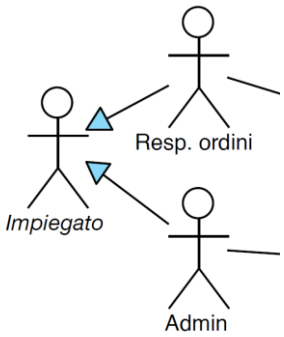
INGEGNERIA DEL SOFTWARE A.A. 2012-13
PROVA SCRITTA DEL 18 SETTEMBRE 2013 (SOLUZIONE)

Esercizio 1

Un negozio di articoli sportivi desidera acquistare un'applicazione che permetta l'acquisto on-line di alcuni articoli. I **clienti** si dovranno registrare. Il cliente potrà scegliere gli articoli secondo alcuni parametri raccolti nel catalogo definito **dall'amministratore del negozio on-line**. Si potranno acquistare più articoli in una sola transazione, questi articoli formeranno un carrello. Al termine dell'acquisto saranno disponibili diversi metodi di pagamento. Gli ordini effettuati dovranno essere smaltiti dal **reparto ordini**, che dovranno inviare la merce all'indirizzo desiderato.

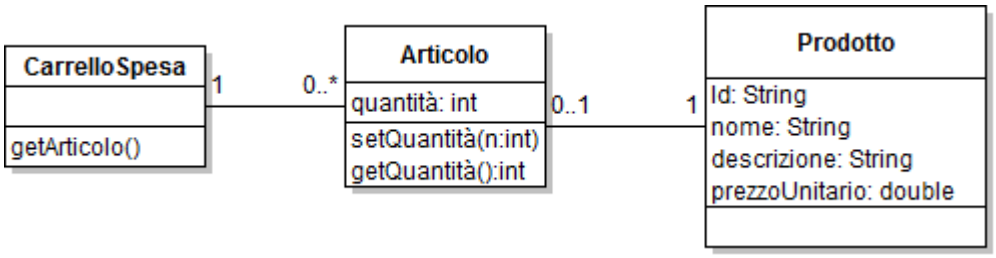
a) Quali sono gli attori dell'applicazione? E perchè?
Clienti, Amministratore del negozio e responsabile reparto ordini



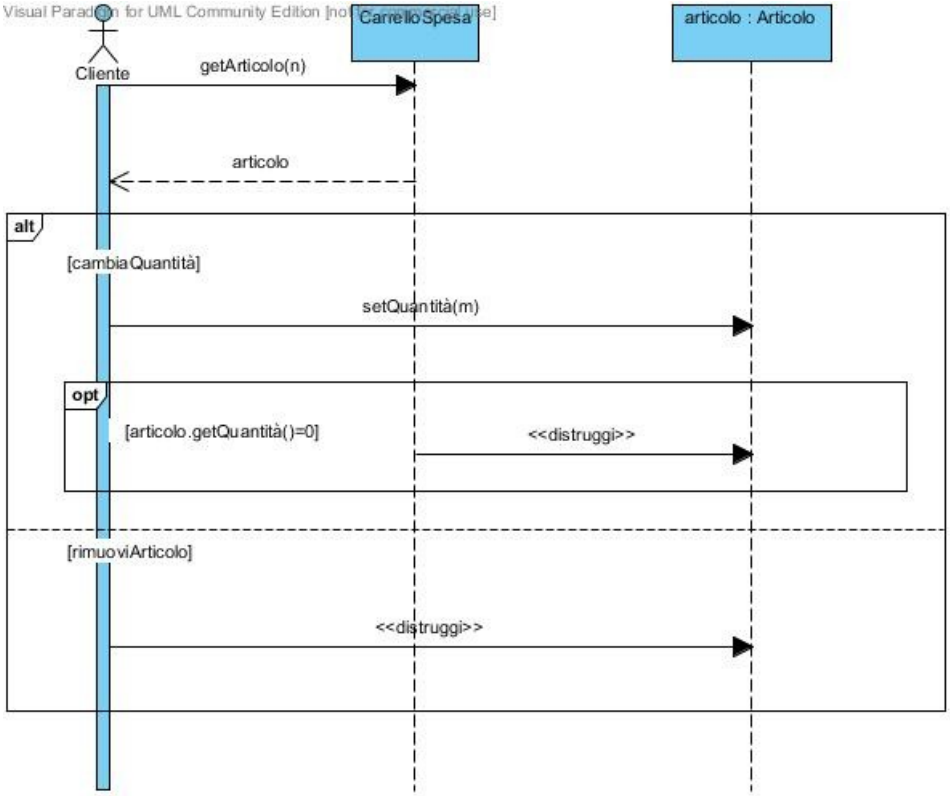
b) Elencare (senza abbozzarli!) almeno tre casi d'uso di questa applicazione diversi da quello indicato al punto c)
Modifica catalogo (da parte dell'amminstratore), evadi ordine (da parte del responsabile reparto ordine) e consulta catalogo (da parte del cliente)

