Documentazione Pa

Silviu Filote 1059252

February 2023

Contents

1 Progetto Java
- 1.1 Persona
- 1.2 Volo
- 1.3 Costrutti Java utilizzati

2 Progetto C++
- 2.1 Cliente
- 2.2 Staff
- 2.3 Admin
- 2.4 Costrutti C++ utilizzati

3 Progetto Haskell: bank management
- 3.1 getIdMaxList
- 3.2 insertInto
- 3.3 insertClienti
- 3.4 makeRandom
- 3.5 makeListRandom
- 3.6 thrd
- 3.7 fstt
- 3.8 orderMaxSaldo
- 3.9 getTuple
- 3.10 findIdClient
- 3.11 addSaldo
- 3.12 addSaldoId
- 3.13 removeCienteIdList
- 3.14 getClienteMaxSaldo

1 Progetto Java

Nel progetto di Java si pone come obiettivo quello di replicare la gestione di un aereoporto. L'areoporto viene trattato come una singola risorsa da cui recuperare informazioni di carattere generale, proprio per questo motivo si applica contemporaneamente il design pattern singleton e facade.
L'aereporto include:

- `Persona`, classe astratta con informazioni di carattere generale
- `Persona:Qualificato`, ossia il personale addetto ai lavori (Hostess, Piloti);
- `Persona:Passeggero`, ossia i clienti che usufruiscono del servizio;
- `Volo`, che contiene la lista di passeggi, la lista del personale (Hostess, Piloti) e infine ulteriori operazioni logiche associate al volo stesso;
- `Aereo` associato al volo;
- `Aereoporto` é una classe che implementa il design pattern facede e singleton, da quest'ultima si possono riperire informazioni generali

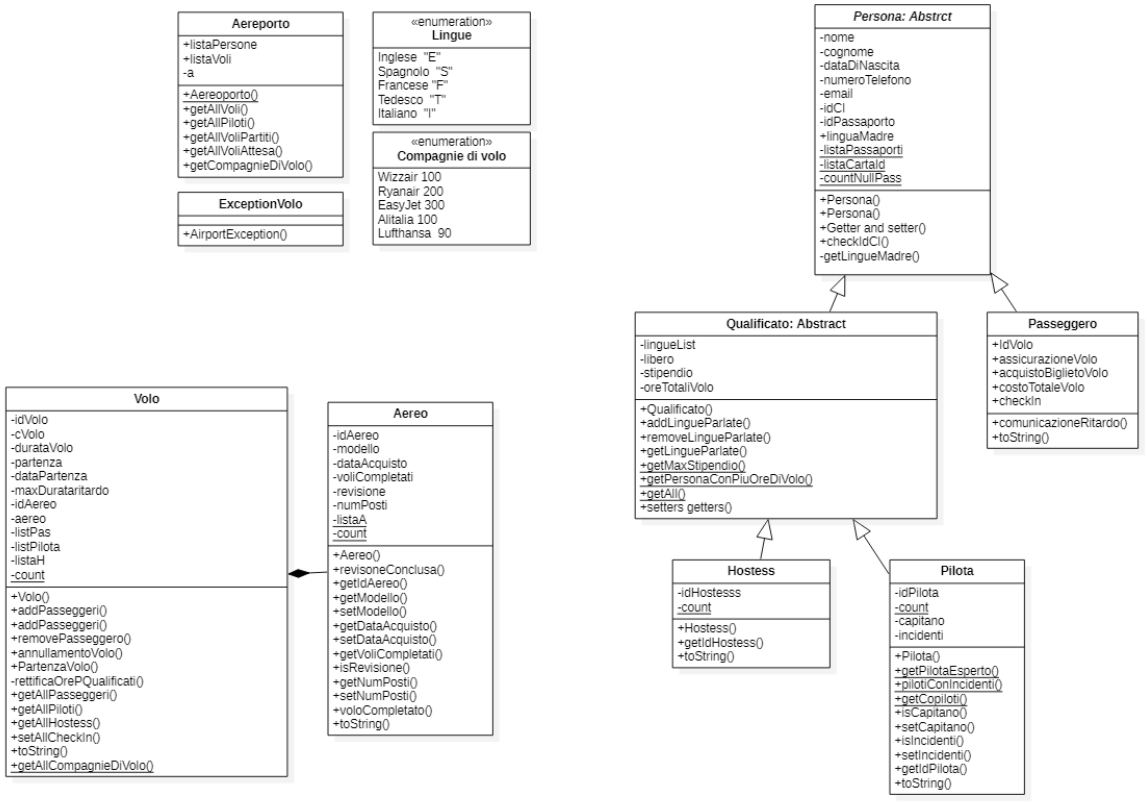

Figure 1: visione intera del class diagram associato al progetto di JAVA

Di seguito si elencano i casi d'uso più particolari.

