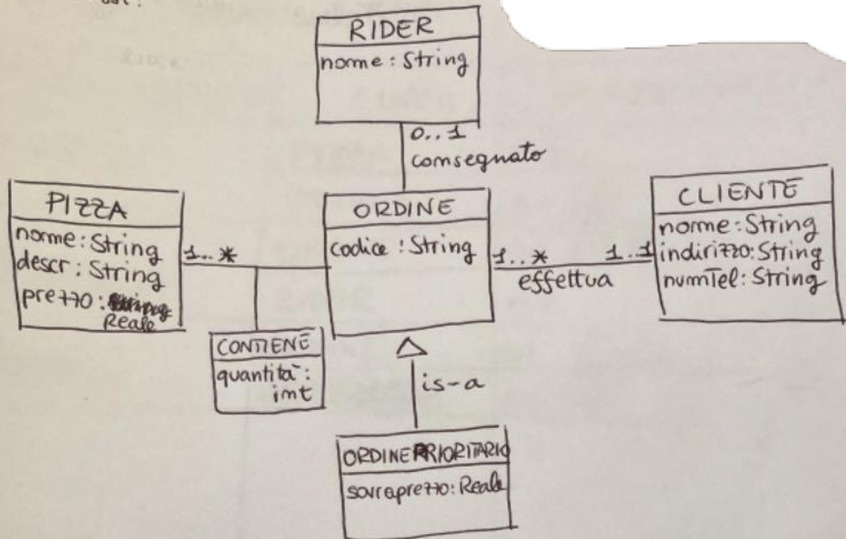
```
            default:
                throw new RunTimeException("Stato non trovato");
        }
    }


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

//ATTIVITA PRINCIPALE

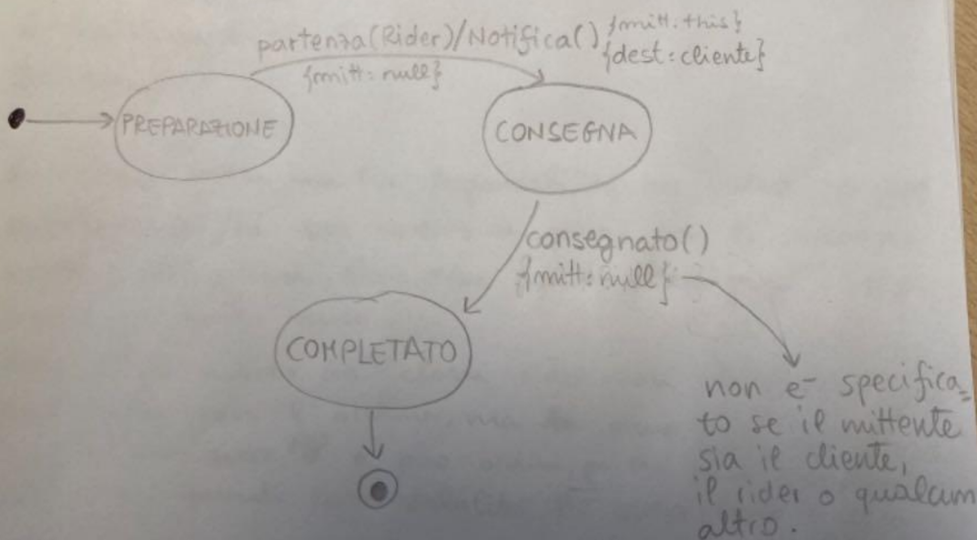
```
public class AttivitaPrincipale implements Runnable{
   private boolean eseguita = false;

   private HashSet<Rider> insRider;
   private HashSet<Ordine> insOrdini;


   public AttivitaPrincipale (HashSet<Rider> insRider,  HashSet<Ordine> insOrdini){
         this.insRider = insRider;
         this.insOrdini = insOrdini;
   }


   public synchronized void run(){
      if ( eseguita == true) return;
      eseguita = true;


      Verifica v = new Verifica(insRider, insOrdini);
      TaskExecutor.getIstance().perform(v);
      boolean ok = v.getResult()


      if(!ok) segnaliIO.richiediAggiornamento();
      else{
         ConsegnaOrdini co = new ConsegnaOrdini(insRider, insOrdini);
         Thread t1 = new Thread(co);
         t1.start();
         Analisi a = new Analisi(insRider, insOrdini);
         Thread t2 = new Thread(a);
         t2.start();
         try {
               t1.join();
               t2.join();
         }catch(InterruptedException e){
               e.printStackTrace();
         }
      }
      String   r = a.getResult();
```