c) Disegnare il diagramma delle classi (di analisi) relativo al seguente caso d'uso

Caso d'uso: AggiornaCarrello
ID: UC2
Attori: Cliente
Precondizioni: 1. Il contenuto del carrello è visibile
Sequenza degli eventi: 1. Il caso d'uso inizia quando il Cliente seleziona un articolo nel carrello. 2. Se il Cliente seleziona "rimuovi articolo" 2.1 Il Sistema elimina l'articolo dal carrello. 3. Se il Cliente digita una nuova quantità 3.1 Il Sistema aggiorna la quantità dell'articolo presente nel carrello
Postcondizioni: 1. Il contenuto del carrello è stato aggiornato
Sequenza alternativa 1: 1. In qualunque momento il Cliente può abbandonare la pagina del carrello
Postcondizioni:



d) Rappresentare con un sequence diagram il caso d'uso **AggiornaCarrello**. Usare gli stessi campi e operazioni introdotte al punto precedente



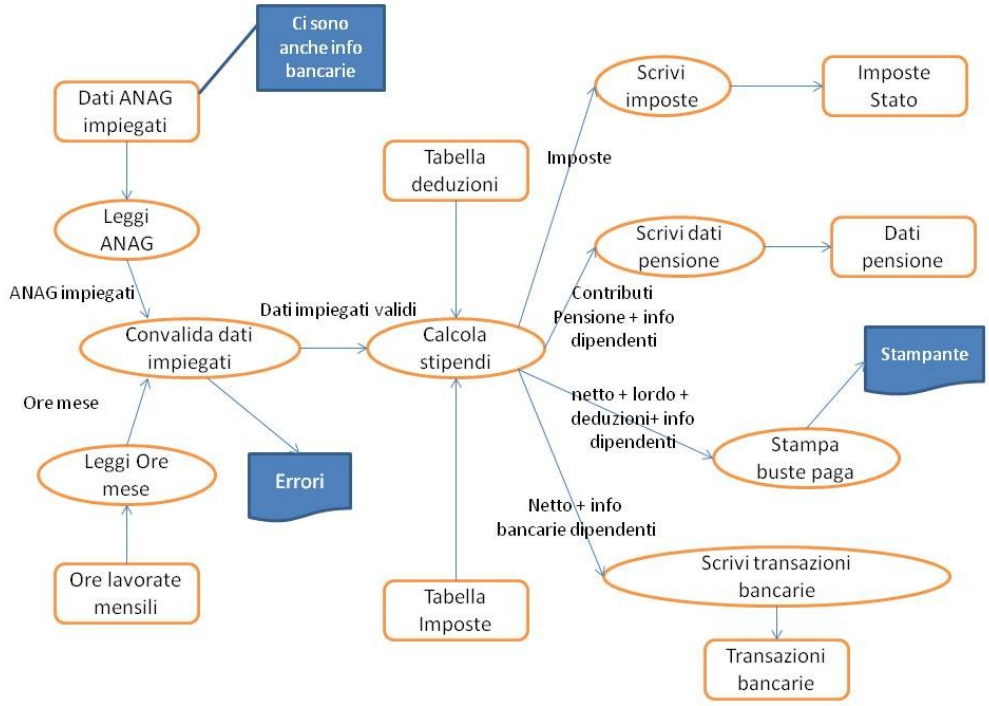
Esercizio 2

Si supponga di dover progettare un modulo software **pagamento degli stipendi** che va a completare un complesso legacy system che integra tutti i processi di business rilevanti di un'azienda (vendite, acquisti, gestione magazzino, contabilità etc.). Questo modulo (che dobbiamo implementare) legge il file impiegati (contenente tutti i dati anagrafici) e il file contenente le effettive ore lavorate (la paga è oraria e viene ricavate mediante un sistema automatico di rilevazione delle presenze). Dopo avere convalidato i dati degli impiegati la componente software **Calcola stipendi**, che deve essere presente nel modulo da implementare, legge le tabella delle imposte e delle deduzioni (anch'esse memorizzata su file) e calcola lo stipendio mensile lordo per ogni impiegato, le varie deduzioni da effettuare, le imposte da pagare e poi lo stipendio mensile netto. Alla fine del calcolo degli stipendi, alcune componenti software di output devono scrivere tre file: (1) imposte da versare allo Stato, (2) contributi pensione e (3) transazioni bancarie da effettuare. Questi file saranno elaborati da altri programmi o trasferiti a chi di interesse (ad esempio il file *transazioni bancarie* verrà trasferito alla banca per effettuare il bonifico ai dipendenti) una volta che i dettagli per tutti gli impiegati sono stati calcolati. Infine il modulo **pagamento degli stipendi** stampa una busta paga per gli impiegati, con i dati del pagamento netto, le deduzioni effettuate e le imposte da pagare.

a) Specificare il nome dello stile architetturale tra quelli che abbiamo visto a lezione che ritenete più appropriato per il modulo software sopra descritto motivando brevemente la vostra scelta ed elencando eventuali pro e i contro dello stile scelto

Pipe and filter

b) Disegnare un diagramma che descrive l'architettura del sistema in accordo a tale stile, commentando brevemente a parole il funzionamento e marcando (quando utile) input e output dei vari componenti software



