# SeQrit - San Siro Stadio

Andrea Cecchin
matricola 2034299

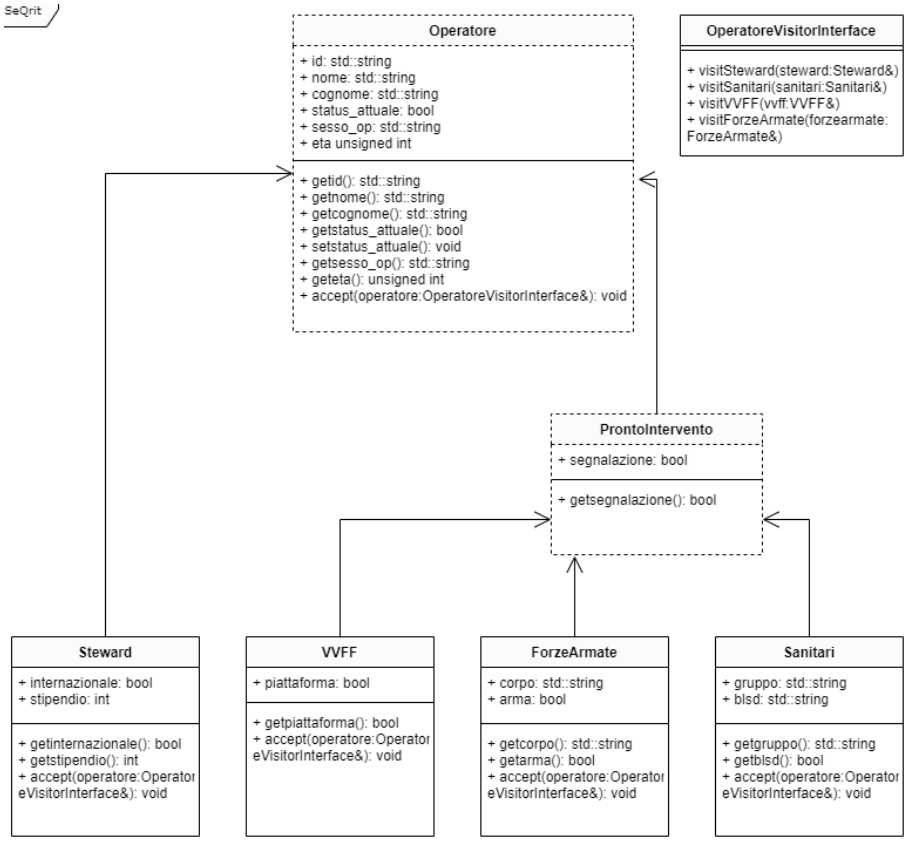
Relazione progetto per Programmazione ad Oggetti

# 1 Introduzione

SeQrit – San Siro Stadio è un'applicazione pensata per la gestione della sicurezza all'interno di uno stadio.
In particolare, SeQrit permette, caricato un "database" (file JSON) di operatori, di visualizzare per ogni persona impiegata nella messa in sicurezza le informazioni generiche (quale nome, cognome, sesso ed età), l'incarico ad esso assegnato e altre caratteristiche che variano in base al tipo di operatore selezionato.
Il gestionale, oltre alla visualizzazione dei dati delle persone già presenti in database, permette la registrazione ex-novo di un nuovo operatore, la sua cancellazione dall'elenco e la modifica, oltre a permettere una ricerca rapida di ogni persona in base al proprio codice identificativo univoco.
In particolare, basandosi su un modello quanto più vicino alla realtà delle aziende incaricate della security, gli operatori sono distinti in quattro categorie, quali steward, forze armate, sanitari e vigili del fuoco, dove i soli steward sono effettivi lavoratori stipendiati da chi gestisce la sicurezza dell'impianto, mentre i restanti ruoli vengono visti come più generici operatori di pronto intervento.

Ho scelto questo ambito per il mio progetto in quanto la struttura ben differenziata tra i vari operatori offriva un ottimo spunto per sfruttare qualche pattern grafico che prevede polimorfismo non banale, come i visitor, oltre al fatto di conoscere per esperienza l'ambito per l'ipotetico uso di questa applicazione.