1.1 Persona

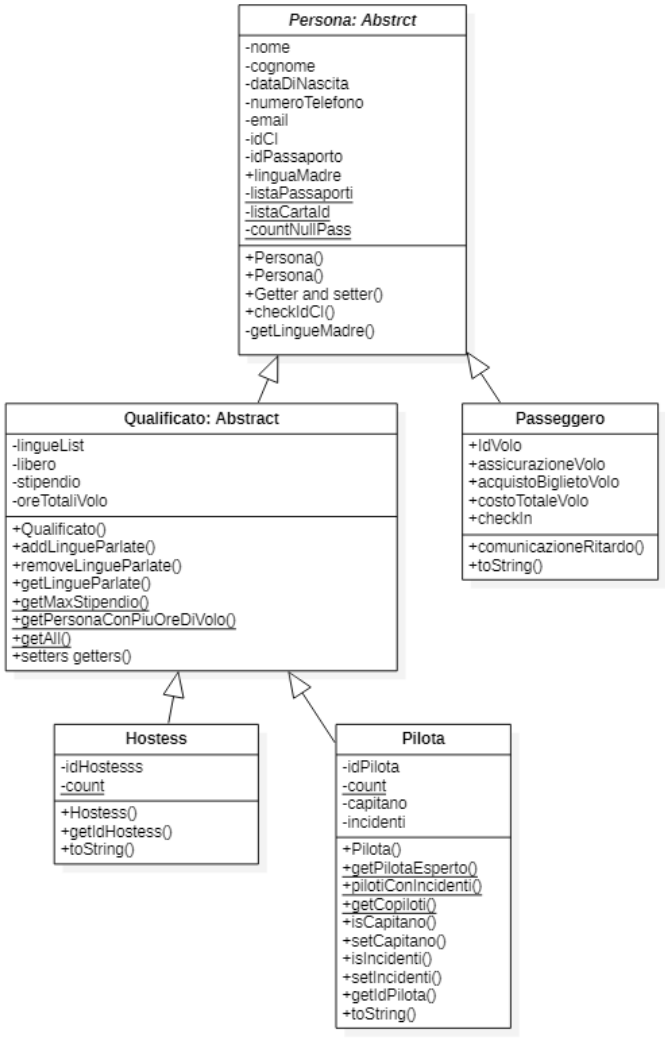

Figure 2: visione parziale del class diagram per rappresentare i casi d'uso di `Persona` e delle sotto-classi

Casi d'uso:

- In base alla prima lettera della carta d'identità `Persona.idCI`, si calcola direttamente la lingua madre della persona. Esempio: I43234 → italiano;
- L'overload di costruttori in `Persona` e `Passeggero` sono stati costruiti per permettere di associare o meno un passaporto;
- Si tiene conto di tutte le persone che non hanno associato un passaporto tramite `"NULL"+countNullPass`, presente in `Persona.listaPassaporti`;
- Al `Passeggero` viene associato un volo;
- Un `Passeggero` può scegliere se aggiungere al costo del volo l'assicurazione, ossia nel caso in cui il volo non venga effettuato viene recapitata al passaggero la nuova data del volo;
- Un `Passeggero` può arrivare in ritardo al volo;
- Il `Passeggero.prezzoBiglietto` viene determinato dato un enum;
- Un `Persona:Qualificato` può parlare più lingue;
- Il campo `Qualificato.libero` indentifica se una persona qualificata è associata ad un volo o meno e quindi risulta essere libera;

- Un volta completato un `Volo` ad ogni `Persona:Qualificato` viene sommato la durata del volo ad `Qualificato.oreTotaliVolo`;
- `Pilota.capitano` identifica se un pilota sia il capitano del volo che si prende tutte le responsabilità oppure il co-pilota;