```
    SegnaliIO.report(r);
  }
  public synchronized boolean estEseguita() { return eseguita;}
}


//ATTIVITA ATOMICA

public class Verifica implements Task{
  private boolean eseguita = false;
  private boolean ok;
  private HashSet<Rider> insRider;
  private HashSet<Ordine> insOrdine;


  public Verifica(insOrdine, insRider){
    this.insRider = insRider;
    this.insOrdine = insOrdine;
  }


  public synchronized void esegui(){
    if(eseguita) return;
    eseguita = true;
    ok = insRider.size()==insOrdine.size();
  }

  public synchronized boolean estEseguita(){ return eseguita;}

  public synchronized boolean getResult(){
    if (!eseguita) throw new RuntimeException("risultato non disponibile");
    return ok;
  }
}
```

Diag. Classi:



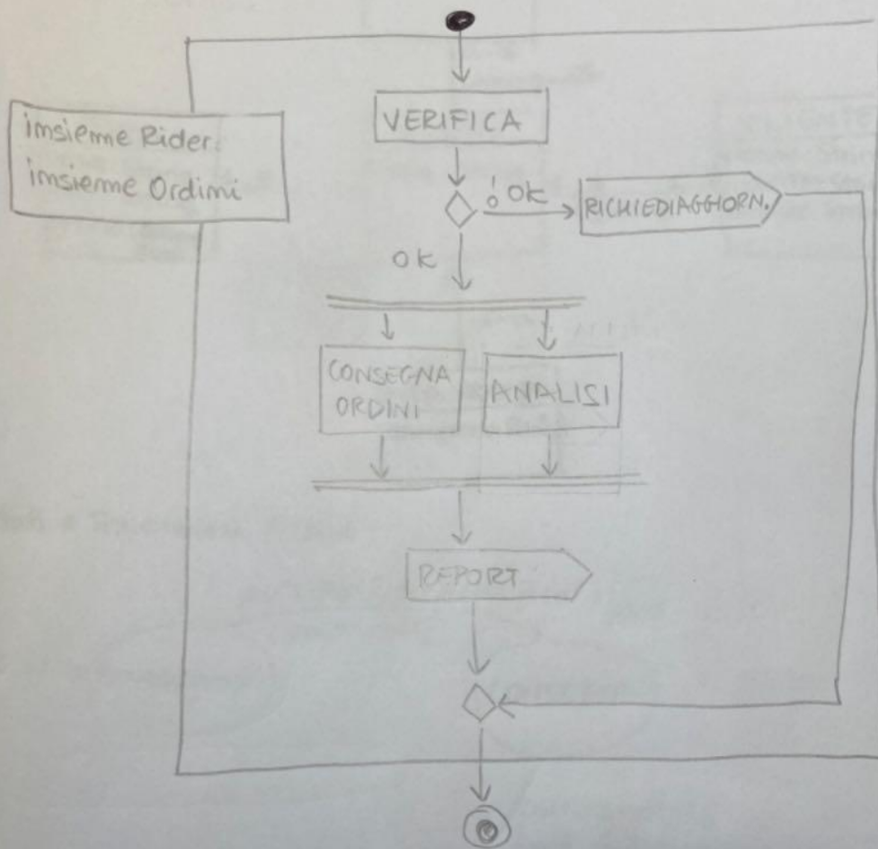
Stati e Transizioni Ordine:



var aux:

Rider rider

Diagramma Attività



ConsegnaOrdini e Analisi sono 2 sottoattività complesse.

Tabella Responsabilità:

ASSOCIAZIONE	CLASSI	HA RESPONSABILITÀ
Contiene	PIZZA	NO
	ORDINE	SI (1)
Consegnato	ORDINE	SI (1) (2)
	RIDER	NO
Effettua	ORDINE	SI (1) (2)
	CLIENTE	SI (1)

1- molteplicità
2- operazioni
3- requisiti

La classe pizza non ha responsabilità su Ordine né per molteplicità, né per operazioni, poiché non le interessa nulla dell'ordine. Non deve quindi "navigare" l'associazione in quel senso.

Allo stesso modo la classe rider non deve comunicare mai con l'ordine, ma deve solo ricevere. Può anche avere zero o più ordini, poiché non specificato. Non ha quindi responsabilità su Ordine.