# 2 Descrizione del modello




Il modello della gerarchia comincia da una classe astratta Operatore, la quale rappresenta le informazioni generiche di ogni persona, comuni ad ogni tipologia di lavoratore della gerarchia: codice identificativo univoco (id), nome, cognome, sesso, età e stato dell'operatore (un valore booleano che indica se l'operatore è schierato all'interno della struttura o se in attesa di assegnamento). Per ognuno di questi attributi, è presente il relativo metodo getter, oltre a un setter per quanto riguarda lo stato. Ho infatti deciso, attenendomi quanto più possibile alle situazioni reali, che effettivamente lo stato è l'unico attributo di cui si dovrebbe voler/poter cambiare il valore, in quanto tutte le altre informazioni sono di operatori il cui impiego in un evento è unico (basti pensare che gli steward seguono contratti a chiamata, dunque l'inserimento in database per uno specifico evento ha durata giornaliera, e che gli operatori di pronto intervento vengono assegnati dalle pubbliche autorità e variano da evento ad evento, e quindi a fine prestazione vengono rimossi dal database tanto quanto gli steward).

La classe Pronto Intervento, derivata di Operatore ed anch'essa astratta, racchiude tutti gli operatori non stipendiati dalla società addetta alla security, ma assegnati dalle pubbliche autorità. A tutti questi operatori viene assegnato un attributo booleano segnalazione, con relativo getter, che segnala un tentativo di comunicazione tra società e singolo operatore.
Derivati da Pronto Intervento, si trovano le classi Forze Armate (con attributi corpo e arma, per conoscere il corpo di appartenenza dell'agente e se ha con sé un'arma da fuoco), Sanitari (con attributi gruppo e blsd, per conoscere il gruppo di appartenenza del medico e se ha conseguito l'abilitazione all'utilizzo del defibrillatore) e VVFF/Vigili del Fuoco (con l'attributo piattaforma, per sapere se la persona ha l'abilitazione per usare carrelli elevabili).

Classe che deriva direttamente da Operatore è Steward, che implementa gli attributi stipendio (si ricorda essere l'unico tipo di lavoratore direttamente assunto e pagato dalla società) e internazionale, un booleano per sapere se possiede qualche attestato di conoscenza elevata di un'altra lingua.

Per contenere gli oggetti di questa gerarchia di classi, è stato sviluppato un array dinamico in grado di contenere dei puntatori ai vari operatori. Per fare questo è stato sviluppato un apposito template per l'array.
È stata implementata inoltre una classe OperatoreVisitorInterface, quale riferimento per tutti i visitor sfruttati come pattern grafico di polimorfismo avanzato: grazie al loro utilizzo, è possibile vedere i dati specifici ad ogni tipologia di operatore, oltre a determinare un migliore incapsulation del codice nella window dell'applicazione. A tal proposito, nelle quattro classi della gerarchia sono stati implementati i relativi metodi accept() per consentire l'utilizzo dei visitor.

# 3 Polimorfismo

L'utilizzo principale e più elevato di polimorfismo non banale implementato in questo progetto riguarda i Visitor.
Essi vengono utilizzati per la costruzione dinamica dell'interfaccia grafica, andando a restituire un widget diverso in base al tipo dell'oggetto visitato.
In particolare, i visitor vengono chiamati dall'interazione dell'utente con un QListWidget popolata con tutti i codici identificativi degli operatori caricati da database.
I Visitor che utilizzano OperatoreVisitorInterface, e rispettivo compito, sono:

- OperatoreGeneralVisitor: mostra i dati generici di ogni operatore. In quanto questi dati sono comuni a tutte le tipologie di lavoratore, l'uso di un visitor è esagerato, ma ho deciso ugualmente di utilizzarlo sfruttando così l'interfaccia visitor che avrei comunque dovuto implementare per altri visitor più specifici, così da riciclare codice.

- OperatoreImageVisitor: mostra l'immagine corretta dell'operatore selezionato. Essa varia in base al ruolo e al sesso della persona, così da rendere

visivamente più veloce determinare due informazioni.

- OperatoreInfoVisitor: mostra i dati più specifici di ogni operatore, ovvero quelli che si diversificano in base alla tipologia di lavoratore selezionato.
- OperatoreStatoVisitor: mostra, in base al valore dello stato attuale dell'operatore che lo visita, l'immagine che identifica un operatore in attesa o dispiegato.
- OperatoreWarningVisitor: restituisce un booleano che indica se mostrare o meno l'icona di segnalazione.