1.2 Volo

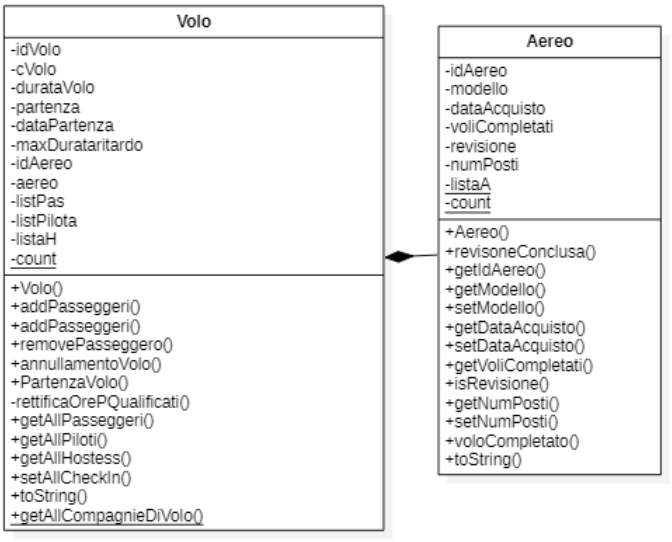

Figure 3: visione parziale del class diagram per rappresentare i casi d'uso di `Volo`

Casi d'uso:

- A un `Volo` si associa la lista di persone qualificate e di passeggeri;
- Un volta che si aggiunge passeggeri ad un `Volo` si sottraggono i posti a sedere, oppure nel caso si tolgano vengono aggiunti dei posti;
- Un `Volo` si può annullare e ai passeggeri che possiede l'assicurazione si conferma la seconda data del volo e si setta il check-in di tutti i passeggeri a false;
- Un volo può partire se tutti i passeggeri hanno effettuato il check-in, se non esiste alcun passeggero in ritardo e se la revisione dell'aereo è conforme;
- Se un `Aereo` completa un volo `Aereo.voliCompletati` si aggiorna, così come le ore totali del personale qualificato associato a tale volo;

1.3 Costrutti Java utilizzati

• Exception personalizzata • final • static • getters\setters
• lambda functions • enum • overload • override
• modificatori di accessibilità • abstract • facade pattern • singleton pattern
• iterators • generics methods • Ereditarietà • Varargs • Java Collections Framework → list

2 Progetto C++

Il progetto riproduce una realtà bancaria di cui gli attori principali sono: il cliente, lo staff e infine l'admin, ossia la persona che detiene più potere decisionale all'interno della gerarchia bancaria.

