UNIVERSITÀ DEGLI STUDI DI BERGAMO

Dipartimento di Ingegneria Gestionale, dell'Informazione e della Produzione

Documentazione progetti per il corso di

PROGRAMMAZIONE AVANZATA

Studente:

Giorgia BRESSANELLI

Matricola n. 1053903

ANNO ACCADEMICO 2023 / 2024

Indice

1 Introduzione

I progetti realizzati in C++ e in Java forniscono una base per la gestione di un campo di emergenza per protezione civile.
In particolare, in un campo possono essere presenti più gruppi ed ognuno è composto da volontari categorizzati in Capi Squadra e Volontari Semplici.
A loro disposizione possono essere messi dei Veicoli, di proprietà dell'associazione a cui appartengono.

Il programma in C++ si occupa della creazione e proprietà dei gruppi, mentre il programma in Java è riservato alla vera e propria attività di campo: ingressi e uscite dei gruppi e dei veicoli, quali oggetti sono presenti nel campo oppure sono impegnati in operazioni esterne.

La sezione inerente ad Haskell invece si occupa di mostrare qualche funzionalità del linguaggio.

2 Progetto in C++

2.1 Contesto

Per gestire la parte logisitca in un campo di protezione civile è necessario che tutti i gruppi richiamati per partecipare all'emergenza vengano registrati.
Tutt'oggi, ogni gruppo si presenta con un documento cartaceo in cui sono elencati tutti i volontari e tutti i mezzi messi a disposizione e, in un secondo momento, la sezione riservata alla segreteria del campo si occupa dell'inserimento di tutti i dati nel database del campo stesso.

Una volta che tutti i volontari e i veicoli vengono registrati al portale, la segreteria è in grado di comunicare la posizione e lo stato di qualsiasi entità schedata, in qualsiasi momento. Questo progetto si occupa appunto della parte di inserimento manuale dei dati: quali sono i gruppi, chi sono e cosa fanno i volontari di ogni gruppo, quali mezzi vengono sfruttati, ecc... La seconda parte, quella più logistica, è stata invece implementata nel progetto Java.

Il software presenta un'interfaccia user friendly, in modo da semplificare l'inserimento dei dati.
In particolare, il volontario di segreteria può

- inserire un nuovo Gruppo;
- inserire un nuovo Volontario (Semplice o CapoSquadra) o un Veicolo selezionando il gruppo in cui registrarlo;
- cancellare un gruppo, un veicolo o un volontario;
- visualizzare i gruppi, i volontari e i veicoli che sono stati registrati.

Nel capitolo verranno elencate e descritte le caratteristiche più importanti del progetto: strutture dati, implementazioni, algoritmi, puntatori.

2.2 Classi

Le classi implementate sono relative alle entità presentate nell'*Introduzione*.
La classe *GestioneGruppo* è stata creata per incapsulare tutte le operazioni possibili, in modo che la chiamata dei metodi nel main risulti semplificata.

Oltre alle classi presenti in Figura 1 è stato necessario creare una `struct` per la memorizzazione delle posizioni dei puntatori.
Per l'eliminazione degli oggetti si fa riferimento ad un solo `vector` di gruppi, tramite cui si accede ai veicoli e ai volontari di ciascuno (anch'essi memorizzati tramite `vector`). Per questo motivo è stato essenziale memorizzare la posizione del gruppo a cui accedere e la posizione del veicolo o volontario desiderato.

Tutte le classi sono state definite tramite interfaccia (file *.h*), mentre l'implementazione dei vari metodi, tranne quelli definiti `inline`, è contenuta nei rispettivi source file (file *.c*).

