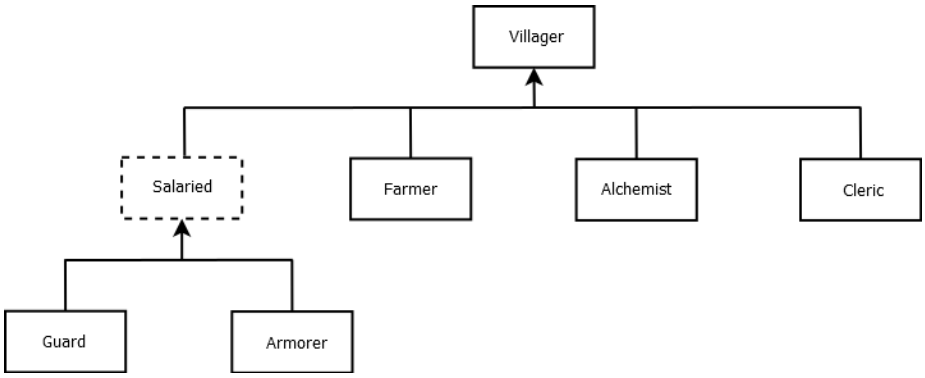
Progetto di Programmazione ad Oggetti: Qontainer

Marco Siragna 1161572

a.a. 2018/2019

1 Gerarchia e Polimorfismo

La gerarchia scelta è ispirata al modello di un villagio medievale. In particolare, riguarda la suddivisione degli abitanti per professione.



- `Villager`: classe base della gerarchia che rappresenta un comune abitante senza una professione. È quindi istanziabile.

  Campi dati:

  - `string name`: nome dell'abitante.
  - `const Gender* gender`: sesso dell'abitante (uomo, donna, altro).
  - `QDate birthDate`: data di nascita dell'abitante.

  Metodi polimorfi:

  - `virtual const VillagerType* getType()`: restituisce il tipo di abitante, rappresentato da una stringa testuale e un indice/id. Viene reimplementato nelle classi derivate conformemente alla classe stessa.
  - `virtual double getTaxes()`: restituisce il valore delle tasse che l'abitante deve pagare. Di default consiste da una tassa base uguale per tutti. Viene reimplementato nelle classi derivate in base anche ai beni prodotti e posseduti.
  - `virtual bool load(const QJsonObject&)`: carica l'abitante dal `QJsonObject` e restituisce il successo dell'operazione di caricamento. Viene reimplementato nelle classi derivate a seconda dei campi dati da caricare.
  - `virtual void save(QJsonObject&)`: salva l'abitante nel `QJsonObject`. Viene reimplementato nelle classi derivate a seconda dei dati da salvare.

- `Salaried`: classe astratta che rappresenta un abitante retribuito.

  Metodi polimorfi:

  - `virtual double getSalary()`: metodo virtuale puro che restituisce il valore dello stipendio retribuito. Viene reimplementato nelle classi derivate in base alla qualità del lavoro svolto.

- `Guard`: classe che rappresenta una guardia a difesa del villaggio.

  Campi dati:

  - `unsigned int kills`: numero di nemici uccisi.
  - `const Rank* rank`: grado di esperienza (recluta, esperto, veterano).

- `Armorer`: classe che rappresenta un armaiolo che produce gli armamenti per la difesa del villaggio.

  Campi dati:

  - `unsigned int swords`: numero di spade prodotte.
  - `unsigned int armors:` numero di armature prodotte.

- `Farmer`: classe che rappresenta un contadino.

  Campi dati:

  - `unsigned int crops`: quantità di raccolto ricavata dalle colture.
  - `unsigned int livestock:` numero di animali da allevamento posseduti.

- `Alchemist`: classe che rappresenta un alchemista.

  Campi dati:

  - `unsigned int potions`: numero di pozioni prodotte.

- `Cleric`: classe che rappresenta un ecclesiastico.

  Campi dati:

  - `double donations`: valore ricavato dalle donazioni per beneficenza.

2 Container

Il `Container` è stato implementato come una lista singolarmente linkata (`forward_list`) in quanto non sono richiesti accessi casuali ma viene scorsa l'intera lista di elementi in avanti. In questo modo viene anche ridotto l'overhead rispetto all'utilizzo di una lista doppiamente linkata. Inoltre, l'aggiunta e la rimozione di elementi risultano essere poco costose rispetto ad altri tipi di strutture dati.

