



**Università degli Studi di Bergamo**

---

SCUOLA DI INGEGNERIA

Corso di Laurea Magistrale in Ingegneria Informatica

## **Progetto corso di Programmazione avanzata**

C++ e Haskell

Prof.  
**Angelo Gargantini**

Candidato  
**Matteo Verzeroli**  
Matricola 1057926

# Indice

<b>1</b>	<b>Progetto C++</b>	<b>2</b>
1.1	Introduzione . . . . .	2
1.2	Funzionamento . . . . .	2
1.2.1	Inserimento nuovo utente . . . . .	3
1.2.2	Cancella utente . . . . .	5
1.2.3	Visualizza utenti . . . . .	5
1.2.4	Visualizza interventi . . . . .	5
1.2.5	Lougout . . . . .	6
1.3	Diagramma delle classi . . . . .	8
1.4	Dettagli implementazione . . . . .	9
1.4.1	Ereditarietà multipla . . . . .	9
1.4.2	Distruttore virtual . . . . .	9
1.4.3	Metodo pure virtual . . . . .	9
1.4.4	Metodi virtual: overriding . . . . .	10
1.4.5	Overloading di metodi . . . . .	10
1.4.6	Pattern Singleton . . . . .	10
1.4.7	STL: std::vector, std::map, std::iterator . . . . .	10
1.4.8	Smart pointers: shared_ptr . . . . .	11
1.4.9	Template . . . . .	11
1.4.10	Enum . . . . .	12
1.4.11	Friend . . . . .	12
<b>2</b>	<b>Progetto Haskell</b>	<b>13</b>
2.1	Descrizione . . . . .	13
2.2	Algoritmo: Crypto Square . . . . .	13
2.2.1	Esempio . . . . .	14
2.3	Algoritmo: Caesar Cipher . . . . .	15
2.3.1	Esempio . . . . .	15
2.4	Main . . . . .	16

# Capitolo 1

## Progetto C++

### 1.1 Introduzione

Il programma realizzato permette di gestire utenti e interventi di una squadra della Protezione Civile, utilizzando una semplice interfaccia grafica realizzata tramite il framework Qt.

Gli utenti che possono utilizzare l'applicazione sono suddivisi in base al loro ruolo:

- volontario;
- caposquadra;
- amministratore.

Ogni utente, in base al proprio ruolo, avrà accesso a particolari funzionalità quali aggiungere o cancellare un utente nel sistema, visualizzare gli utenti della propria squadra, visualizzare e pianificare interventi. A differenza degli amministratori, il volontario e il caposquadra appartengono a una squadra, caratterizzata da un nome e da un'area di intervento.

Per accedere all'applicazione è necessario effettuare il login, inserendo lo username, rappresentato dal proprio numero di matricola, e la password scelta al momento della registrazione dell'utente.

L'interfaccia grafica è stata realizzata attraverso gli oggetti QtWidget che permettono di semplificare la realizzazione della GUI tramite l'editor integrato nell'IDE Qt Creator.

### 1.2 Funzionamento

All'avvio del programma, viene subito richiesto di effettuare il login (Fig.1.1), inserendo lo username (costituito dalla matricola assegnata dal sistema al momento dell'inserimento dell'utente) e la password. In caso di username e/o password errati, viene mostrato un messaggio di errore e lo username e la password errati vengono cancellati. Non è quindi possibile procedere oltre nell'applicazione fino a quando si inseriscono delle credenziali corrette. Al contrario, se le credenziali di accesso sono corrette viene mostrato all'utente la schermata principale (Fig.1.2).

La schermata principale è divisa in due sezioni. Nella parte sinistra è possibile vedere tutte le informazioni memorizzate dell'utente (fotografia, matricola, nome, cognome, data di nascita, email, numero di telefono, sesso) e le informazioni relative alla squadra (identificativo e nome della squadra, nome e coordinate dell'area di intervento). Le informazioni relative alla squadra non sono mostrate nel caso in cui l'utente sia un amministratore, in quanto non appartiene a nessuna squadra.

Nella parte destra è possibile vedere un piccolo menù dal quale scegliere le operazioni da eseguire: aggiungere un utente, eliminare un utente, visualizzare utenti e interventi. A seconda del ruolo dell'utente loggato, questo menù mostrerà diverse opzioni. In particolare, al volontario è permesso