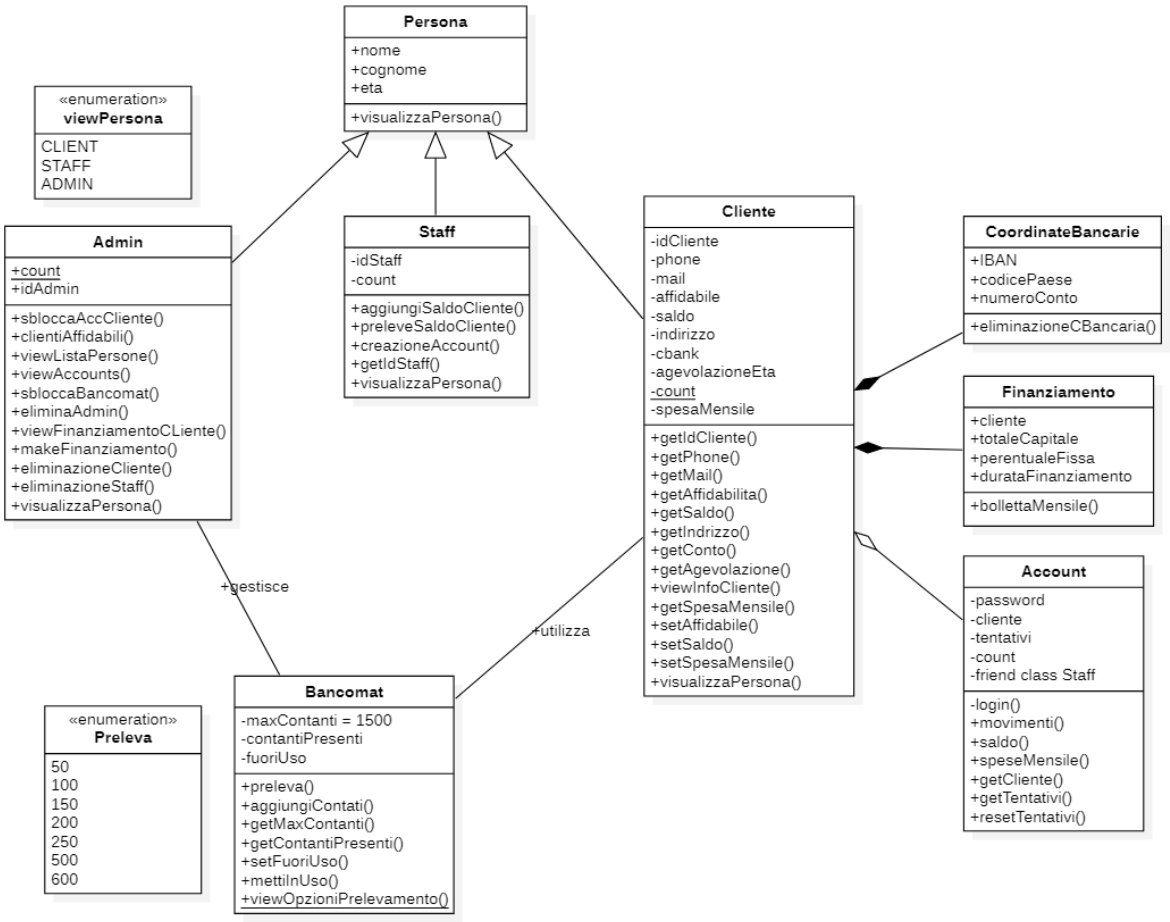

Figure 4: visione intera del class diagram associato al progetto di C++

Di seguito si analizzano i vari attori e le operazioni che ogni attore può eseguire.

2.1 Cliente

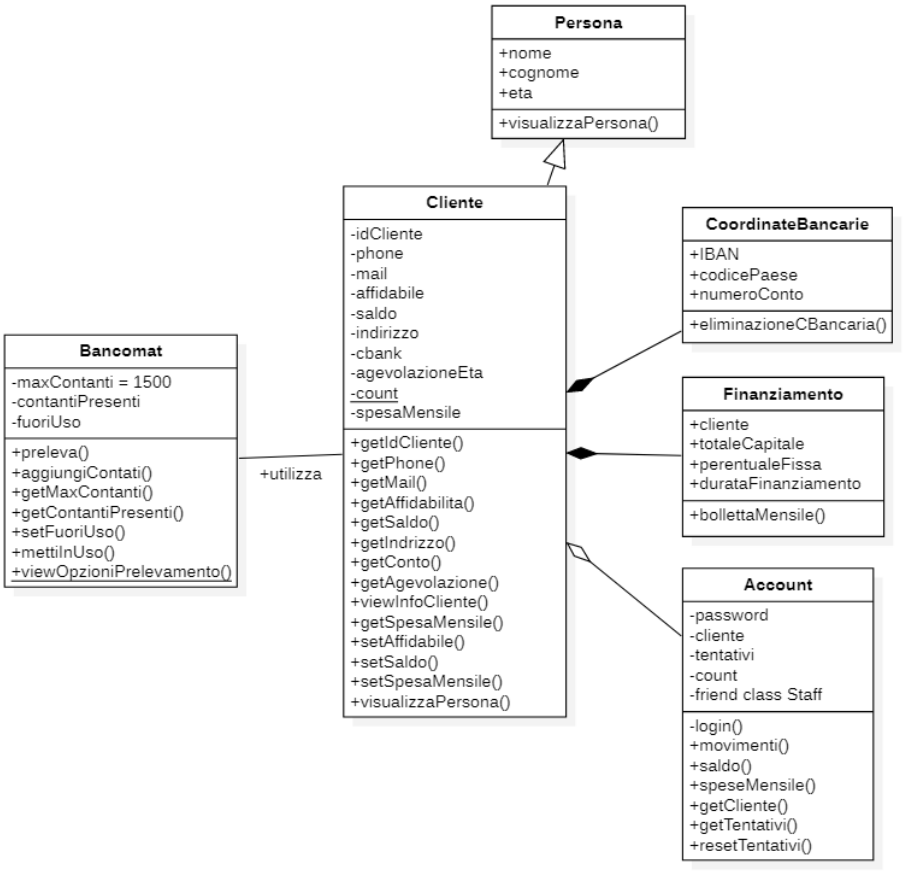

Figure 5: visione parziale del class diagram per rappresentare i casi d'uso di di `Cliente`

Casi d'uso:

- Il cliente alla creazione tramite costruttore gli vengono associati: informazioni generali, le coordinate bancarie e un saldo di partenza;
- La spesa mensile è una quota che inizialmente risulta essere pari a 0 e quest'ultima si aggiorna a 10 una volta che il cliente richiede allo staff un account di home banking. La spesa mensile inoltre ingloba al suo interno la tassa mensile dovuta a un possibile finanziamento da parte del cliente;
- Nel caso in cui un cliente possieda un'eta inferiore ai 26 anni, nel caso di finanziamento presentarà un tasso di interesse di 10% in meno;
- Il mancato pagamento di una spesa mensile da parte di un cliente pone il flag affidabile a false;
- Alla creazione di un bancomat vengono depositati n contanti oppure un massimo di 1500. Il cliente può interagire con il bancomat prelevando e in questo caso si aggiornerà il saldo del cliente.
- Esistono poi infine metodi di get e set per impostare e visualizzare informazioni generali;

2.2 Staff

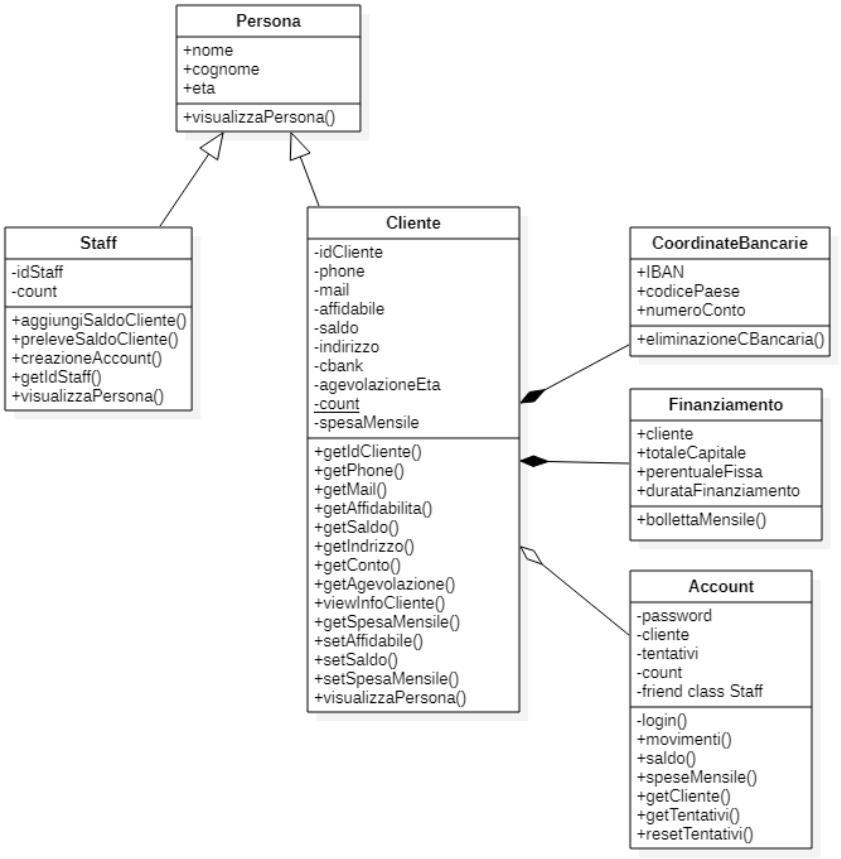

Figure 6: visione parziale del class diagram per rappresentare i casi d'uso di di `Staff`

Casi d'uso:

- Lo staff dato un cliente può aggiungere saldo;
- Permette ad un cliente di effettuare un prelievo e di conseguenza diminuisce il saldo totale del cliente;
- Lo staff può abilitare un account di home banking ad un cliente aggiungendo una spesa mensile per coprire i costi di 10€
- metodi di getter e setter

