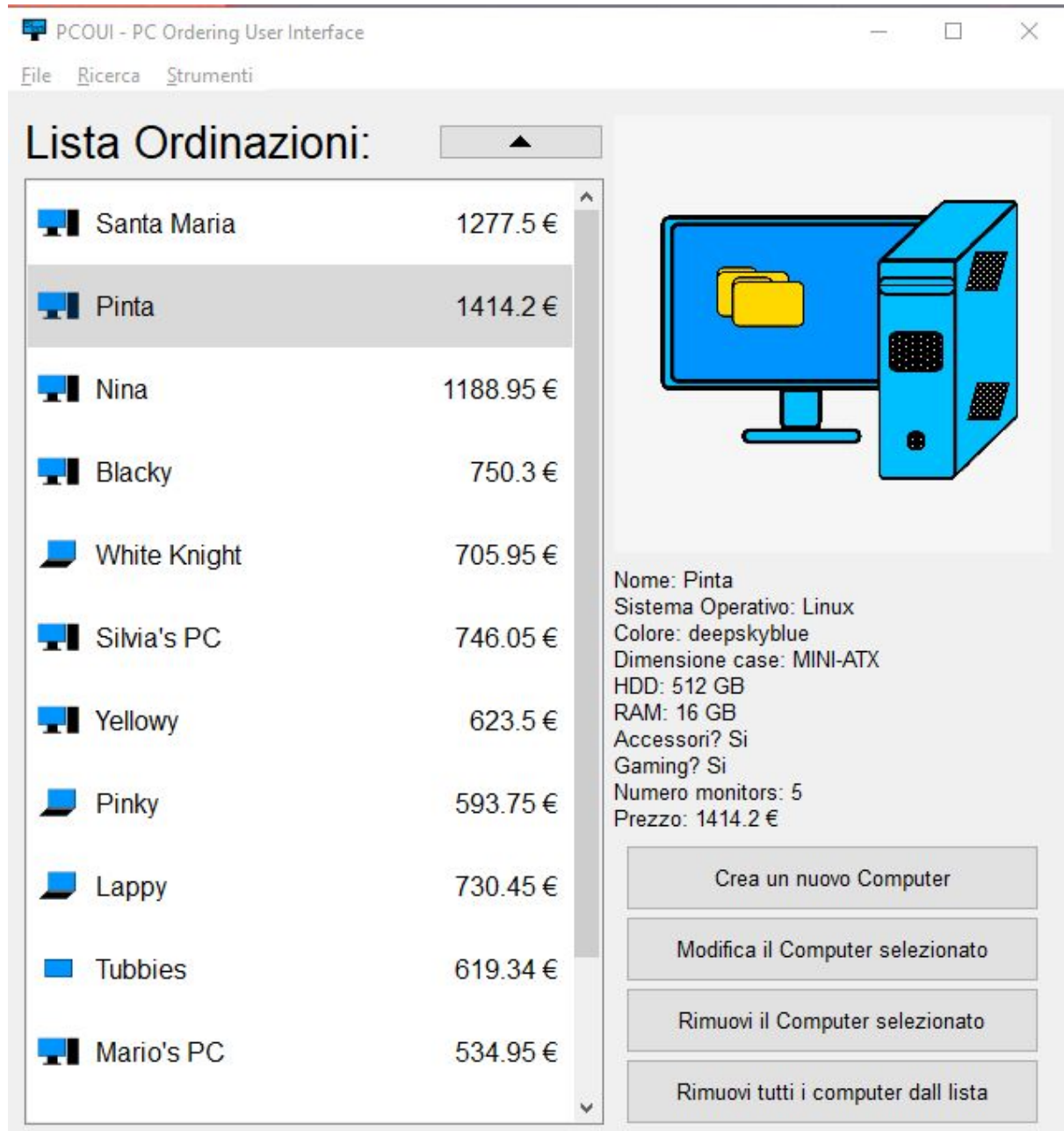


# RELAZIONE PROGETTO P2



## PC - ORDERING USER INTERFACE

<b>INTRODUZIONE</b>	<b>3</b>
<b>CLASS LAYOUT</b>	<b>3</b>
Hierarchy:	3
GUI:	3
Elements GUI:	4
QComboBoxes	4
ComputerListItem	4
StyleGUI	4
<b>POLIMORFISMO</b>	<b>5</b>
<b>SALVATAGGIO / CARICAMENTO</b>	<b>5</b>
<b>MANUALE UTENTE</b>	<b>5</b>
BASELAYOUT	5
INSLAYOUT	6
SEARCHDIALOG	6
FILE MENU	6
VIEW MENU	6
ABOUT MENU	7
TOOLS DIALOG	7
WARNING DIALOG	7