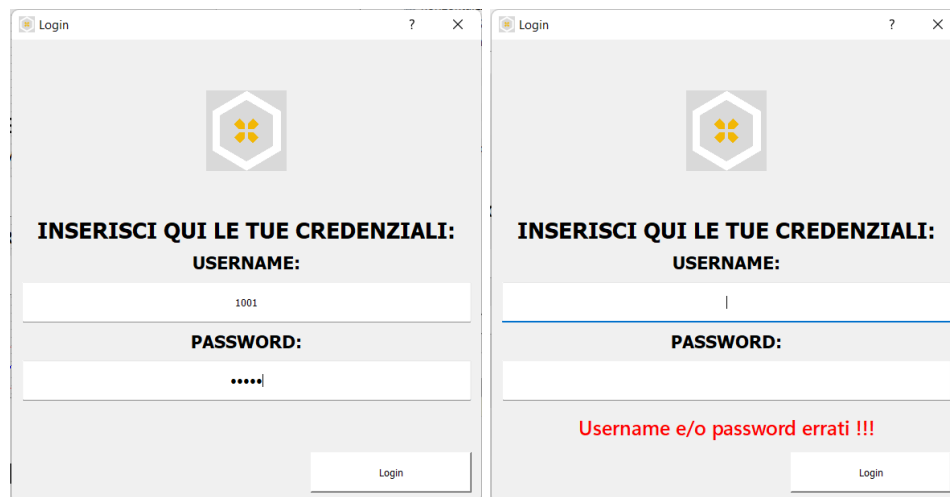


Figura 1.1: Finestra di login e login errato.

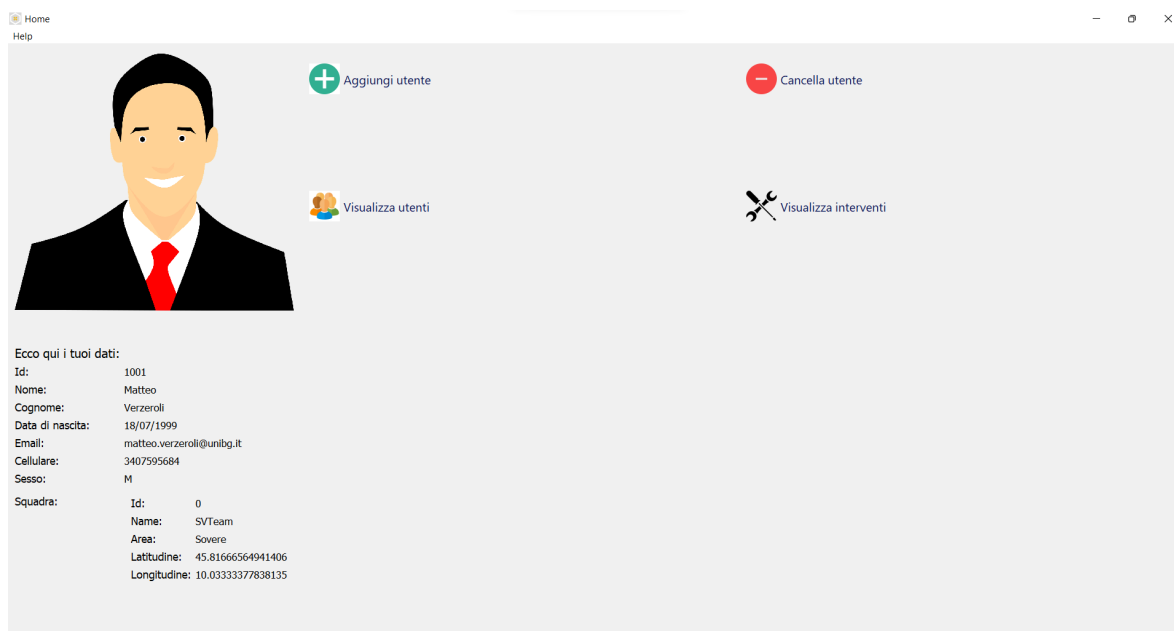


Figura 1.2: Finestra principale di un caposquadra.

solamente visualizzare gli interventi, mentre all'amministratore non è consentito di visualizzare gli interventi. Cliccando un pulsante nel menù sarà possibile eseguire le operazioni richieste.

### 1.2.1 Inserimento nuovo utente

Cliccando su *Aggiungi utente* viene visualizzata una form da compilare per inserire i dati dell'utente da registrare nel sistema, come mostrato nella Fig.1.3.

Si noti in particolare che bisogna anche indicare il ruolo del nuovo utente. Se viene cliccato il ruolo *Volontario* oppure *Caposquadra*, viene mostrata anche una *combo box* nella quale è possibile scegliere una delle squadre disponibili in cui inserire l'utente.

Home Help

Aggiungi utente Cancella utente

Visualizza utenti Visualizza interventi

Ecco qui i tuoi dati:

Id: 1001  
 Nome: Matteo  
 Cognome: Verzeroli  
 Data di nascita: 18/07/1999  
 Email: matteo.verzeroli@unibg.it  
 Cellulare: 3407595684  
 Sesso: M  
 Squadra: Id: 0, Name: SVTeam, Area: Sovero, Latitudine: 45.8166564941406, Longitudine: 10.03333377838135

Nome: \_\_\_\_\_  
 Cognome: \_\_\_\_\_  
 Password: \_\_\_\_\_  
 Data di nascita: 01/01/2000  
 Email: \_\_\_\_\_  
 Cellulare: \_\_\_\_\_  
 Path immagine: C:/Users/Matteo Verzeroli/Desktop/PA Project/Progetto\_C++/PCTeams/img/avatar.png  
 Sesso: ☒ Maschio ☐ Femmina  
 Ruolo: ☐ Amministratore ☒ Volontario ☐ Caposquadra  
 Squadra: 0 SVTeam

Torna alla home

Aggiungi utente

Figura 1.3: Inserimento nuovo utente.