3 Persistenza

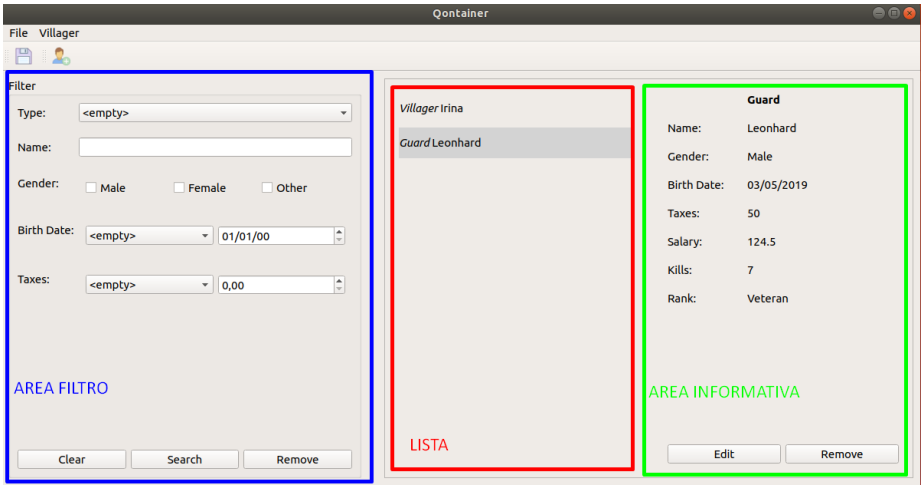
Il formato di caricamento e salvataggio utilizzato è JSON. Il `Model` si occupa di salvare e caricare gli elementi del `Container` invocando i rispettivi metodi polimorfi `save` e `load` della classe `Villager`.

4 Design Pattern Model-View

È stato adottato il pattern Model-View, il quale è usato anche dal framework Qt in, quanto è stato ritenuto più semplice da implementare e più conciso. In tal caso, la parte del Controller viene svolta dalla View.
Il codice che riguarda la gerarchia è quindi separato da quello che riguarda la parte grafica. Tuttavia, per semplicità e dovuto alla scelta del formato del caricamento/salvataggio e alla mancanza di un formato "data" nella libreria std, sono stati utilizzati nella gerarchia alcuni elementi di Qt Core quali `QDate` e il supporto per il formato JSON. È stata ritenuta trascurabile questa scelta ai fini del progetto e dell'indipendenza della gerarchia rispetto al framework Qt.

5 Manuale Utente

All'avvio dell'applicazione gli abitanti e la tasse base impostata veranno caricati automaticamente dal file `data.json` che si trova nella cartella dalla quale viene avviata l'applicazione.



Tassa Base Per modificare la tassa base per ogni abitante è necessario navigare dalla menubar `Villager` → `Base Tax`. Apparirà una finestra in cui è possibile cambiare il valore corrente. Se non è mai stato impostato, di default è 50.0.

Creazione È possibile creare un nuovo abitante navigando dalla menubar `Villager` → `New`, oppure cliccando sull'icona corrispondente della toolbar, o infine usando lo shortcut `Ctrl+N`. Apparirà una finestra in cui scegliere la professione ed inserire i relativi dati. Il pulsante di conferma sarà abilitato solo se tutti i dati verranno inseriti correttamente.
Una volta creato, l'abitante apparirà nella lista scorrevole, ammesso che rispetti il filtro impostato.

Visualizzazione Per visualizzare le informazioni complete relative ad un abitante basta selezionarlo nella lista cliccandoci sopra ed appariranno di fianco nell'area dettagli, insieme a due bottoni per la modifica e l'eliminazione.

Eliminazione È possibile eliminare un abitante cliccando, dopo averlo selezionato, sull'apposito pulsante `Remove` dell'area dettagli.
In alternativa, è possibile eliminare tutti gli abitanti che soddisfano il filtro impostato, ovvero che appaiono correntemente nella lista, cliccando il pulsante `Remove` dall'area filtro. Successivamente, il filtro verrà resettato in automatico.

Modifica Per modificare un abitante è necessario, dopo averlo selezionato, cliccare sull'apposito pulsante `Edit` presente nell'area dettagli.

Ricerca È possibile cercare uno o più abitanti che rispettano alcune caratteristiche impostando i rispettivi campi nell'area filtro e poi cliccando il pulsante `Search`. I risultati appariranno nella lista.
Per resettare il filtro corrente e visualizzare nella lista tutti gli abitanti basta cliccare sul pulsante `Clear` dell'area filtro.

Salvataggio Alla chiusura dell'applicazione, se sono state apportate delle modifiche, verrà chiesto se si desidera salvare oppure ignorare e chiudere l'applicazione.
In alternativa, è comunque possibile salvare in ogni momento navigando dalla menubar `File` → `Save`, oppure cliccando sull'icona corrispondente nella toolbar, o infine usando lo shortcut `Ctrl+S`.

6 Compilazione ed Esecuzione

Il progetto prevede l'utilizzo di funzionalità di C++11 (lambda expressions, keywords `nullptr` e `auto`), pertanto, viene fornito il file `Qontainer.pro`, diverso da quello ottenuto eseguendo il comando `qmake -project`.
Per la compilazione è quindi necessario eseguire in sequenza i comandi `qmake` e `make`, supponendo di trovarsi correntemente nella cartella del progetto nel terminale.
Per l'esecuzione basta eseguire il comando `./Qontainer`.

7 Sviluppo

Per la realizzazione del progetto sono state impiegate all'incirca 50 ore, di cui:

- 2 ore per l'analisi del problema;
- 8 ore per la progettazione e codifica del modello (gerarchia + container);
- 15 ore per l'apprendimento del framework Qt;
- 20 ore per la progettazione e codifica della GUI;
- 5 ore per testing e debugging.

Le specifiche dell'ambiente di sviluppo sono le seguenti:

- Sistema operativo: Windows 10 Home;
- IDE: CLion 2019.2;
- C++11;
- Qt 5.12.0.