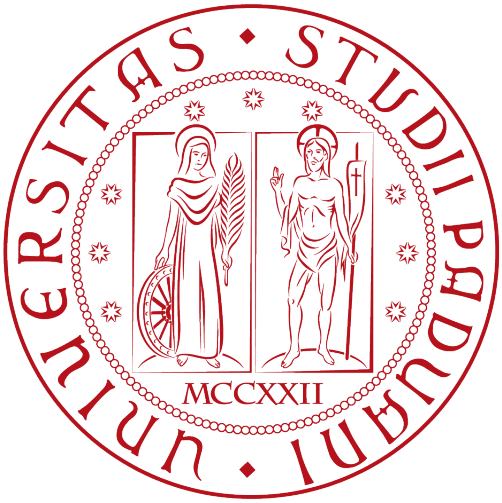
UNIVERSITÀ DEGLI STUDI DI PADOVA

SCUOLA DI SCIENZE



CORSO DI LAUREA IN INFORMATICA

PROGETTO

DI

PROGRAMMAZIONE AD OGGETTI

PAO: Pets And Owners

Federico Silvio Busetto Matricola:1026925

ANNO ACCADEMICO 2015-2016

Indice

Capitolo 1. Scopo del Progetto

1.1 Abstract

E' stata sviluppata un'applicazione Desktop denominata Pets and Owners (da qui in poi PAO).
L'applicazione si prefigge di modellare e gestire, molto semplicemente, la parte amministrativa di un piccolo database di una clinica veterinaria fornendo un'interfaccia utente semplice ed intuitiva sviluppata in *Qt*.

1.2 Note per la compilazione

Per compilare il progetto, viene fornito un file .pro. E' necessario utilizzare la versione 5.3.2 o successiva delle librerie *Qt* seguendo le istruzioni riportate all'url a fine pagina

- lanciare lo script per la compilazione *qt-532.sh* (non incluso nella consegna)
- Posizionarsi nella cartella del progetto contente i sorgenti
- lanciare *qmake*
- *lanciare make ./PAO*
  Vengono inoltre forniti 3 file *.json* che compongono un piccolo DB dimostrativo da inserire nella stessa cartella contenente il file eseguibile.

L'applicazione è stata sviluppata usando:

- QtCreator 3.2.1
- Qt 5.3.2
- GCC 4.8.6 e GCC 5.3.0
- Ubuntu 16.04 Xenial Xerus 64 Bit - Windows 7 Professional 64 Bit, su quest'ultimo aggiungendo le direttive per QMake *INCLUDEPATH += . e DEPENDPATH += .*

Nota: Su Windows la build viene eseguita correttamente da QtCreator ma usando MinGW da linea di comando si potrebbe incappare in un'errore, si sconsiglia quindi una compilazione sotto questo ambiente. Ulteriori dettagli seguendo il secondo link indicato a fine pagina.

1.3 Vincoli obbligatori

- **Definizione ed utilizzo di una gerarchia G di tipi di altezza ≥ 1 e larghezza ≥ 1:**
  A tal proposito è stata sviluppata una gerarchia di Visite, descritta in seguito.
- **Definizione di un opportuno contenitore C, con relativi iteratori, che permetta inserimenti, rimozioni, modifiche.**: E' stato sviluppata una classe *PAOContainer* per la gestione delle visite che verrà illustrata nella sezione *Orientamento agli oggetti*
- **Utilizzo del contenitore C per memorizzare oggetti polimorfi della gerarchia G.**
  Il *PAOContainer* memorizza e gestisce puntatori polimorfi di *AbstractVisit*, l'uso di puntatori è necessario per utilizzare polimorfismo e i metodi virtuali descritti in seguito.
- **Il front-end dell'applicazione è una GUI sviluppata usando il framework Qt.**

[1] `http://www.studenti.math.unipd.it/index.php?id=corsi#c688`
[2] `http://goo.gl/NQGlk3`