Nel caso in cui si voglia inserire un volontario, le squadre mostrate vengono così selezionate:

- se l'utente attualmente loggato è un amministratore, vengono mostrate tutte le squadre inserite nel sistema;
- se l'utente attualmente loggato è un caposquadra, viene mostrata solamente la propria squadra.

Invece, se si vuole inserire un caposquadra vengono mostrate tutte le squadre inserite nel sistema che non hanno ancora un caposquadra. Premendo il pulsante *Inserisci utente*, l'utente viene inserito nel sistema e viene mostrato un messaggio di conferma nella *status bar* in fondo alla finestra. Se non vengono compilati tutti i campi, il sistema segnala finestra di errore (Fig.1.4).

Visualizza utenti Visualizza interventi

Errore

Alcuni campi vuoti!

OK

Nome: \_\_\_\_\_  
 Cognome: \_\_\_\_\_  
 Password: \_\_\_\_\_  
 Data di nascita: 01/01/2000  
 Email: \_\_\_\_\_  
 Cellulare: \_\_\_\_\_

Figura 1.4: Errore inserimento nuovo utente.

## 1.2.2 Cancella utente

Nel caso in cui si voglia cancellare un utente, è possibile selezionare una squadra (tramite una *combo box*) e indicare l'utente da eliminare, selezionandolo nella lista sottostante (Fig.1.5). Un caposquadra può solamente cancellare utenti della propria squadra. Un amministratore invece può eliminare utenti da tutte le squadre, compresi gli amministratori. Nessuno può eliminare se stesso: in tal caso viene mostrato un messaggio di errore.

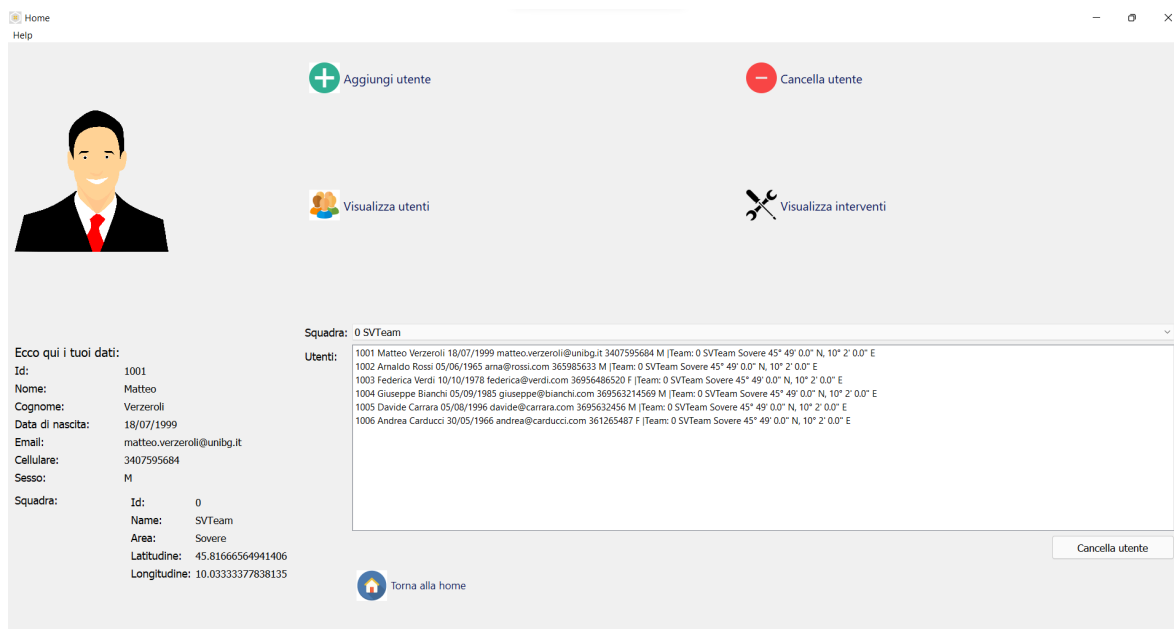


Figura 1.5: Cancellazione utente.

## 1.2.3 Visualizza utenti

Un caposquadra o un amministratore possono visualizzare gli utenti appartenenti a una squadra (Fig.1.6), selezionandola dalla *combo box*. È possibile visualizzare anche gli amministratori. In questa funzionalità, se l'utente loggato è un amministratore potrà visualizzare la lista degli utenti appartenenti a tutte le squadre inserite nel sistema. Al contrario, se è un caposquadra potrà vedere solamente gli amministratori o la propria squadra.

## 1.2.4 Visualizza interventi

I volontari possono visualizzare gli interventi della propria squadra, mentre i capisquadra quelli di tutte le squadre. Cliccando su *Visualizza interventi*, viene mostrato un calendario sul quale selezionare una data (Fig.1.7). Vengono così mostrati gli interventi programmati in quella giornata.