Un polimorfismo di entità minore è stato utilizzato nella persistenza dei dati, in particolare nel salvataggio degli oggetti contenuti nel contenitore verso il file, dove i differenti metodi di salvataggio vengono stabiliti tramite dynamic cast.

# 4 Persistenza dei dati

Per adempiere alla richiesta di persistenza dei dati, ho optato per un file strutturato, in particolare per il formato JSON. Questo perché la struttura del file è abbastanza semplice ed intuitiva da utilizzare, l'utilizzo delle chiavi è molto comodo, oltre al fatto che Qt implementa molte librerie indirizzate alle operazioni di input output da/per file JSON che hanno reso l'implementazione di questa scelta alquanto semplice.

Con il progetto, sono stati preparati tre file JSON da utilizzare come test per l'applicazione: uno denominato "database" che contiene vari operatori al suo interno, comunque di tutte le tipologie, così da poter osservare la struttura per ogni ruolo di lavoratore e poter provare sia le funzionalità di modifica ed eliminazione, così come l'aggiunta; un file "fulldatabase" e un "emptydatabase" che contengono rispettivamente 396 (il massimo possibile) e zero operatori, così da mostrare il corretto funzionamento dell'applicazione anche nei due "casi limite".

# 5 Funzionalità implementate

Funzionalità logico/funzionali:

- Gestione di quattro differenti tipologie di operatore
- Caricamento e salvataggio dei dati in formato JSON
- Salvataggio automatico ad ogni modifica delle informazioni caricate in partenza
- Presenza di scorciatoie da tastiera per l'utilizzo di pulsanti, con apposite segnalazioni visibili graficamente (lettera della shortcut sottolineata nei pushbutton)
- Ricerca di uno specifico operatore in lista in base al proprio codice ID

- Gestione di possibili errori a run-time (file da caricare non selezionato, tentativo di associazione di un codice già in uso a un nuovo operatore ecc...) tramite semplici controlli booleani e pop-up QMessageBox
- Gestione del controllo dell'intera applicazione e del comportamento dei widget tramite funzioni connect()

Funzionalità grafiche:

- Barra di ricerca con testo nello sfondo a spiegare la sua funzione
- Utilizzo di un'icona per il pulsante di ricerca
- Lista con colore delle righe alternate per vivacizzarne la visione, nonche semplificarla
- Utilizzo di colori in tono, della scala bianco nero, per sfondo dell'applicazione, della lista e dei pulsanti
- Comparsa del logo dell'applicazione all'accensione e durante le operazioni di aggiunta di un operatore, così da riempire lo schermo e rendere più apprezzabile visivamente l'interfaccia grafica
- Associazione del logo semplificato come icona dell'applicazione, e settaggio del titolo della finestra con il nome dell'app
- Shortcut da tastiera segnalate attentamente nel testo dei pulsanti mediante sottolineatura del carattere in questione
- Messaggi pop-up informativi in caso di determinate azioni
- Visualizzazione dinamica delle informazioni specifiche di ogni operatore
- Pannello informativo dell'operatore selezionato composto da una serie di Visitor
- Utilizzo di immagini nel pannello informativo, per rendere più immediata la lettura delle informazioni, e come abbellimento della grafica
- Utilizzo di un pannello per l'aggiunta di un operatore dinamico in base alla tipologia di lavoratore che si vuole aggiungere

# 6 Rendicontazione ore di lavoro*

| **Sezione di lavoro** | **Tempo impiegato** |
|---|---|
| Analisi, studio e progettazione | 6 ore |
| Sviluppo codice del modello | 11 ore |
| Studio documentazione framework Qt | 12 ore** |
| Sviluppo codice GUI | 12 ore |
| Test e debug | 4 ore |
| Stesura relazione | 5 ore |

* Le ore segnate sono da considerarsi approssimative, in quanto arrotondate all'ora intera più vicina

** Sono state conteggiate anche le 5 ore circa effettive di introduzione a Qt svolte a lezione

Complessivamente il monte ore richiesto è stato rispettato, e solo leggermente sforato causa una non sempre lineare progettazione secondo l'ordine ottimale spiegato a lezione.

# 7 Sviluppo e test

L'applicazione è stata sviluppata su sistema operativo Windows 10.0.17763 con framework Qt 6.2.4.
Come richiesto, l'applicazione funziona correttamente anche su macchina virtuale Ubuntu 22.04 LTS appositamente fornita per il progetto.