Esercizio 3

Dato il seguente codice che simula un semplice distributore di bevande:

```
public class Distributore {
        private Map<Integer,Tessera> tessere;
        private Map<String,Bevanda> bevande;
        private List<Colonna> colonne;
        public Distributore() {
                this.tessere = new HashMap<Integer,Tessera>();
                this.bevande = new HashMap<String,Bevanda>();
                this.colonne = new ArrayList<Colonna>();
        }
        public void aggiungiBevanda(String cod, String nome, double prezzo) {
                Bevanda b = new Bevanda(cod, nome, prezzo);
                bevande.put(cod, b);
        }
        public double getPrice(String code) throws BevandaNonValida {
                if(!bevande.containsKey(code)) throw new BevandaNonValida(code);
                Bevanda b = bevande.get(code);
                return b.getPrezzo();
        }
        public String getName(String code) throws BevandaNonValida {
                Bevanda b = bevande.get(code);
                if(b==null) throw new BevandaNonValida(code);
                return b.getNome();
        }
        public void caricaTessera(int codice, double credito) {
                Tessera t = new Tessera(codice, credito);
```

```
                tessere.put(codice, t);
        }
        public double leggiCredito(int codice) throws TesseraNonValida {
                Tessera t = tessere.get(codice);
                if(t!=null) return t.getCredito();
                throw new TesseraNonValida();
        }
        // Il distributore è costituito da quattro colonne in cui sono disposte le lattine delle bevande. Ogni colonna
        contiene un unico tipo di bevanda ed è caratterizzata dal numero di lattine contenute e dal tipo di bevanda che
        contiene. In partenza tutte le colonne sono vuote. Quando viene ricaricato il distributore viene assegnato ad ogni
        colonna un tipo di bevanda ed il numero di lattine presenti; a questo scopo si usa il metodo aggiornaColonna() che
        riceve come parametro il numero della colonna, il tipo di bevanda e il numero di lattine presenti nella colonna. Le
        colonne sono numerate a partire da 1
        public void aggiornaColonna(int numeroColonna, String nomeBevanda, int numeroLattine) {
                colonne.add(numeroColonna-1, new Colonna(nomeBevanda,numeroLattine));
        }
        public int lattineDisponibili(String code) throws BevandaNonValida {
                String nome = getName(code); int count=0;
                for(Colonna element: colonne){
                        if(element!=null &&
                                element.getNome().equals(nome)){
                                count += element.getQ();
                        }
                }
                return count;
        }
        public int eroga(String codiceBevanda, int codiceTessera) throws BevandaNonValida, BevandaEsaurita,