Gli eventi possono essere creati solo dal caposquadra, cliccando sul pulsante *Inserisci un nuovo intervento*. Quando il pulsante viene cliccato, viene mostrata una form per l'inserimento delle informazioni dell'intervento, come mostrato in figura 1.8. Cliccando su *Inserisci intervento* un messaggio di conferma indica che l'evento è stato inserito. Se tutti i campi non sono stati completati, viene invece mostrato un messaggio di errore.

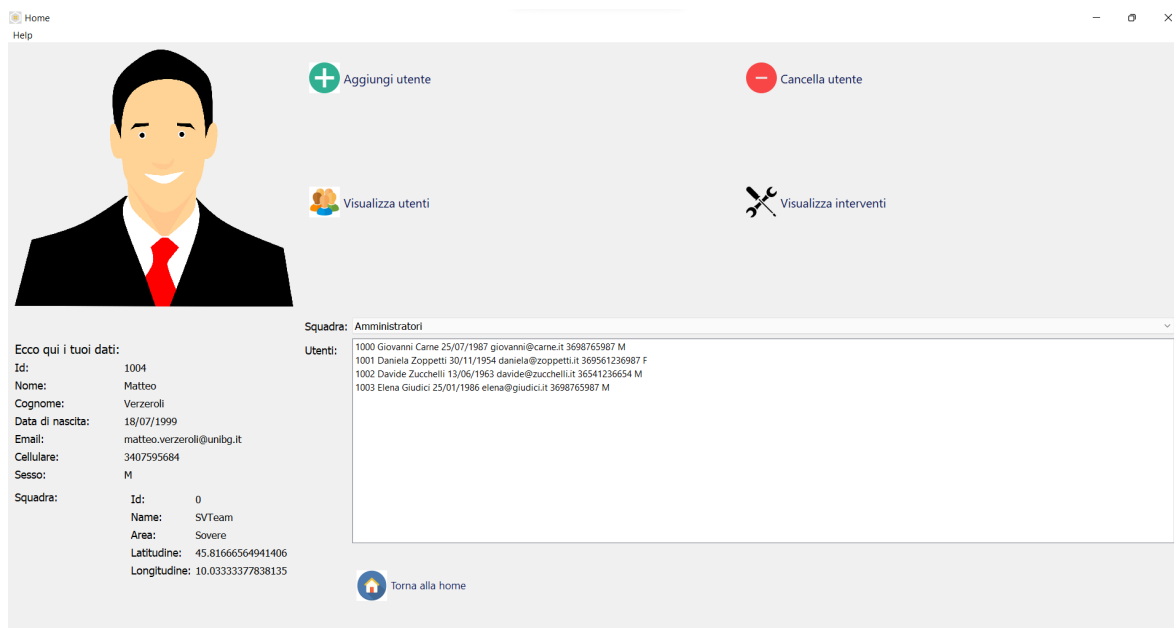


Figura 1.6: Visualizzazione utenti.

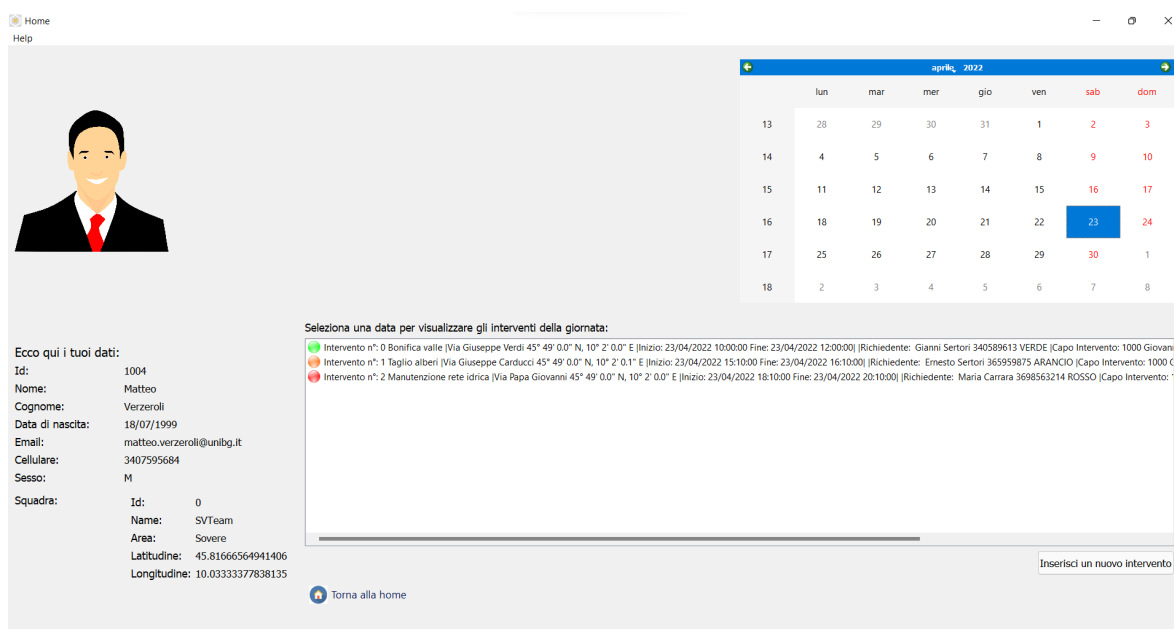



Figura 1.7: Visualizzazione interventi.