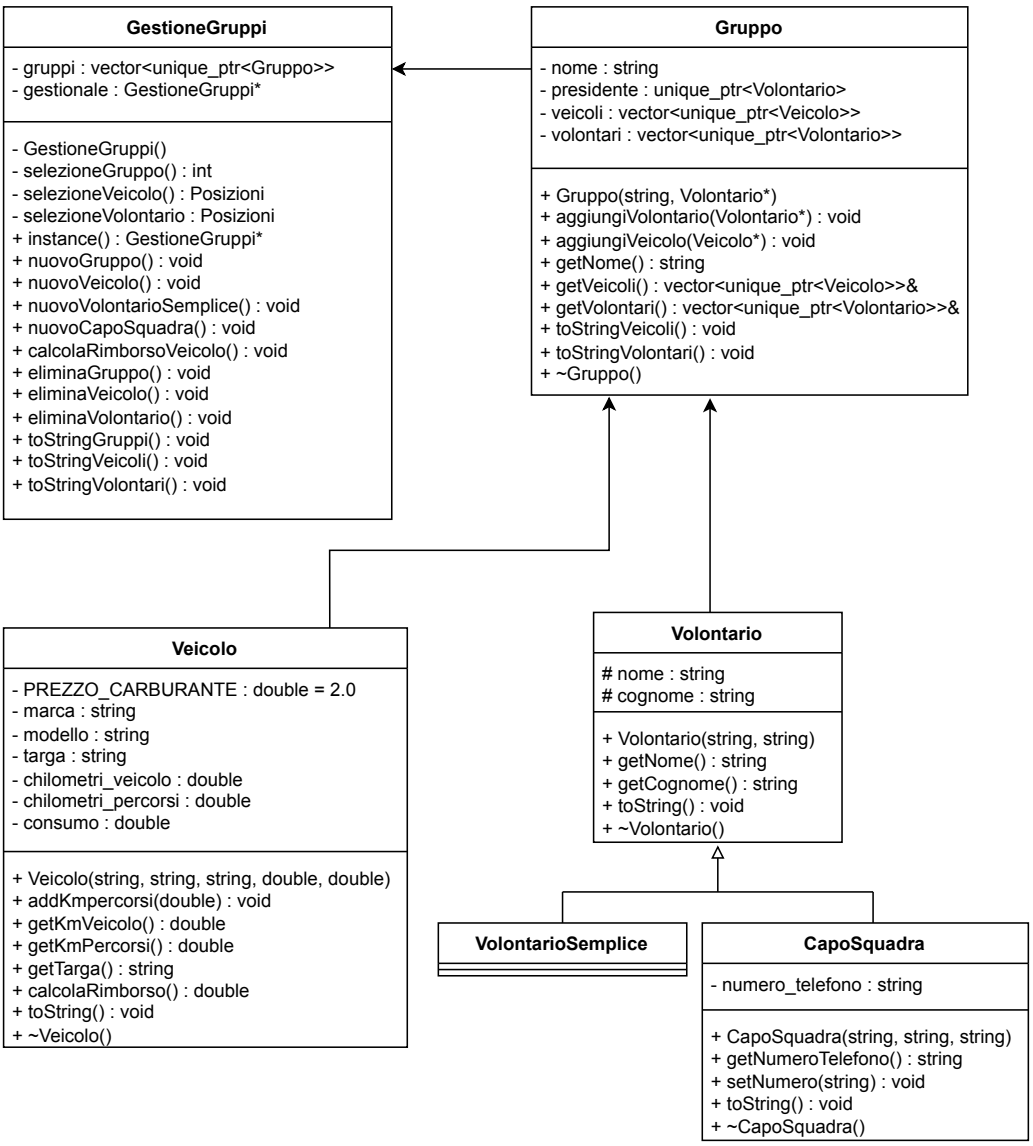

Figura 1: Diagramma delle classi

2.3 Distruttori

Ogni oggetto è dotato di un costruttore e un distruttore, i quali contengono semplicemente una stampa di uscita.
In particolare, per la classe Volontario il distruttore è stato dichiarato `virtual` in modo che le classi derivate invochino in ordine corretto anche il distruttore della classe base.

In un campo è possibile ricorrere all'utilizzo di un distruttore quando, ad esempio, un Volontario deve rientrare a casa oppure anche se l'intero gruppo è costretto al rientro.

```
virtual ~Volontario(){
 cout << "Cancellazione Volontario in corso...";
}
```

```
virtual ~VolontarioSemplice(){
 cout << "Cancellazione Volontario Semplice in corso...";
}
```

```
virtual ~CapoSquadra(){
 cout << "Cancellazione Capo Squadra in corso...";
}
```

Questo ordine è evidente proprio nel momento in cui accade una delle casistiche sopra elencate.
Supponiamo di avere un gruppo così composto:

Tipo	Nome	Cognome
Volontario	Johnny	Bravo
CapoSquadra	Thomas	Magnum
VolontarioSemplice	Theodore	Calvin

Ipotizzando di eliminare tale gruppo, l'output fornito dal programma è il seguente:

```
Cancellazione Volontario in corso...

Cancellazione Capo Squadra in corso...
Cancellazione Volontario in corso...

Cancellazione Volontario Semplice in corso...
Cancellazione Volontario in corso...
```

2.4 Funzioni inline

L'istruzione `inline` permette di implementare i metodi direttamente nell'interfaccia della classe, ma solo nel caso in cui il codice sia di poche righe.
Questa modalità è stata usata soprattutto nella classe *Veicolo* per la definizione dei metodi `get`:

```
inline double getKmVeicolo()  {return chilometri_veicolo;}
inline double getKmPercorsi() {return chilometri_percorsi;}
inline string getTarga()      {return targa;}
```

2.5 Ereditarietà

La classe base utilizzata per mostrare l'ereditarietà è la classe *Volontario*. La classi derivate in modo pubblico sono invece *VolontarioSemplice* e *CapoSquadra*.
La classe *CapoSquadra*, oltre ad ereditare metodi e campi della classe *Volontario*, aggiunge un terzo campo e i metodi annessi.
Entrambe le classi derivate fanno override del metodo `toString`.

2.6 STL

2.6.1 Contenitori

Per poter memorizzare tutti gli oggetti è stato utilizzato il contenitore `vector`, sia nella classe *Gruppo* per allocare tutti i volontari e tutti i veicoli, ma soprattutto nella classe *GestioneGruppi*.
In alternativa sarebbe stato più semplice utilizzare il contenitore `list` poiché fornisce un apposito metodo di cancellazione dei suoi elementi in base al valore, a differenza di `vector` che necessita di un iteratore.
La scelta è ricaduta comunque su `vector`: permette di accedere agli elementi in posizione random, sia tramite l'utilizzo delle parentesi quadre [ ] sia tramite la funzione `at()`.

```
vector<unique_ptr<Gruppo>> gruppi;
...
void nuovoVeicolo(){
 ...
 int gruppo = selezioneGruppo();
 Veicolo *ve = new Veicolo(ma, mo, ta, km, con);
 gruppi.at(gruppo)->aggiungiVeicolo(ve);
}
```

Nell'esempio viene inserito un nuovo veicolo.
In particolare, `aggiungiVeicolo` è un metodo fornito dalla classe *Gruppo* per inserire un veicolo nel proprio contenitore STL.
La variabile `gruppo` è il risultato di una funzione privata che permette all'utente di selezionare il gruppo su cui si vogliono eseguire le operazioni.

2.6.2 Iteratori

Gli iteratori sono stati utilizzati per accedere agli elementi presenti nei vari contenitori `vector`. Ad esempio per poter cancellare un determinato elemento:

```
void eliminaGruppo(){
 gruppi.erase(gruppi.begin() + selezioneGruppo());
}
```

Oppure per accedere ai metodi di un determinato elemento:

```
void Gruppo::toStringVeicoli(){
 for(auto &v : veicoli)
  v->toString();
}
```

2.6.3 Algoritmi

I metodi `toString` presenti nella classe *GestioneGruppi* sfruttano l'algoritmo STL `for_each()` che sfrutta comunque gli iteratori:

```
void GestioneGruppi::toStringVeicoli(){
 for_each(gruppi.begin(),
          gruppi.end(),
          [] (unique_ptr<Gruppo> &gr){
  gr->toStringVeicoli();
 });
}
```

2.7 Smart pointers

I puntatori intelligenti sono stati utilizzati per la gestione degli oggetti all'interno del contenitore `vector`. In questo modo l'allocazione e la deallocazione, tramite riferimenti, viene gestita più efficientemente rispetto all'eventuale utilizzo dei soli *raw pointers*.
In particolare si fa riferimento agli `unique pointers` che non possono essere copiati, appunto perché unici. Per questo motivo si ricorre all'istruzione `move()`.

```
string nome;
unique_ptr<Volontario> presidente;
vector<unique_ptr<Volontario>> volontari;
...
Gruppo::Gruppo(string n, Volontario *pres){
 nome = n;
 unique_ptr<Volontario> pt_pres (pres);
 presidente = move(pt_pres);

 volontari.push_back(move(presidente));
}
```