Capitolo 2. Orientamento agli Oggetti

2.1 Descrizione delle Gerarchie

L'intera applicazione si compone di 20 classi (12 per le gerarchie principali, 3 di tipi utilità e 6 per la GUI). La gerarchia principale, nonché la più importante è la gerarchia delle visite. Essa si compone di quattro classi:

- ***AbstractVisit***: Una classe virtuale pura (non istanziabile direttamente), contiene 3 metodi virtuali puri, uno per scrivere su file i campi contenuti nelle classi derivate, uno per calcolare il prezzo finale della visita e uno di utilità che ritorna una stringa indicante il tipo di Visita.
- ***BasicVisit***: Deriva pubblicamente da *AbstractVisit* e calcola il costo finale, *override* dell'apposito metodo virtuale puro, restituendo il prezzo base.
- ***VaccineVisit*** Deriva pubblicamente da *AbstractVisit* e calcola il costo finale tramite override, come il doppio del prezzo base di una visita di base.
- ***SpecializedVisit*** Deriva pubblicamente da *AbstractVisit* e calcola il costo finale con un 30 % aggiuntivo rispetto al prezzo di un Vaccino.

2.1.1 Utilità e ruolo dei tipi

Oltre alle quattro classi descritte sopra, ve ne sono altre:

- ***AbstractOwner***: Modella un generico proprietario di animali (può essere un veterinario), è una classe virtuale pura, madre della seconda gerarchia. Contiene due metodi virtuali puri, usati per restituire una stringa e per salvare i dati su file; contiene inoltre un puntatore ad una classe *OwnerAccount* e una stringa per memorizzare il Codice Fiscale.
- ***Owner***: Deriva pubblicamente da *AbstractOwner* e rappresenta un proprietario privato di animali. Contiene un puntatore ad una classe *OwnerData*.
- ***OwnerAccount***: Utilizzata per funzionalità future. Permette di distinguere se un utente è o no un veterinario. Inoltre fornisce delle funzionalità di Encryption della password, memorizzata tramite un hash.
- ***OwnerData***: Contiene le informazioni di un proprietario, quali nome, cognome e informazioni di contatto, queste ultime tramite un campo dati di tipo *ContactInfo*
- ***ContactInfo***: Utilizzata in *OwnerData*, descrive indirizzo fisico, mail e numero di telefono di un proprietario.
- ***Animal***: Descrive un animale generico. E' una classe virtuale pura, madre della terza gerarchia utilizzata dall'applicazione. Contiene due metodi virtuali puri per restituire una stringa e per salvare i campi dati su file. Contiene inoltre un campo ad una classe *AnimalData*
- ***Pet***: Estende pubblicamente *Animal* e rappresenta un animale domestico, caratterizzato dal tipo (Cane, Gatto, Volatile o Altro), dal colore del pelo e dalla razza dell'animale.
- ***AnimalData***:Contiene le informazioni essenziali di un animale, quali nome, data di nascita, sesso e peso ed eventualmente data di applicazione del microchip.

Vi sono poi due classi d'utilità: Il contenitore delle visite e il database vero e proprio. Le ulteriori sei classi che modellano la parte grafica verranno trattate nella sezione *Interfaccia Grafica*.

- ***PaoDB***: Gestisce gli utenti tramite una *QMap<QString, AbstractOwner*>*, gli animali tramite una *QMap<QString, Animal*>* e le visite tramite il contenitore preposto, *PaoContainer*.
  La classe fornisce metodi per inserire, modificare, rimuovere proprietari, animali e visite, per leggere i dati delle tre entità da file e caricarli in memoria e per salvarli su disco rigido in formato *JSON*

2.1.2 Descrizione del contenitore

La classe PaoContainer altro non è che una lista che permette inserimenti in testa ed in coda.
Essa si avvale di due classi interne private: *node*, la lista vera e propria di nodi e *SmartPointer* per la gestione della memoria condivisa, così come visto durante il corso.
Il contenitore mette a disposizione per iterare sui suoi elementi due tipi di iteratori: Un *iterator* per accedere normalmente agli elementi del contenitore e un *const_iterator* utilizzato per la sola lettura.
La classe *PaoContainer* è amica degli iteratori e viceversa, questo permette agli iteratori di accedere alla parte privata di *PaoContainer* e a *PaoContainer* di accedere alla parte privata degli iteratori.