## 1.2.5 Logout

Cliccando nel menù in alto alla schermata *Help*, è possibile effettuare il logout. Viene quindi mostrata la finestra di login.

Home

Help



Ecco qui i tuoi dati:

Id: 1004

Nome: Matteo

Cognome: Verzeroli

Data di nascita: 18/07/1999

Email: matteo.verzeroli@unibg.it

Cellulare: 3407595684

Sesso: M

Squadra:

Id: 0

Nome: SVTeam

Area: Sovere

Latitudine: 45.81666564941406

Longitudine: 10.03333377838135

Nome:

Indirizzo:

Latitudine:

Longitudine:

Referente:

Cellulare:

Criticità: ROSSO

Squadra: 0 SVTeam

Responsabile: 1004 Matteo Verzeroli

Data inizio: 23/04/2022 00:00

Data fine: 23/04/2022 00:00

[Torna alla home](#)

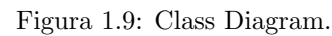
april, 2022

	lun	mar	mer	gio	ven	sab	dom
13	28	29	30	31	1	2	3
14	4	5	6	7	8	9	10
15	11	12	13	14	15	16	17
16	18	19	20	21	22	23	24
17	25	26	27	28	29	30	1
18	2	3	4	5	6	7	8

Inserisci intervento

Figura 1.8: Inserimento nuovo intervento.



$\infty$ 

## 1.4 Dettagli implementazione

L'*entry point* dell'applicazione sviluppata è la funzione *main* definita nel file `main.cpp`. All'interno del `main` vengono create le istanze dell'applicazione e della finestra della *Home* e del *Login*. Inoltre, vengono creati alcuni utenti, squadre e interventi per poterli utilizzare come esempio. Infine, viene lanciata l'applicazione. Le finestre sono realizzate tramite i *QWidgets* offerti da Qt e sono descritte nei file `.ui`. Ogni finestra (`MainWindow` che rappresenta la home e `LoginForm` che rappresenta la pagina di login) è descritta da una classe, implementata nei file `.h` (header) e `.cpp`. All'interno di queste classi, sono definiti anche gli *slots*, tipici del linguaggio Qt, che rappresentano degli *event handler* degli eventi sollevati dall'interfaccia grafica (o da altre classi). In questo caso, entrambe le classi derivano dalla classe *QObject*, che integra alcune funzioni base utili per lo sviluppo in Qt.

### 1.4.1 Ereditarietà multipla

Gli utenti sono rappresentati tramite una gerarchia di classi, che ne permette la differenziazione in base al ruolo. In particolare, la classe base è rappresentata dalla classe `User`, astratta in quanto contiene un metodo *pure virtual*. Da questa derivano, in modo virtuale, la classe `Volunteer` e `Administrator`. Infine, la classe `Foreman`, che rappresenta il caposquadra, deriva sia da `Volunteer` che da `Administrator`. Si va quindi a costituire una struttura a *diamante*, che porterebbe a problemi negli oggetti di tipo `Foreman`, che conterebbero due differenti istanze di `User`: una ereditata da `Volunteer` e una da `Administrator`. Il *problema del diamante* si risolve utilizzando l'ereditarietà virtuale della classe `User` nelle classi `Volunteer` e `Administrator`.

```
1 class User { /*...*/ }
2
3 class Administrator : virtual public User { /*...*/ }
4
5 class Volunteer : virtual public User { /*...*/ }
6
7 class Foreman : public Administrator, public Volunteer { /*...*/ }
```

### 1.4.2 Distruttore virtual

Tutti i distruttori delle classi che ammettono sottoclassi sono stati dichiarati *virtual*, in modo tale che la distruzione di un oggetto puntato da un riferimento della superclasse richiami in modo corretto i distruttori delle sottoclassi.

```
1 virtual ~User() { /*...*/ }
2
3 virtual ~Administrator() { /*...*/ }
4
5 virtual ~Volunteer() { /*...*/ }
6
7 virtual ~Foreman() { /*...*/ }
```

### 1.4.3 Metodo pure virtual

La classe `User` è definita astratta a causa della presenza di un metodo *pure virtual*. Per cui, non è possibile creare una istanza del tipo `User`. Questo metodo, deve essere quindi ridefinito in ogni sottoclasse.

```
1 virtual std::shared_ptr<Team> getTeam() const = 0;
```

### 1.4.4 Metodi virtual: overriding

Alcuni metodi sono stati definiti virtual, in modo da poter utilizzare il polimorfismo offerto dal linguaggio tramite l'utilizzo di riferimenti agli oggetti. Ad esempio:

```
1 virtual void initializeMainWindow(Ui::MainWindow* ui);
```

Questo metodo è stato implementato nella classe `User` e in tutte le sottoclassi e si occupa di inizializzare la finestra *Home* in base al ruolo dell'utente autenticato.