2.3 Admin

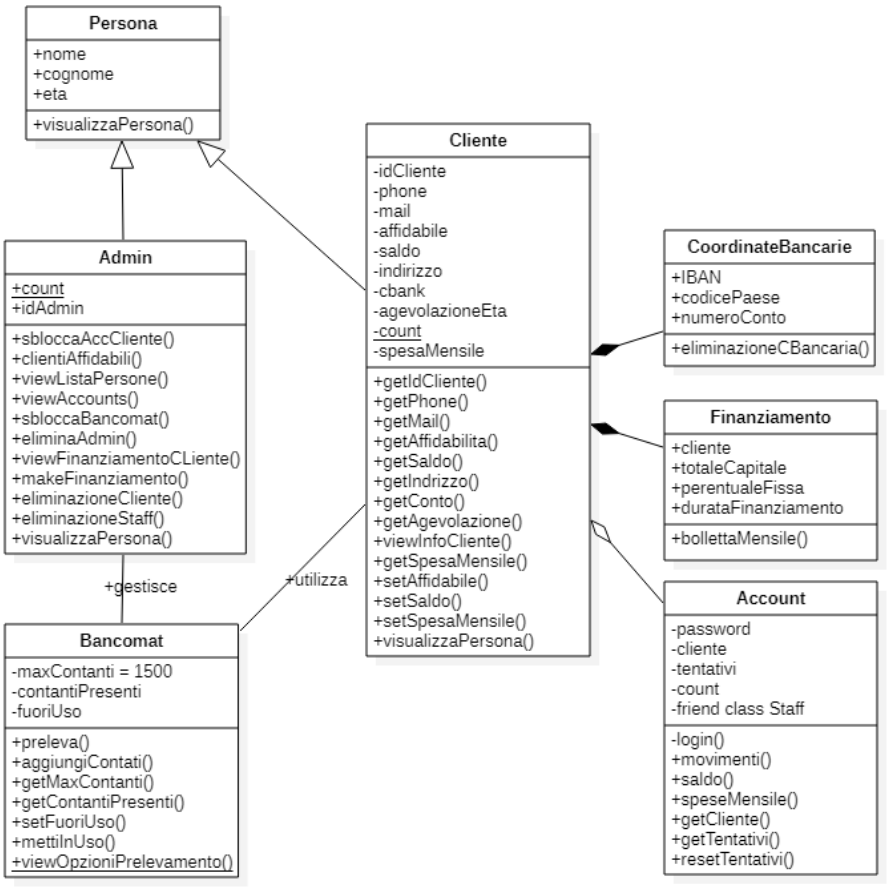

Figure 7: visione parziale del class diagram per rappresentare i casi d'uso di di `Admin`

Casi d'suo:

- Un admin può eliminare un dipendente dello staff, un admind e un cliente. L'eminazione del cliente è resa possibile se quest'ultimo non possiede alcun finanziamento in corso. Inoltre l'eminazione del cliente comporta la cancellazione: delle coordinate bancarie e dell'account di home banking se quest'ultimo ne possiede uno;
- L'admin può visualizzare i clienti affidabili, tutte le persone associate al sistema bancario, gli account di tutti i clienti registrati e i finanziamento in corso;
- L'utente nel caso dovesse sbagliare a loggarsi 3 volte (la password associata al cliente) l'account si blocca. L'admin può sbloccare tale account di home banking;
- Se il bancomat non dovesse avere più soldi al suo interno, a quest'ultimo si abilita il flag fuori uso. L'admin può abilitare tale bancomat e depositare al suo interno ulteriori contanti senza superare il tetto massimo imposto.
- Infine l'admin gestisce i finanziamenti dei clienti;
- All'interno del codice, cosi come nel caso di bancomat, può essere interpretata come "serve un admin per"

```
void Bancomat::distruggiBancomat(Admin * const a){
   if(a != NULL){
       delete(this);
   }
}
```

2.4 Costrutti C++ utilizzati

• Ereditarietà • Costruttori/distruttori • Variabili e metodi statici
• Utilizzo di const • Iterators •Delete/free • New/malloc
• overload • override, virtual, override • costruttore friend • enum e casting
• modificatori di accessibilità • suddivisione interfaccia, implementazione • final • costanti
• eccezioni • smart pointers • list inizializator • Function Template
• default parameter • foreach • Containers $\rightarrow$ $vector < T >$ e metodi a loro associati

3 Progetto Haskell: bank management

Il progetto di haskell data una lista di clienti aderenti a un sistema bancario vengono imagazzinati all'interno di una lista sottoforma di tupla. Per ogni cliente/tupla si denotano: identificativo, nome del cliente e saldo.

$$(ID, nome, saldo) = (int, String, int)$$

```
clienti = [(1, "Marcello", 1000), (2, "Martina", 500), (3, "Leonardo", 1500),
(4, "Giovanni", 2000)]
```

Di seguito si elencano le funzioni che vengono applicate alla lista di clienti e quelle di supporto per poterle manipolarle:

3.1 getIdMaxList

Calcola il massimo ID di una lista di clienti.

```
getIdMaxList :: [(Int, String, Int)] -> Int
getIdMaxList [] = error "Error list is not appropriate"
getIdMaxList [x] = fstt x
getIdMaxList (x:xs) | (getIdMaxList xs) > fstt x = getIdMaxList xs
                    | otherwise                  = fstt x
```

3.2 insertInto

Dato un cliente/tupla di cui si passano solo il nome e il saldo, veiene aggiunto in testa a una lista di clienti. L'ID viene aggiunto in maniera automatica utilizzando la funzione ***getIdMaxList***.

```
insertInto :: (String, Int) -> [(Int, String, Int)] -> [(Int, String, Int)]
insertInto (a,b) xs =  (getIdMaxList xs + 1, a, b) : xs
```

3.3 insertClienti

Inserimento di una lista di clienti all'interno di della lista Clienti dichierata a inizio file.

```
insertClienti :: [(Int, String, Int)] -> [(Int, String, Int)]
insertClienti xs = xs ++ clienti
```

3.4 makeRandom

Data una funzione matematica definita come input viene applicata al saldo del cliente.

```
makeRandom :: (Int -> Int) -> (Int, String, Int) -> (Int, String, Int)
makeRandom f (a, b, c) = (a, b, (f c))
```

3.5 makeListRandom

Viene utilizzata la funzione di supporto ***makeRandom*** a una lista di clienti per modificare a tutti il saldo.

```
makeListRandom :: (Int -> Int) -> [(Int, String, Int)] -> [(Int, String, Int)]
makeListRandom f xs =  [makeRandom f x | x <- xs ]
```

3.6 thrd

Dato un cliente restituisce il proprio saldo.

```
thrd :: (a, b, c) -> c
thrd (_, _, c) = c
```

3.7 fstt

Dato un cliente restituisce il proprio ID.

```
fstt :: (a, b, c) -> a
fstt (a, _, _) = a
```

3.8 orderMaxSaldo

Ordinamento crescente rispetto al saldo di una lista clienti.

```
orderMaxSaldo :: [(Int, String, Int)] -> [(Int, String, Int)]
orderMaxSaldo [] = []
orderMaxSaldo xs = sortBy (compare `on` thrd) xs
```

3.9 getTuple

Restitusce i clienti uno alla volta data una lista di clienti.

```
getTuple :: [(a)] -> (a)
getTuple [] = error "Error list is not appropriate"
getTuple (x:xs) = x
```

3.10 findIdClient

Fornito un ID restituisce il cliente con tale ID impegando la funzione di supporto ***getTuple***.

```
findIdClient :: Int -> [(Int, String, Int)] -> (Int, String, Int)
findIdClient x [] = error "Error list is not appropriate"
findIdClient x (y:ys) = getTuple [y | y <- ys, fstt(y) == x]
```

3.11 addSaldo

Aggiunge/toglie saldo a un cliente.

```
findIdClient :: Int -> [(Int, String, Int)] -> (Int, String, Int)
findIdClient x [] = error "Error list is not appropriate"
findIdClient x (y:ys) = getTuple [y | y <- ys, fstt(y) == x]
```

3.12 addSaldoId

Dato un ID aggiunge/toglie un determinato saldo al cliente con tale ID, vengono impegate le funzioni di support ***addSaldo*** e ***findIdClient***.

```
addSaldoId :: Int -> Int -> [(Int, String, Int)] -> (Int, String, Int)
addSaldoId x y [] = error "Error list is not appropriate"
addSaldoId x y xs = addSaldo y (findIdClient x xs)
```

3.13 removeCienteIdList

Dato un ID rimuove un determinato cliente dalla lista clienti.

```
removeCienteIdList :: Int -> [(Int, String, Int)] -> [(Int, String, Int)]
removeCienteIdList x [] = error "Error list is not appropriate"
removeCienteIdList i xs = smaller
                          where smaller = [ x | x <- xs, fstt x /= i]
```

3.14 getClienteMaxSaldo

Restituisce il cliente con il saldo piu alto utlizzando la funzione di supporto ***orderMaxSaldo***.

```
getClienteMaxSaldo :: [(Int, String, Int)] -> (Int, String, Int)
getClienteMaxSaldo [] = error "Error list is not appropriate"
getClienteMaxSaldo [x] = x
getClienteMaxSaldo xs =  head (reverse (orderMaxSaldo xs))
```