3 Progetto in Java

3.1 Introduzione

Il progetto riservato alla programmazione in Java si occupa della effettiva gestione di un campo di protezione civile durante un'emergenza.
Si suppone che tutti i dati inerenti a volontari, veicoli e gruppi siano già stati inseriti precendentemente, per questo motivo è stata implementata una funzione *popolaCampo()* che si occupa proprio di questo.
Attraverso questo portale è possibile eseguire le seguenti operazioni:

- **stampa** di tutti i gruppi, volontari e veicoli registrati al campo attraverso il classico metodo `toString();`
- gestione delle **attività** che i volontari devono svolgere durante l'emergenza;
- gestione dei **rimborsi** dovuti ai volontari e per i veicoli.

Le classi implementate sono le medesime del progetto sviluppato in C++, è stata aggiunta una classe relativa alle Attività e le interfacce per sfruttare il *Visitor Pattern*.
Di seguito viene riportato il diagramma delle classi (Figura 2).

Nella classe riservata al test del portale, oltre al *main*, sono stati implementati i seguenti metodi:

- *popolaCampo()* per poter avere dei dati su cui testare il software;
- *gestisciAttivita()* per permettere all'utente di visualizzare la disponibilità di mezzi e volontari, creare una nuova attività con i relativi volontari oppure dichiarare la chiusura di un'attività terminata;
- *gestisciRimborsi()* per permettere all'utente di calcolare il rimborso relativo ad un veicolo o ad un volontario.

Tutti questi metodi fanno riferimento alla classe *Campo* che viene resa disponibile grazie all'utilizzo del *Singleton pattern*: viene creata una sola istanza di Campo alla quale si può accedere solamente tramite il metodo `instance()`, chiamato in ciascuno dei metodi appena citati (vedi Listing 1).