### 1.4.5 Overloading di metodi

In alcuni casi, si è eseguito l'overload dei metodi, definendo diverse funzioni con lo stesso nome ma differente segnatura, in modo da differenziarne il comportamento in base ai parametri passati. Ad esempio:

```
1 class OperationRepository{
2     /*...*/
3 public:
4     std::vector<std::shared_ptr<Operation>> getAllOperation();
5     std::vector<std::shared_ptr<Operation>> getAllOperation(const QDate& date);
6     std::vector<std::shared_ptr<Operation>> getAllOperation(int idteam);
7     std::vector<std::shared_ptr<Operation>> getAllOperation(int idteam, const QDate&
8         date);
9 };
```

### 1.4.6 Pattern Singleton

Le classi `UserRepository`, `TeamRepository` e `OperationRepository` rappresentano delle tabelle (costituite da *mappe chiave-valore*) nelle quali vengono salvate le informazioni relative rispettivamente a utenti, squadre e interventi, simulando le operazioni di un *database*. Per garantire l'unicità di ogni *repository*, queste classi sono state definite secondo il pattern *Singleton*, definendo il costruttore privato. In questo modo non è possibile creare istanze della classe al di fuori della stessa. Tuttavia, è possibile accedere all'istanza tramite il metodo `getInstance()`, che restituisce un puntatore all'istanza della classe. In questo modo, si può accedere ai metodi per eseguire operazioni sulla tabella che conserva i dati. Si riporta di seguito la definizione della classe *UserRepository*:

```
1 class UserRepository
2 {
3 public:
4     static UserRepository* getInstance();
5     std::shared_ptr<User> getUserById(int iduser);
6     std::vector<std::shared_ptr<User>> getAllUser();
7     void insertUser(User* user);
8     bool removeUser(int iduser);
9
10    ~UserRepository();
11
12 private:
13     std::map<int, std::shared_ptr<User>> usertable;
14     static UserRepository* instance;
15
16     UserRepository();
17 };
```

### 1.4.7 STL: std::vector, std::map, std::iterator

All'interno dell'applicazione sono stati utilizzati i *containers* `vector` e `map` forniti dalla *Standard Template Library*. In particolare, le classi descritte nella sezione 1.4.6, memorizzano utenti, squadre e

interventi utilizzando una mappa *chiave-valore*, in cui la chiave è di tipo `int` (costituita dall'id dell'oggetto) mentre il valore è uno *smart pointer* all'oggetto. Ad esempio, la mappa che permette la memorizzazione degli utenti è stata dichiarata nel seguente modo:

```
1 std::map<int, std::shared_ptr<User>> usertable
```

Si è scelto di utilizzare una mappa per rendere efficiente le operazioni di ricerca tramite l'id dell'utente. Per permettere il polimorfismo (ossia memorizzare oggetti delle sottoclassi del tipo `User`), evitando il fenomeno dello *slicing*, è stato necessario definire il *valore* della mappa come puntatore di tipo `User`. Inoltre, affinché fosse garantita la distruzione corretta dell'oggetto puntato, si sono utilizzati i puntatori smart di tipo *shared*, descritti nella sezione successiva. In questo modo, quando viene cancellato dalla mappa il riferimento alla risorsa, viene chiamato il distruttore dell'oggetto `User` (se non sono più attivi altri riferimenti).

In alcuni casi, sono stati utilizzati anche oggetti di tipo *vector*, che permettono la definizione di una lista di oggetti. Ad esempio, si riporta la definizione del metodo che restituisce tutti gli utenti inseriti nel sistema:

```
1 std::vector<std::shared_ptr<User>> getAllUser();
```

Per scorrere vettori e mappe sono stati usati gli iteratori, sempre forniti dalla STL:

```
1 /** OperationRepository.cpp */
2
3 std::vector<std::shared_ptr<Operation> > OperationRepository::getAllOperation(int
   idteam)
4 {
5     auto operations = getAllOperation();
6     std::vector<std::shared_ptr<Operation>> result;
7
8     for(auto it = operations.begin(); it != operations.end(); ++it){
9         if((*it)->getTeam()->getIdteam() == idteam){
10             result.push_back(*it);
11         }
12     }
13     return result;
14 }
```

### 1.4.8 Smart pointers: `shared_ptr`

Nel progetto realizzato, si sono spesso utilizzati degli *smart pointers* per semplificare l'utilizzo di puntatori agli oggetti che permettono di evitare problemi quali *dangling pointers* e *memory leaks*, tipici dell'utilizzo non corretto dei puntatori. In particolare, si sono utilizzati gli `shared_ptr<...>`, che permettono proprietari multipli del puntatore raw sottostante. In questo modo, possono essere copiati e passati come parametri ai metodi.

```
1 /** Team.h */
2 std::shared_ptr<Team> team;
```