# INTRODUZIONE

Questo progetto si basa sulla creazione di un'applicazione Qt per gestire un servizio di ordinazioni di computer. Che fornisce ad un utente funzionalità di creazione, modifica, rimozione etc. sul Qontainer che contiene gli oggetti Computer.

## CLASS LAYOUT

Questa applicazione segue il modello MVC (Model View Controller). Quindi l'utente interagisce solo con il Controller, che a sua volta comunica con: il Model e la View. Il Model (Parte logica dell'applicazione) non è collegato in nessun modo alla View (GUI) infatti il Model potrà essere utilizzato in un Framework diverso da Qt.

Questo progetto è stato realizzato seguendo le seguenti cartelle: **hierarchy, GUI, elementsGUI**.

### Hierarchy:

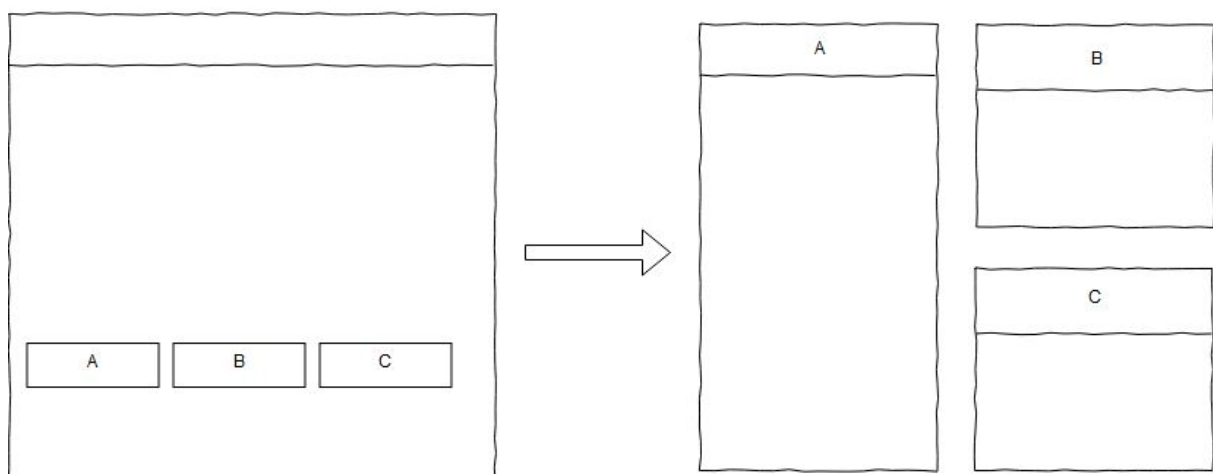
Il Modello viene gestito da una classe Store, che inizializza il Qontainer di oggetti. Computer é la classe base astratta; le classi Desktop, Laptop e Tablet sono classi children istanziabili. Secondo le necessità questa gerarchia può essere ampliata inserendo nuove classi children istanziabili o aggiungendo altre sottoclassi alle classi Desktop, Laptop e Tablet.

- Store
- Computer **-BASE ASTRATTA**
  - Desktop **-SOTTOCLASSE**
  - Laptop **-SOTTOCLASSE**
  - Tablet **-SOTTOCLASSE**

Oltre a questa gerarchia di classi ho realizzato un template di classe Qontainer che serve a contenere oggetti di tipo Computer. Nella classe Store ho utilizzato due Qontainer, uno per avere a disposizione gli oggetti Computer dopo un caricamento di file Json. L'altro, vuoto, viene riempito al momento della ricerca a partire dal Qontainer. Se la ricerca viene resettata allora anche il Qontainer utilizzato per la ricerca sarà resettato.