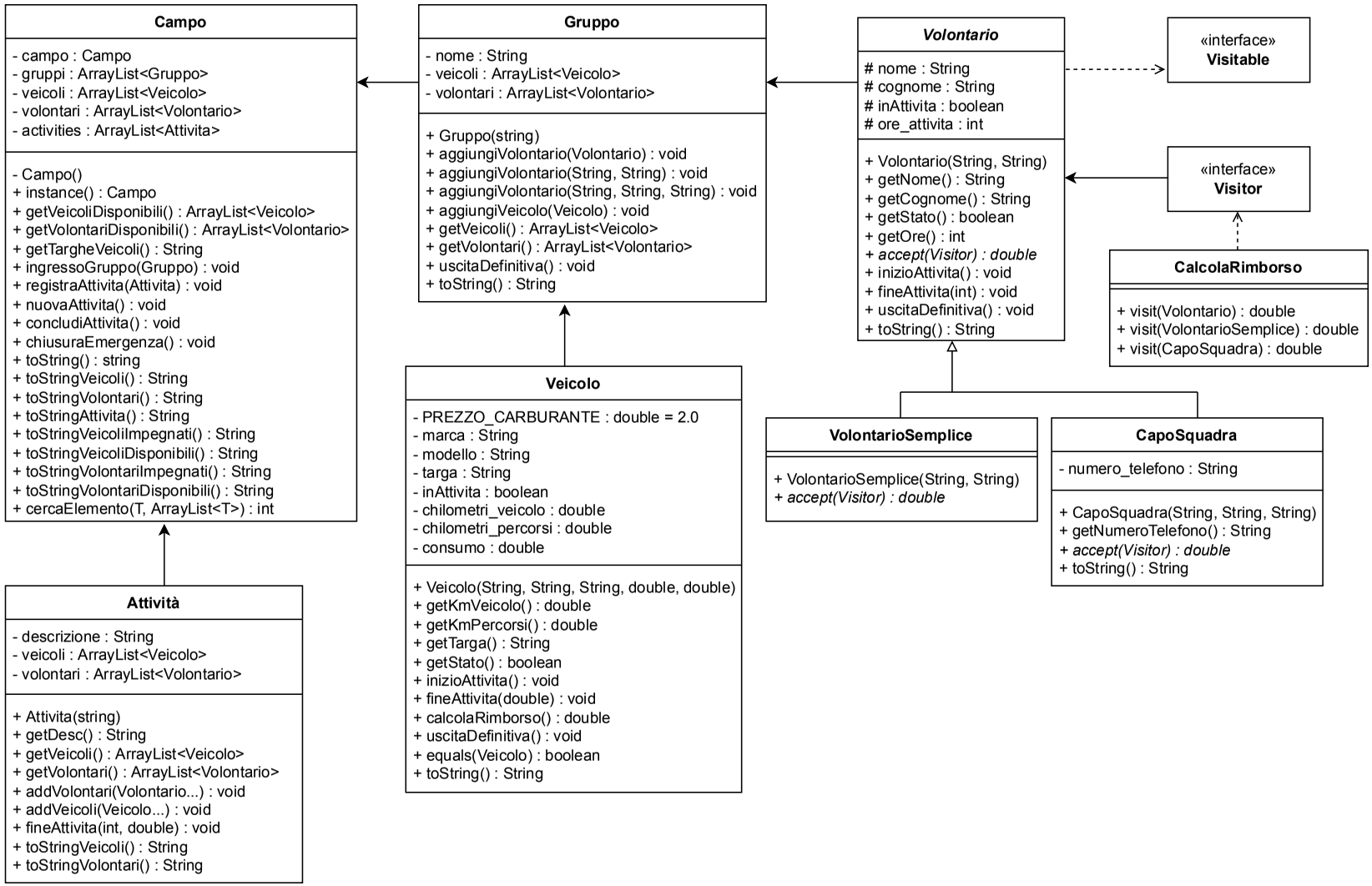

Figura 2: Diagramma delle classi

Listing 1: Utilizzo del pattern Singleton

```
public class Campo{
 private static Campo campo = null;
 ...
 private Campo() { }
 public static Campo instance(){
  if (campo == null){
   //inizializzazione delle variabili
  }
  return campo;
 }
}

public class GestioneCampo{
 public static void main(String[] args){
  Campo campo = Campo.instance();
  ...
 }

 public static popolaCampo(){
  Campo campo = Campo.instance();
  //operazioni per la popolazione
 }
}
```

3.2 Il metodo popolaCampo()

Come già anticipato, all'interno di questo metodo vengono richiamate tutte quelle funzioni rese disponibili dalle classi *Gruppo*, *Veicolo*, *VolontarioSemplice* e *CapoSquadra* per la creazione dei rispettivi oggetti. Viene infine aggiunto il gruppo creato al campo.
Sempre tramite questo metodo sono state aggiunte alcune attività.

```
public static void popolaCampo(){
 Campo campo = Campo.instance();

 Gruppo g = new Gruppo("Serpeverde");

 Volontario v1 = new CapoSquadra("Riga", "Fabio", "3334444444");
 Volontario v2 = new VolontarioSemplice("Rossi", "Mario");
 g.aggiungiVolontario(v1);
 g.aggiungiVolontario(v2);
 //sfrutto anche l'overload del metodo aggiungiVolontario()
 g.aggiungiVolontario("Verdi", "Silvana");
 g.aggiungiVolontario("Fiori", "Viola", "3333333333");
 g.aggiungiVeicolo(new Veicolo("Ford","Ranger","GX077SA",10000,3));

 campo.ingressoGruppo(g);

 Attivita a = new Attivita("Taglio piante fiume");
 a.addVolontari(v1, v2);
}
```

3.2.1 Utilizzo dei Varargs

In particolare, i metodi presenti nella classe *Attivita* utili all'assegnamento di mezzi e volontari sfruttano i Varargs:

```
public void addVolontari(Volontario...v){
 for (int i = 0; i < v.length; i++){
  //Ogni volontario deve essere segnato come "impegnato"
  v[i].inizioAttivita();
  volontari.add(v[i]);
 }
}
```

3.3 Il metodo gestisciAttivita()

All'interno di questo metodo viene presentato all'utente un menù attraverso il quale può decidere se

- visualizzare quali volontari sono impegnati in attività;
- visualizzare quali volontari sono disponibili, cioè sono presenti al campo;
- visualizzare quali mezzi sono utilizzati in attività;
- visualizzare quali mezzi sono disponibili;
- inserire una nuova attività;
- dichiarare la conclusione di un'attività.