### 1.4.9 Template

In molte occasioni, si sono utilizzati i *template*, come ad esempio nella definizione del tipo degli *smart pointer*, delle *map* e dei *vector* creati. Nella classe `Helpers` si è anche scritto un metodo che restituisce un vettore di tutti i valori contenuti nella mappa passata come argomento. Affinché questo metodo fosse generico per ogni tipo di mappa chiave-valore, si è scelto di implementarlo utilizzando i *template* nel seguente modo:

```
1 class Helpers{
2 public:
3     template<typename TK, typename TV>
4     static std::vector<TV> extract_values_from_map(std::map<TK, TV> const& input_map){
```

```

5     std::vector<TV> retval;
6     for (auto const& element : input_map) {
7         retval.push_back(element.second);
8     }
9     return retval;
10 }
11 };

```

#### 1.4.10 Enum

All'interno dell'applicazione si è inoltre definito un tipo `enum` per rappresentare la criticità di un intervento, definendolo nel seguente modo:

```

1 enum class COLOR {
2     RED,
3     ORANGE,
4     GREEN
5 };

```

Sono stati poi inseriti dei metodi statici nella classe `Operation` per gestire l'utilizzo della enumerazione.

#### 1.4.11 Friend

A scopo di debug, è stato definito l'operatore `<<` per gli oggetti di tipo `User`. Dal momento che si rendeva necessario accedere ai membri privati della classe, la definizione è stata dichiarata `friend` nella classe `User` nel modo seguente:

```

1 /** User.h */
2 /**...*/
3 friend std::ostream& operator<<(std::ostream&, const User&);

```

## Capitolo 2

# Progetto Haskell

### 2.1 Descrizione

Il progetto scritto in linguaggio Haskell permette di ottenere la versione criptata di una frase secondo l'algoritmo di Crypto Square (<https://exercism.org/tracks/haskell/exercises/crypto-square>) e il Cifrario di Cesare (tratto dall'esercizio 8 capitolo 5 del libro *Programming in Haskell - Graham Hutton*). In particolare, viene richiesto all'utente di inserire una frase ed indicare quale metodo utilizzare per la cifratura. Sulla console viene poi mostrata la stringa criptata.

### 2.2 Algoritmo: Crypto Square

All'interno del programma è stato inizialmente implementato un metodo per la creazione di messaggi segreti chiamato *square code*.

Inizialmente la stringa viene normalizzata: vengono rimossi spazi e punteggiatura, trasformando tutte le lettere in carattere minuscolo. Questa operazione viene effettuata dal seguente codice:

```
1 normalize :: String -> String
2 normalize = delwhitespace . alltolower . filter isAlphaNum
3
4 delwhitespace :: String -> String
5 delwhitespace xs = [x|x<-xs, x /= ' ' ]
6
7 alltolower :: String -> String
8 alltolower [] = []
9 alltolower (x:xs) = toLower x : alltolower xs
```

Ora, la stringa deve essere organizzata in un rettangolo tale che:

- il numero di colonne **c** sia maggiore del numero di righe **r**;
- la differenza **c - r** sia minore o uguale a 1.

Per fare ciò, inizialmente si calcola la radice quadrata del numero di caratteri presenti nella stringa, arrotondata all'intero successivo (funzione *dim*). Successivamente, si divide la stringa in sotto-sequenze di caratteri della lunghezza calcolata (funzione *split*'). Se vengono rispettate le condizioni della definizione del rettangolo, si procede oltre. Altrimenti si divide la stringa riducendo il numero di righe di una unità (funzione *splitcontrol*). Di seguito si riportano le righe di codice:

```
1 dim :: String -> Int
2 dim xs = ceiling . sqrt . fromIntegral $ length xs
3
4 split :: String -> Int -> [String]
5 split [] n = []
```

```

6 split xs n = (take n xs): (split (drop n xs) n)
7
8 split' :: String -> [String]
9 split' xs = split xs (dim xs)
10
11 splitcontrol :: String -> [String]
12 splitcontrol xs = if (length (split' xs) <= (dim xs)) then split' xs else split xs ((
    dim xs)-1)

```

Per far sì che il rettangolo abbia tutte le righe di uguale lunghezza, vengono aggiunti degli spazi bianchi a destra tramite la funzione *addws*:

```

1 padright :: Int -> String -> String
2 padright n s
3   | length s < n = s ++ replicate (n - length s) ' '
4   | otherwise    = s
5
6 addws :: [String] -> [String]
7 addws xs = map (\x->padright (length (head xs)) x) xs

```

Il messaggio cifrato si ottiene leggendo le colonne dall'alto verso il basso, carattere per carattere, da sinistra verso destra (funzione *transpose*).

```

1 encode_cs :: String -> String
2 encode_cs = unwords . transpose . addws . splitcontrol . normalize

```

Infine, a seconda della scelta dell'utente, il messaggio può essere normalizzato per nascondere le dimensioni del rettangolo, che si comportano come "chiave" utile per decifrare il messaggio.

### 2.2.1 Esempio

Definiamo la frase: "If man was meant to stay on the ground, god would have given us roots.". La versione normalizzata sarà: "ifmanwasmeanttostayonthebackgroundgodwouldhavegivenusroots".

Creiamo il rettangolo:

```

"ifmanwas"
"meanttos"
"tayonthe"
"groundgo"
"dwouldha"
"vegivenu"
"sroots  "

```

Il messaggio criptato in forma normalizzata risulta quindi essere: "imtgdvsfearwermayoogoanouuiontnnlvtwttddesaohghnsseoau".