### GUI:

La GUI è gestita grazie alla classe Controller che comunica con il Model e la GUI per rendere l'interfaccia utente responsiva. Infatti ad ogni azione dell'utente la classe Controller impartisce gli "ordini" a Model e dopo aver ottenuto la risposta chiama l'update della GUI. La classe Controller contiene tutti i metodi principali per il funzionamento dell'applicazione: i metodi di popolamento della QListWidget, la creazione dei connect, gli slot delle operazioni della GUI etc. La GUI dell'applicazione è realizzata seguendo questo schema:



La schermata principale presenta collegamenti a altre schermate che completano la view con informazioni e funzioni. Questa scelta è stata adottata per mantenere ordine nella schermata principale e aumentare la modularità dell'applicazione in caso di cambiamenti.

In questa applicazione la schermata principale, chiamata `BaseLayout` viene utilizzata per mostrare la lista dei Computer all'interno di Store. E come spiegato prima sono presenti dei collegamenti a queste schermate:

- `InsLayout`: inserimento, modifica oggetti
- `SearchDialog`: ricerche oggetti
- `ToolDialog`: costo totale ordine corrente
- `MyFileDialog`: apertura / salvataggio di un file

Per completare l'interfaccia utente ho implementato anche una finestra di "Warning" per far notare all'utente che se l'operazione corrente viene continuata potrebbe cambiare in modo drastico la finestra. Per esempio se viene effettuato il caricamento di un file json dopo il caricamento iniziale la lista originale è persa oppure se viene creato un oggetto uguale a uno già presente nel `Qontainer`.

## Elements GUI:

### QComboBoxes

Un'altra scelta progettuale è stata la divisione tra i file della GUI (schermate) e gli elementi usati dalle suddette schermate. Per rendere il codice più leggibile e meno ripetitivo possibile ho creato una cartella (`elementsGUI`) dove sono contenuti tutti i file, i campi dati e gli oggetti necessari alle schermate. In particolare per le schermate `InsLayout` e `SearchDialog` ho creato delle classi che derivano da `QComboBox` che contengono dati da cui è possibile scegliere. Queste classi sono:

- `CaseColorComboBox`: combo box per colore case
- `CaseSizeComboBox`: combo box per grandezza case
- `HddComboBox`: combo box per dimensione hard disk
- `OperativeSystemComboBox`: combo box per sistema operativo
- `MonitorComboBox`: combo box per numero di monitor
- `RamComboBox`: combo box per numero di ram
- `TypeCombox`: combo box per il tipo di computer

### ComputerListItem

Per riempire la `QListWidget` della schermata principale ho creato una class `ComputerListItem` che deriva da `QListWidgetItem`; `ComputerListItem` è caratterizzato da un puntatore a un oggetto `Computer`, ho preso questa scelta in modo da "collegare" un `QListWidgetItem` ad un oggetto `Computer`. Per completare `ComputerListItem` ho introdotto un'altra classe: `WidgetComputerListItem`, usata per definire un `QWidget` che, al momento di inserimento di un oggetto in `QListWidget` verrà aggiunto ad ogni `ComputerListItem`.

### StyleGUI

Ho inoltre realizzato una classe `StyleGUI` che, al suo interno, contiene campi dati statici `QSize`, `QFont`, `QPalette` e `int` in modo da dover modificare solo un file per effettuare cambi di dimensioni, font e colori.

# POLIMORFISMO

Ho utilizzato il polimorfismo per la gerarchia di classi con il metodo `virtual double getTotalPrice() const` che ritorna il prezzo totale di un oggetto `Computer`. Per effettuare il controllo del tipo di un puntatore `Computer` utilizzo la funzione virtuale pura (in `Computer`) che ritorna una stringa con il tipo di oggetto.