3.3.1 Inserimento di un'attività e il metodo generico

Per inserire una nuova attività viene richiesta all'utente una breve descrizione di essa.

Successivamente viene richiesto l'inserimento di un capo squadra per poi passare all'inserimento di volontari semplici e di mezzi. La scelta dei volontari viene effettuata permettendo all'utente di cercare il Volontario tramite i propri dati (cognome e nome), ma solo tra quelli che sono disponibili al campo.
Per consentire ciò è stato implementato un metodo generico utilizzato sia per la ricerca dei volontari, sia per la ricerca dei mezzi, basato sulla costruzione dei metodi `toString()` presenti nel progetto.

```
public <T> int cercaElemento(T el, ArrayList<T> elenco){
 int pos = - 1;
 for (int i = 0; i < elenco.size(); i++){
  if(elenco.get(i).toString().contains(el.toString())){
   pos = i;
   break;
  }
 }
 return pos;
}
```

E, ad esempio, può essere invocato nel seguente modo:

```
//cognome, nome e desc inserite dall'utente
Attivita a = new Attivita(desc)

Volontario v = new VolontarioSemplice(cognome, nome);
int pos = cercaElemento(v, getVolontariDisponibili());

if (pos == -1)
 syserr("Volontario non disponibile");
else
 a.addVolontari(getVolontariDisponibili().get(pos));
```

3.3.2 Conclusione di un'attività

Per far sì che l'utente possa scegliere l'attività da segnalare come terminata, viene semplicemente visualizzato un elenco numerato delle attività che sono ancora in corso. In questo modo, selezionando il numero corrispondente a quella terminata, l'utente può segnalarne la conclusione.

Prima che un'attività venga effettivamente cancellata è necessario che vengano registrate le ore di servizio che i volontari hanno prestato e il chilometraggio che i mezzi hanno riportato durante tale attività, andando ad incrementare le variabili interessate per il rimborso.

Infine, i volontari e i mezzi che erano assegnati all'attività vengono resi nuovamente disponibili e pronti per essere nuovamente selezionati; l'attività viene rimossa dalla lista di quelle attive nell'emergenza.

3.4 Il metodo gestisciRimborsi()

Il calcolo dei rimborso spese viene effettuato sia per i volontari sia per i mezzi.
Nello specifico, il rimborso per i Volontari Semplici considera solamente le ore di servizio effettuate, mentre per i Capo Squadra, esiste un supplemento fisso.
Il rimborso per i mezzi invece considera il prezzo del carburante, il consumo del mezzo e i chilometri percorsi durante le attività.

3.4.1 Visitor Pattern

Per semplificare il calcolo del rimborso sui Volontari è stato implementato il *Visitor Pattern*: viene eseguito la stessa operazione ma su oggetti che sono di tipi diversi e, in questo caso, il calcolo effettuato è altrettanto differente.

Il pattern utilizza due **interfacce**: l'interfaccia *Visitable* (implementata dalla classe *Volontario*) e l'interfaccia *Visitor* che consente l'esecuzione dell'operazione desiderata.

È importante che la classe astratta *Volontario* abbia un metodo (astratto) `accept(Visitor v)` e che le sottoclassi *VolontarioSemplice* e *CapoSquadra* lo implementino, per fare in modo che venga calcolata l'operazione sul tipo corretto.

```
public abstract class Volontario implements Visitable{
 ...
 public abstract double accept(Visitor v);
}

public class VolontarioSemplice extends Volontario{
 ...
 public double accept(Visitor v){
  return v.visit(this);
 }
}
```

Di seguito viene riportato il codice della classe che implementa l'interfaccia *Visitor*, ovvero la classe che identifica l'operazione di calcolo rimborso.

```
public class CalcolaRimborso implements Visitor{
 public double visit(VolontarioSemplice vs){
  return 5.0 * vs.getOre();
 }

 public double visit(CapoSquadra vcs){
  return 5.0 * vcs.getOre() + 15;
 }
}
```

4 Haskell

In questo capitolo verranno esposte alcune semplici funzionalità della programmazione in Haskell.