        TesseraNonValida, CreditoInsufficiente {
                double prezzo = getPrice(codiceBevanda);
                if(leggiCredito(codiceTessera)<prezzo) throw new CreditoInsufficiente();
                String nome = getName(codiceBevanda);
                Colonna colonna = null;
                for(Colonna col: colonne){
                        if(col!=null && col.getNome().equals(nome) && col.getQ() > 0){
                                colonna = col;
                                break;
                        }
                }
                if (colonna==null) throw new BevandaEsaurita();
                Tessera tessera = tessere.get(codiceTessera);
                tessera.decrementaCredito(prezzo);
                colonna.diminuisciQ();
                return colonne.indexOf(colonna)+1;
    }
}

class Bevanda {
        private String codice;
        private String nome;
        private double prezzo;
        public Bevanda(String cod, String n, double cred){
                this.codice=cod;
                this.nome=n;
                this.prezzo=cred;
        }
        public String getCod(){return this.codice;}
        public String getNome(){return this.nome;}
        public double getPrezzo(){return this.prezzo;}
}

public class BevandaEsaurita extends Exception {
        public BevandaEsaurita() {
                super("Bevanda esaurita");
        }
}

public class BevandaNonValida extends Exception {
  public BevandaNonValida(String code) {
        super("Bevanda non valida: " + code);
  }
  public BevandaNonValida(){
        super("Bevanda non valida");
  }
}

class Colonna {
        private String nome;
```

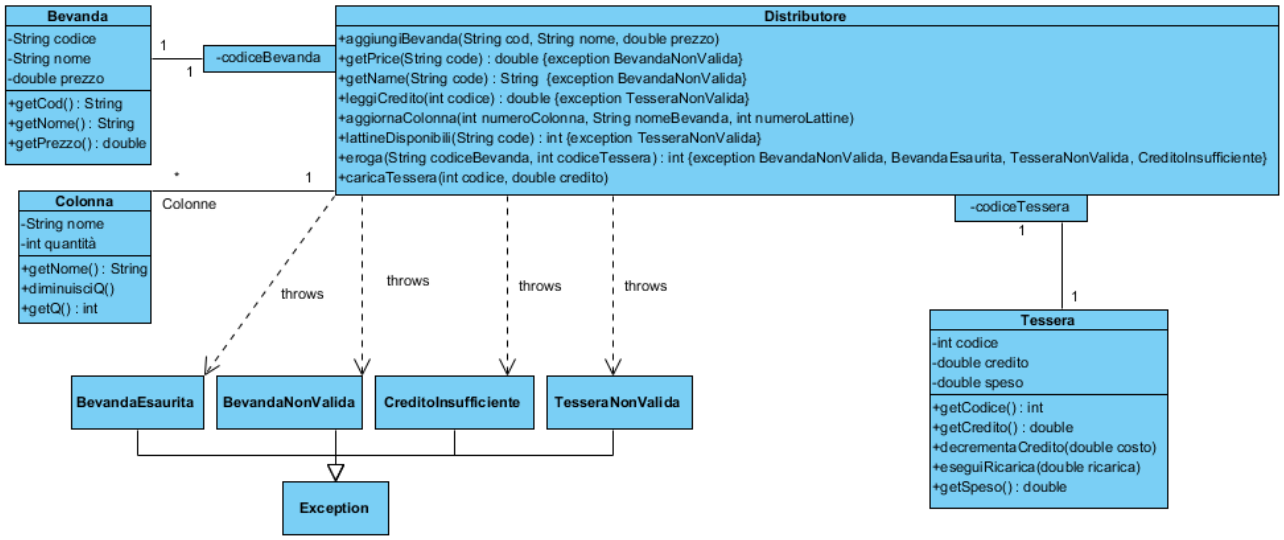
```
        private int quantità;
        public Colonna (String nome, int quantità){
                this.nome=nome;
                this.quantità=quantità;
        }
        public String getNome(){return this.nome;}
        public void diminuisciQ(){this.quantità--;}
        public int getQ(){
                return this.quantità;
        }
}

public class CreditoInsufficiente extends Exception {}

public class Tessera {
        private int codice;
        private double credito;
        private double speso;
        public Tessera (int cod, double cred){
                this.codice=cod;
                this.credito=cred;
                this.speso = 0;
        }
        public int getCodice(){return this.codice;}
        public double getCredito(){return this.credito;}
        public void decrementaCredito(double costo){
                this.credito -= costo;
                this.speso +=costo;
        }
        public void eseguiRicarica(double ricarica){this.credito += ricarica;}
        public double getSpeso() {
                return speso;
        }
}

public class TesseraNonValida extends Exception {
        public TesseraNonValida(){
                super("Tessera non valida");
        }
}
```