# SALVATAGGIO / CARICAMENTO

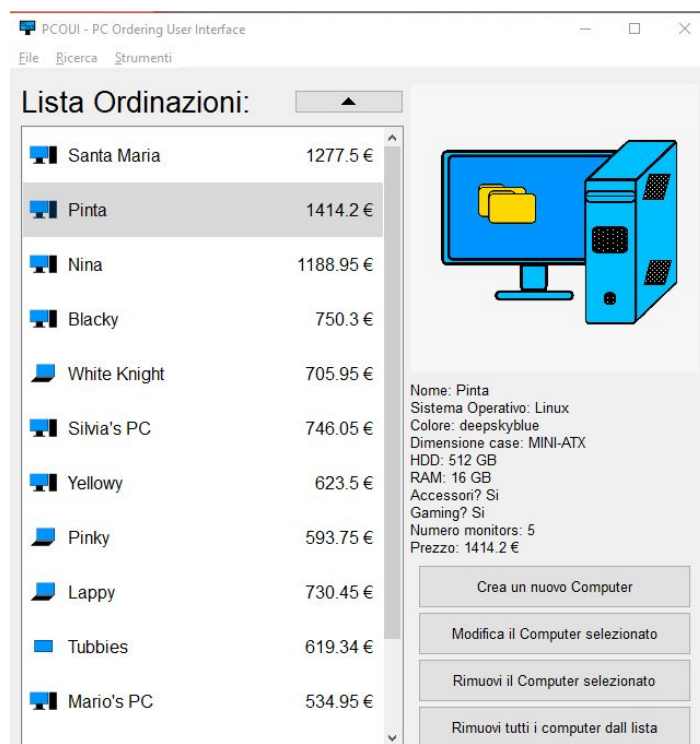
Per il salvataggio di dei dati dell'applicazione ho deciso di optare per `Json` dato il supporto dato dal framework `Qt`. Infatti grazie a: `QJsonArray`, `QJsonDocument`, `QJsonObject`, `QJsonValue` e `QFile` ho realizzato delle funzioni che permettono il salvataggio, sotto forma di un file `Json` partendo da un `Qontainer` di `Computer*`, e la lettura di un file `json` e la costruzione di un `Qontainer` di `Computer*`.

Le quattro funzioni possono essere divise in funzioni di *salvataggio* e *caricamento*:

- Salvataggio:
  - `QJsonObject createJsonObject(Computer *)`
  - `void writeJsonDoc(QJsonArray*)`
  - `void saveJson()`
- Caricamento:
  - `void readJson()`

Il processo di salvataggio inizia con la chiamata: `saveJson()`. Questa funzione itera per il `Qontainer`, e per ogni oggetto `Computer*` viene chiamata la funzione: `createJsonObject(Computer *)`, che ritorna un `QJsonObject` che viene inserito in un `QJsonArray`. Alla fine della iterazione si ha un array di oggetti `Json`. Viene chiamato quindi `writeJsonDoc(QJsonArray*)` che scrive in un file `.json` tutti gli oggetti `Json`.

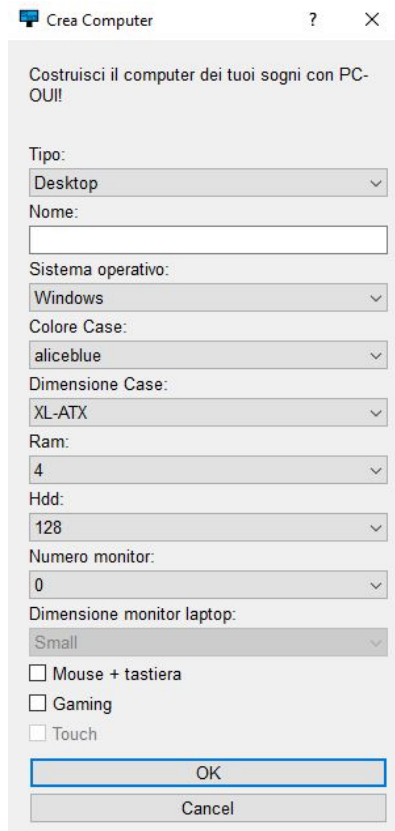
Il processo di caricamento é effettuato solo grazie alla funzione `readJson()` che apre un file `.json`. Esso viene letto e viene estratto dal documento `json` un array di oggetti `Json`. Questo array viene iterato e per ogni oggetto viene creato un oggetto di tipo `Computer*` che viene infine inserito nel `Qontainer<Computer*> container`.