PaoContainer fornisce funzioni per determinare la sua dimensione (*size()*), per verificare se è vuoto (*isEmpty()*) per l'inserimento in testa (*push_front*) e in coda (*push_back*), per ottenere una visita da modificare, per ritornare l'iteratore iniziale e l'iteratore *Past the End* e per rimuovere visite. I suoi due campi dati sono uno *SmartPointer* che punta alla testa del Contenitore e un intero che ne rappresenta la size, cioè il numero di elementi contenuti. Container fornisce inoltre l'*overloading* dell'operatore di accesso a membro *operator[]*, altri operatori sono stati ridefiniti nelle classi interne.

2.2 Incapsulamento

Si è scelto di garantire un incapsulamento fornendo alle varie classi, alcune funzioni cercando di usare, quanto più, metodi costanti e gli specificatori di accesso *private* e *protected* rendendo *public* solo dei metodi *getter* e *setter* di interfaccia.
Questo, da una parte permette di non accedere direttamente ai campi dati ma dall'altra espone in parte la struttura delle classi. Alla fine, dopo averci riflettuto su a lungo, per quanto la soluzione non sia delle migliori e visti gli scopi del progetto è risultata essere la scelta più conveniente.
Un incapsulamento soddisfacente è stato ottenuto per il salvataggio dei dati delle singole classi su file, dove ogni classe è responsabile di salvare i propri campi dati.

2.3 Modularità

Ogni classe è stata suddivisa in un file di header .h contenente le dichiarazioni e in un file .cpp contenente le definizioni implementative. Si è cercato inoltre di suddividere i compiti in varie funzioni secondo il principio della *Single-Responsibility*, ovvero un compito specifico per ciascuna funzione.

2.4 Estensibilità \ Evolvibilità

Si è scelto di far partire le tre gerarchie con delle classi astratte, in modo da permettere una eventuale estensione futura del software. Ad esempio, si potranno creare dei diversi tipi di proprietario derivando da *AbstractOwner*, per rappresentare uno zoo o un parco nazionale. Si potrà derivare *Animal* per nuove categorie di animali (esotici, selvaggi, da giardino) e soprattutto si potranno creare nuovi tipi di visite a partire da *AbstractVisit*, includendo, ad esempio, un accertamento radiologico con un prezzo appositamente calcolato.

[3] `http://goo.gl/CPF3up`
[4] `https://dzone.com/articles/getter-setter-use-or-not-use-0`

2.4.1 Esempi d'uso di codice polimorfo

```
double SpecializedVisit::calcPrice() const{
  return (basic_price*2)+(basic_price*2)*0.3;

}
```

cpp/specializedvisit.cpp

Un esempio di *override* di un metodo virtuale puro, per il calcolo del costo finale di una visita specialistica.

```
void VetControl::putOwnersOnTable() {
  QMap<QString, AbstractOwner*> us = db->getOwnerList();
  foreach (AbstractOwner* abso, us){
    Owner *own_ptr = dynamic_cast<Owner*>(abso);
    QString tmpCF = own_ptr->getFiscalCode();
    QString tmpName = own_ptr->getOwnerInfo()->getName();
    QString tmpSur = own_ptr->getOwnerInfo()->getSurname();
    QString tmpAddr = own_ptr->getOwnerInfo()->getContacts().getAddress();
    QString tmpMail = own_ptr->getOwnerInfo()->getContacts().getMail();
    QString tmpPh = own_ptr->getOwnerInfo()->getContacts().getPhone();
    QStringList nh;
    nh << tmpCF << tmpName << tmpSur << tmpAddr << tmpMail << tmpPh;
    putInTable(nh);
  }
    tbl->resizeRowsToContents();
}
```

cpp/vetcontrol.cpp

Un esempio di RTTI per estrarre i dati dei proprietari da aggiungere alla tabella.

```
void SpecializedVisit::saveObj(QJsonObject& js) const {
  saveAbsVisit(js);
  js["type"] = this->typeOfVisit();
  js["complete_price"] = this->calcPrice();
}
```

cpp/specializedvisit.cpp

Sempre sulla visita specialistica, un esempio di *override* per salvare i dati su file.