a) Effettuare un'operazione di **ridocumentazione** mostrando il class diagram relativo (attenzione a lezione non abbiamo visto come si rappresenta in UML un HashMap, vedere associazioni qualificate in M. Fowler o escogitare un modo per rappresentarle)



b) Dato il seguente SetUp scrivere due casi di test JUnit per testare: il `CreditoInsufficiente` e la `bevandaEsaurita`

```
public void setUp() {
    distributore = new Distributore();

    distributore.aggiungiBevanda("A", "Acqua", 0.20);
```

```
    distributore.aggiungiBevanda("B", "Coca", 0.30);
    distributore.aggiungiBevanda("C", "Birra", 1.00);

    distributore.caricaTessera(12, 5.5);
    distributore.caricaTessera(21, 10.0);
    distributore.caricaTessera(99, 0.25);

    distributore.aggiornaColonna(1, "Acqua", 40);
    distributore.aggiornaColonna(2, "Coca", 1);
    distributore.aggiornaColonna(3, "Birra", 50);
    distributore.aggiornaColonna(4, "Acqua", 50);
  }
```

```
public void testCreditoOK() {
                try {
                        distributore.eroga("A", 12);
                        assertTrue (true);
                } catch (BevandaNonValida e) {
                        fail();
                } catch (BevandaEsaurita e) {
                        fail();
                } catch (TesseraNonValida e) {
                        fail();
                } catch (CreditoInsufficiente e) {
                        fail();
                }

        }

        public void testCreditoKO() {
                try {
                        distributore.eroga("C", 99);
                        fail();
                } catch (BevandaNonValida e) {
                        fail();
                } catch (BevandaEsaurita e) {
                        fail();
                } catch (TesseraNonValida e) {
                        fail();
                } catch (CreditoInsufficiente e) {
                        assertTrue (true);
                }
        }

        public void testBevandaOK() {
                try {
                        distributore.eroga("A", 12);
                        assertTrue (true);
                } catch (BevandaNonValida e) {
                        fail();
                } catch (BevandaEsaurita e) {
                        fail();
                } catch (TesseraNonValida e) {
                        fail();
                } catch (CreditoInsufficiente e) {
                        fail();
                }
        }

        public void testBevandaKO() {
                try {
                        distributore.eroga("B", 12);
                        distributore.eroga("B", 12);
                        fail();
                } catch (BevandaNonValida e) {
                        fail();
                } catch (BevandaEsaurita e) {
                        assertTrue (true);
                } catch (TesseraNonValida e) {
                        fail();
                } catch (CreditoInsufficiente e) {
                        fail();
                }
        }
```