Haskell viene utilizzato per la programmazione funzionale, è basato sulle funzioni pure che possono essere viste come delle definizioni. Infatti la questo tipo di programmazione è di tipo dichiarativo e non imperativo (come ad esempio lo sono C++ o Java).

Bisogna inoltre tenere presente che in Haskell i dati sono immutabili, ovvero non possono essere modificati in esecuzione, per questo motivo vengono creati nuovi dati se necessario.

Per questa parte viene sempre utilizzato come esempio il Volontario, definito tramite *Record* in questo modo:

```
data Volontario =
  VS{nome::String,cognome::String,eta::Int}
 |CS{nome::String,cognome::String,num::String,eta::Int}

instance Eq Volontario where
 (==) (VS n1 c1 e1) (VS n2 c2 e2) = (n1==n2)&&(c1==c2)&&(e1==e2)
 (==) (CS n1 c1 t1 e1) (CS n2 c2 t2 e2) = (t1==t2)
 (==) (VS n1 c1 e1) (CS n2 c2 t2 e2) = False
 (==) (CS n2 c2 t2 e2) (VS n1 c1 e1) = False
```

Il vantaggio dell'uso dei record è la "creazione" automatica di metodi che possiamo definire `get()`, come ad esempio `nome vol` restituisce il nome del Volontario `vol`.

La parte inferiore del codice è riservata alla creazione dell'istanza della TypeClass Eq, questo permette di personalizzare le funzioni che questa classe comprende.
Oltre all'uguaglianza, la classe Eq dispone anche della funzione di disuguaglianza che però Volontario non definisce: è possibile derivarla dalle uguaglianze.

Nel codice sono stati creati alcuni volontari e poi inseriti in una lista in modo da semplificare il test delle funzioni implmentate. Di seguito si riporta un esempio di definizione:

```
v1 = VS "Giorgia" "Bressanelli" 25
--altri volontari
v4 = CS "Pietro" "Bressanelli" "5555" 55


--creazione della lista
volontari = [v1,v2,v3,v4]
```

Per definire le funzioni in Haskell la sintassi è la seguente:
`name :: arg1 -> arg2 -> ... -> argn -> returntype`
Sono state definite le funzioni di stampa che sfruttano la funzione `map` per visualizzare i dati in maiuscolo:

```
visualizzaMaiuscolo :: [Char] -> [Char]
visualizzaMaiuscolo dato = map toUpper dato

toStringVolontario :: Volontario -> [Char]
toStringVolontario vol = "- "
 ++ visualizzaMaiuscolo (nome vol)
 ++ " "
 ++ visualizzaMaiuscolo (cognome vol)

toStringTutti :: [Volontario] -> [Char]
toStringTutti [] = []
toStringTutti (x:xs) = toStringVolontario x
                       ++ " - "
                       ++ toStringTutti xs
```

Le funzioni in Haskell spesso sono definite in modo ricorsivo, come si può notare nella definizione della funzione che stampa i dati di tutti i volontari.

Infine, è stata definita una funzione per visualizzare tutti i volontari che hanno un'età maggiore di un certo numero ricevuto come parametro:

```
--controllo sul singolo
olderThan :: Volontario -> Int -> [Char]
olderThan vol 0 = toStringVolontario vol
olderThan vol e
 | (eta vol) > e = toStringVolontario vol
 | otherwise     = []

--controllo su tutti
allOlderThan :: [Volontario] -> Int -> [Char]
allOlderThan [] _ = []
allOlderThan xs 0 = toStringTutti xs
allOlderThan (x:xs) et = (olderThan x et)
                         ++ " - "
                         ++ (allOlderThan xs et)
```

Di seguito un esempio di main per poter testare le funzioni definite:

```
main = do
 -- Stampa tutti i volontari nella lista
 print(toStringTutti volontari)
 -- Stampa solamente i dati relativi al volontario 2
 print(toStringVolontario v2)

 print(v1==v2) -- False
 print(v4==v4) -- True

 -- Stampa i dati di tutti i volontari che hanno piu di 40 anni
 print(allOlderThan volontari 40)

 -- Inverte la lista e stampa, stampa di seguito il numero di
 -- elementi presenti nella lista
 print(toStringTutti (reverse volontari))
 print(length volontari)
```