Nota: Si voleva utilizzare il marcatore esplicito *override* previsto da *C++11*, in modo da rendere più chiaro quali metodi sono degli override all'interno delle classi, tuttavia ciò non è stato possibile in quanto la versione di GCC presente in laboratorio non supportava ancora tale funzione.

2.5 Efficienza \ Robustezza

Si è cercato di rendere robusta soprattutto l'interfaccia grafica tramite dei controlli elementari sui campi dati inseriti, ad esempio le chiavi non possono essere una *QString* vuota.

Capitolo 3. Funzionalità

Le funzionalità messe a disposizione dall'applicazione *PAO* sono:

- ***Aggiunta \ Rimozione \ Modifica***: Di proprietari e animali sfruttando le peculiarità di *QMap*. Ogni modifica avverrà tramite un inserimento di nuovi valori utilizzando la stessa chiave. Dal punto di vista grafico però non è il massimo.
- ***PaoContainer***: Il contenitore, già descritto in precedenza. Permette la modifica, l'aggiunta e la rimozione di Visite.
- ***Encryption***: Ogni classe *AbstractOwner* ha un campo di tipo *OwnerAccount* che a sua volta memorizza un *QCryptographicHash* che rappresenta l'hash della password di un utente. La classe è pronta all'uso, in vista di future evoluzioni del software che potranno includere una schermata di login e un lato client.
- ***Salvataggio su file***: Tramite apposite funzioni, viene aperto un file in sola scrittura per ogni tipo di entità principale (Owner, Animal e Visit), viene allocato un *QJsonObject* principale e successivamente, viene creato un *QJsonArray* di *QJsonObject*.
  Per ogni elemento dei contenitori, vengono invocati i metodi *save* dei singoli oggetti definiti nelle varie classi. Il contenuto viene quindi riversato nell'array ed infine, tramite il metodo *write* di *QJsonDocument*, viene effettuata la scrittura vera e propria su file.
- ***Caricamento da file***: Viene fatto il processo opposto a quello di salvataggio, leggendo dai file scritti in precedenza tramite l'invocazione di apposite funzioni e i costruttori *JSON* dei vari oggetti. Per implementare il tutto si è fatto uso della documentazione *Qt* online. Un esempio di documentazione utilizzata è reperibile a fondo pagina .

3.1 Interfaccia Grafica (GUI)

Per la parte di interfaccia grafica si è cercato di dividere quanto più possibile la parte logica che interagisce con il database (il model) dalla parte grafica vera e propria (la view), secondo un pattern *MVC*. Per implementare questo pattern si è preferito creare due classi principali, descritte di seguito, piuttosto che avvalersi del MVC stile *Qt* anche se quest'ultimo avrebbe garantito maggior riusabilità del codice e risultati migliori. Le due classi sviluppate sono:

- ***vetcontrol***: Interagisce col database, predisponendo gli elementi nella tabella e richiamando i metodi di inserimento, modifica e rimozione di elementi. Apre inoltre i pannelli per l'inserimento e la modifica di nuovi elementi.
  Si è pensato di generare i nuovi pannelli all'interno del controller, anche se sarebbe stata una cosa prettamente di competenza della view, in quanto questi ultimi interagiscono anche loro col database sottostante.
- ***vetview***: La vista amministrativa principale. Questa vista è divisa in tre sezioni tramite cinque *QGroupBox*.
  A sinistra abbiamo i pulsanti per aprire i pannelli di inserimento, a destra, dinamicamente, viene visualizzato uno dei tre *QGroupBox* per modifica e rimozione (in base al valore della *QComboBox* presente nel *Widget* centrale) e al centro abbiamo, oltre alla già menzionata *QComboBox*, un *QTableWidget* e un *QPushButton* d'uscita.
  Una funzione principale prepara la GUI, impostando i vari elementi grafici e un'altra imposta le *connect* per associare tra loro vari eventi secondo il sistema di *signal* e *slot* fornito dal framework *Qt*.