Vediamo che se suddividiamo il messaggio per creare un rettangolo delle stesse dimensioni di quello calcolato prima, otteniamo:

```

"imtgdv"
"sfearw"
"ermayo"
"ogoano"
"uuioin"
"tntnlv"
"twttdd"
"esaohg"
"hn  s"
"seoau "

```

Se leggiamo per colonne, carattere per carattere dall'alto verso il basso, troviamo la frase normalizzata di partenza.

## 2.3 Algoritmo: Caesar Cipher

Il secondo algoritmo proposto è il cifrario di Cesare. In particolare, per codificare una stringa bisogna sostituire ogni lettera con una lettera dell'alfabeto in posizione  $L + K$ , dove  $K$  indica lo shift da effettuare su ogni lettera, mentre  $L$  la posizione nell'alfabeto della lettera corrente. Per decodificare, è sufficiente eseguire la medesima operazione considerando uno shift di  $-K$ . In particolare si riporta la versione in grado di gestire lettere maiuscole, minuscole e numeri.

Per implementare questo algoritmo, sono necessarie delle funzioni che permettono di effettuare lo shift nell'alfabeto. Queste operazioni sono basate sulle funzioni *ord*, che restituiscono il codice ASCII associato al carattere e *chr*, che esegue l'operazione inversa:

```
1 low2int :: Char -> Int
2 low2int c = ord c - ord 'a'
3
4 int2low :: Int -> Char
5 int2low n = chr (ord 'a' + n)
6
7 upp2int :: Char -> Int
8 upp2int c = ord c - ord 'A'
9
10 int2upp :: Int -> Char
11 int2upp n = chr (ord 'A' + n)
12
13 num2int :: Char -> Int
14 num2int c = ord c - ord '0'
15
16 int2num :: Int -> Char
17 int2num n = chr (ord '0' + n)
```

L'operazione di shift risulta quindi essere facilmente implementabile:

- se il carattere è una lettera, maiuscola o minuscola, viene convertita nel codice ASCII corrispondente (sottraendo rispettivamente l'offset del carattere 'A' o 'a'); poi viene sommato lo shift da eseguire, effettuando sul risultato l'operazione *modulo* 26; infine, il valore ottenuto viene riconvertito in un carattere ASCII;
- se il carattere è un numero viene convertito nel codice ASCII corrispondente (sottraendo l'offset del carattere '0'); viene quindi sommato lo shift da eseguire, effettuando sul risultato l'operazione *modulo* 10; infine, il valore ottenuto viene riconvertito in un carattere ASCII;
- se il carattere considerato non rientra in una delle categorie precedente, viene lasciato invariato (ad esempio spazi bianchi, caratteri di punteggiatura o speciali).

Queste operazione vengono eseguite dal seguente frammento di codice:

```
1 shift :: Int -> Char -> Char
2 shift n c | isLower c = int2low ((low2int c + n) `mod` 26)
3           | isUpper c = int2upp ((upp2int c + n) `mod` 26)
4           | isDigit c = int2num ((num2int c + n) `mod` 10)
5           | otherwise = c
```

Infine, è sufficiente che la funzione di shift, appena descritta, venga applicata ad ogni carattere della stringa fornita:

```
1 encode_c :: Int -> String -> String
2 encode_c n xs = [shift n x | x<-xs]
```

### 2.3.1 Esempio

Supponiamo di voler codificare la stringa:

"Il mio numero preferito e il 37!",

con uno shift di  $K = 5$ .



Procedendo si ottiene la frase decodificata:

"Nq rnt szrjwt uwjkjwnyt j nq 82!".

Per effettuare la decodifica, si applica lo stesso algoritmo, con  $K = -5$ , da cui si ottiene:

"Il mio numero preferito e il 37!".

## 2.4 Main

All'interno del main dell'applicazione, è stato inserito un blocco `do`, nel quale viene richiesto di inserire una frase all'utente da decodificare. Successivamente, l'utente, inserendo i caratteri "C" oppure "CS", seleziona il metodo di codifica desiderato. Se si utilizza il cifrario di Cesare, viene anche richiesto lo shift da applicare. Invece, se l'utente sceglie il metodo Crypto Square, viene mostrato il risultato. Poi, viene chiesto all'utente se si desidera vedere una versione normalizzata o no della stringa codificata.

```
1 main = do
2   putStrLn "inserisci una frase da criptare: "
3   phrase <- getLine
4   putStrLn "con quale metodo vuoi codificarla? Caesar cipher (C) oppure Crypto Square
      (CS)"
5   algorithm <- getLine
6   if algorithm == "C"
7   then do
8     putStrLn "Inserisci shift"
9     line <- getLine
10    let s = (read line :: Int)
11    print(encode_c s phrase)
12  else do
13    let result = encode_cs phrase
14    print(result)
15    putStrLn "vuoi vedere la forma normalizzata? Y/N"
16    yn <- getLine
17    if yn == "Y" then print(normalize result)
18    else exitWith ExitSuccess
```