## MANUALE UTENTE BASELAYOUT

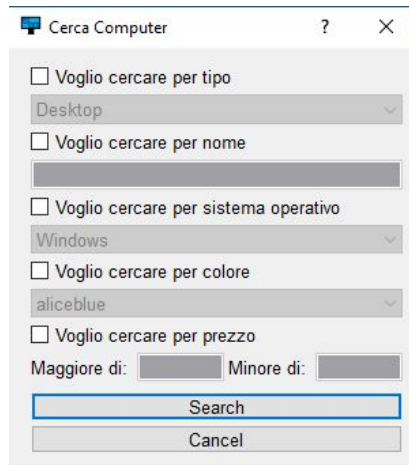
Questa è la schermata principale, il `QListWidget` sulla sinistra rappresenta la lista `Qontainer<Computer*>` all'interno di `Store` che serve a contenere l'ordinazione corrente. Sopra la `QListWidget` è presente un pulsante con una freccia. Esso serve a riordinare la lista per il prezzo dei singoli computer. Se si clicca su oggetto vengono mostrate nel `QLabel` a destra le informazioni dell'oggetto e viene cambiato il colore dell'immagine per rispecchiare il colore del

Computer. In basso a destra ci sono 4 pulsanti che servono all'utente per creare, modificare, rimuovere un computer o tutti.



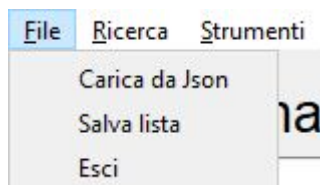
## INSLAYOUT

Questa è la schermata di inserimento di nuovo oggetto o della modifica di un oggetto nella lista. Sono presenti tutti i campi dati necessari alla creazione degli oggetti e i QComboBox e QCheckBox vengono modificati a seconda del tipo di Computer che si sta creando. Dopo aver premuto "OK" verrà inizializzato il processo di creazione o modifica di un oggetto.



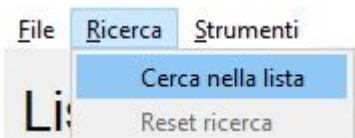
## SEARCHDIALOG

Questa è la schermata di ricerca. Si accede a SearchDialog dal menu "View" in BaseLayout. Dopo aver premuto "Search" verrà mostrato su QListWidget il risultato della ricerca.



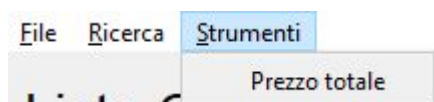
## FILE MENU

Questa menu rende disponibili le funzioni di salvataggio, caricamento e uscita dall'applicazione. La funzione di salvataggio non è disponibile quando la lista è vuota.



## RICERCA MENU

Questo menu rende disponibile la funzione di ricerca. La funzione di "Reset search" viene resa disponibile solo dopo una ricerca.



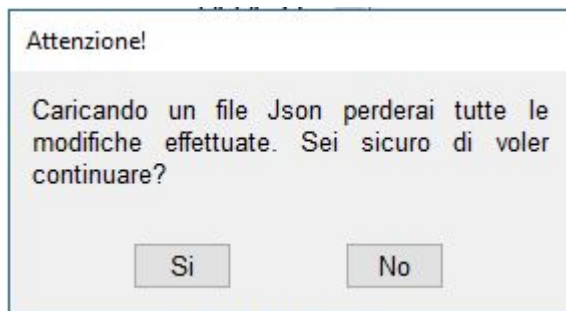
## STRUMENTI MENU

Questo menu rende disponibile la funzione di visualizzare il totale in euro della ordinazione (lista corrente)



## PREZZO TOTALE

Mostra il prezzo totale in euro della ordinazione.



## WARNING DIALOG

### CONTEGGIO ORE:

- Ideazione progetto: 5 ore
- Realizzazione Qontainer: 4 ore
- Realizzazione gerarchia: 3 ore
- Realizzazione GUI: 15 ore
- Realizzazione classe Store (salvataggio/caricamento): 3 ore
- Apprendimento libreria Qt: 8 ore
- Debugging e testing: 12 ore
- Stesura relazione: 2 ore

Totale: 52 ore