[5] `https://doc.qt.io/qt-5/qtcore-json-savegame-example.html`

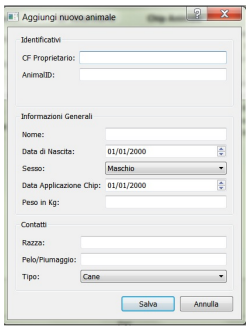


Figura 3.2: Pannello Aggiunta animali

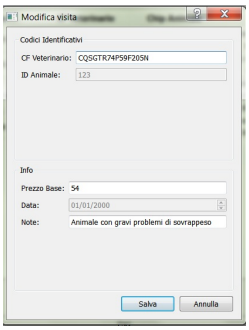


Figura 3.3: Pannello Modifica Visite

La maggior parte dei metodi di questa classe, sono metodi privati, non accessibili all'esterno della stessa.

Rimangono quattro classi che implementano i pannelli di inserimento e quello di modifica.

- ***AddOwnerDialog***: Permette l'inserimento di proprietari. Delle *QLineEdit* permettono all'utente di inserire i vari dati, uno slot pubblico si occuperà al momento del salvataggio di fare un minimo di controlli, creare un *Owner* e aggiungerlo alla mappa del database passando attraverso il controller.
- ***AddAnimalDialog***: Analoga a *AddOwnerDialog*
- ***AddVisitDialog***: Analoga a *AddOwnerDialog* e a *AddAnimalDialog*
- ***EditVisitDialog***:Permette la modifica di una visita selezionata. E' lo stesso pannello di inserimento, riadattato per disabilitare alcune componenti grafiche.

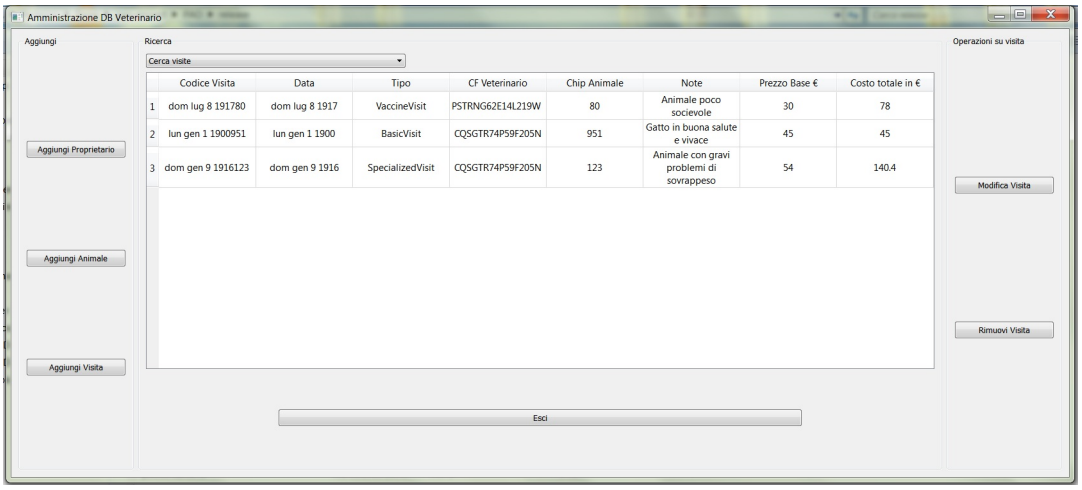


Figura 3.1: La finestra d'amministrazione principale con le visite caricate

3.1.1 Nota

All'inizio la vista è impostata su modalità *Owner* ma la tabella appare vuota. Cambiando il valore della *QComBobox* verranno caricati dinamicamente i dati richiesti.