Progetto di Programmazione a Oggetti

Agatea Riccardo, matricola 1170718

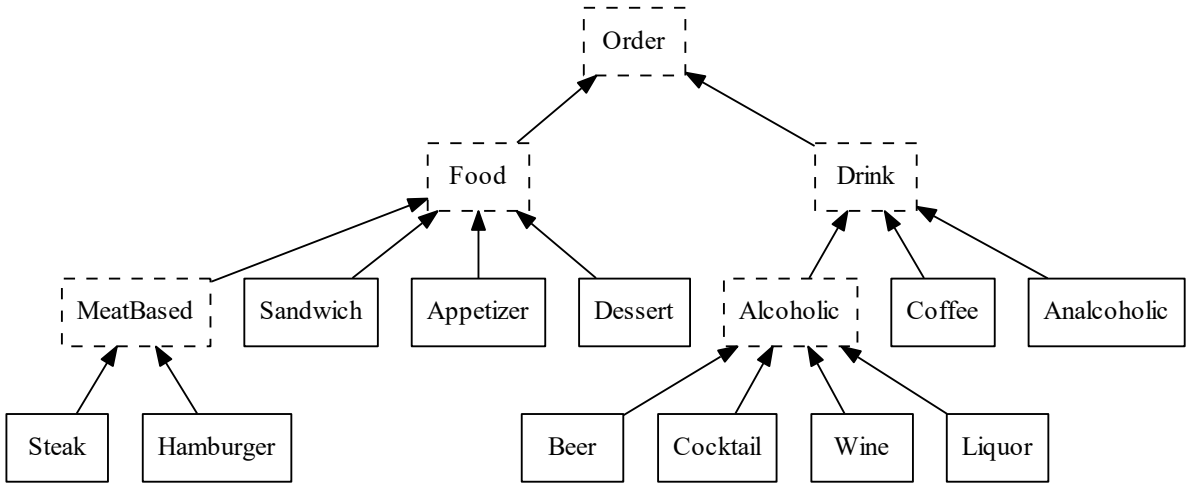
a.a. 2018/2019

1 Introduzione

L'applicazione consiste in un sistema di gestione delle ordinazioni di un ristorante; permette l'immissione di nuove ordinazioni, la loro modifica e rimozione, il loro salvataggio su file e caricamento da file, e la loro ricerca secondo diversi parametri. Inoltre, gestisce la separazione fra ordinazioni in attesa e completate.

2 Descrizione aspetti progettuali

2.1 Gerarchia di tipi



Informazioni Generali La gerarchia modella le ordinazioni inviate alla cucina del ristorante. Radicata nella classe polimorfa astratta `Order`, si dirama in due direzioni, `Food` e `Drink`, associate rispettivamente a ordinazioni di piatti e bevande. Il polimorfismo è dato da alcuni metodi virtuali, descritti in seguito, che forniscono funzionalità di copia e confronto polimorfi, e permettono di ricavare informazioni esplicite sul tipo degli oggetti. La gerarchia è fortemente estensibile sia "in orizzontale", aggiungendo nuovi sottotipi alle classi base presenti, sia "in verticale", fornendo sottotipi alle classi più derivate.

2.1.1 Order

Campi Dati La classe `Order` incapsula informazioni di base relative ad un ordine attraverso tre campi dati privati di istanza:

- `table`, il numero del tavolo,
- `item`, il nome della pietanza,

- `quantity`, il numero di porzioni

È presente un campo dati privato statico, `class_name`, che incapsula il nome della classe sottoforma di stringa. Sono inoltre presenti 4 metodi statici privati che permettono l'accesso a degli oggetti che, concettualmente, dovrebbero essere campi statici, ma per motivi progettuali (spiegati in seguito) sono allocati sullo heap e vengono acceduti attraverso puntatori di classe statica:

- `static std::vector<std::string> &abstracts()`
  Metodo che ritorna un `std::vector` contenente i nomi sottoforma di stringa delle classi astratte derivate da `Order`.
- `static std::vector<std::string> &types()`
  Metodo che ritorna un `std::vector` contenente i nomi sottoforma di stringa delle classi istanziabili derivate da `Order`.
- `static std::multimap<std::string, std::pair<DetailType, std::string>> &info()`
  Metodo che ritorna una `std::multimap` che associa a ciascun tipo derivato da `Order` la forma ed il nome dei dettagli specifici per quel tipo.
- `static std::map<std::string, std::function<DeepPtr<Order>(unsigned int, const std::string &, unsigned int, const std::vector<std::string> &)>> &make()`
  Metodo che ritorna una `std::map` che associa a ciascun tipo derivato da `Order` una funzione che agisce da costruttore, ritornando un puntatore smart ad un nuovo oggetto.

Membri Statici La classe `Order` comprende anche dei membri statici, che forniscono due tipi di funzionalità:

- Forniscono informazioni complessive sulla gerarchia.
- Forniscono informazioni e funzionalità relative ad un tipo a partire da una stringa.

In questo modo è possibile sfruttare una sorta di polimorfismo che non richiede un oggetto di invocazione.

Metodi Per l'interazione con gli oggetti sono disponibili 13 metodi pubblici di istanza (virtuali e non) che comprendono un costruttore ed il distruttore, a cui vanno aggiunti il costruttore di copia e l'operatore di assegnazione di copia standard forniti dal compilatore. I metodi di istanza sono:

- `Order(unsigned int, const std::string &, unsigned int)`
  Costruisce un oggetto inizializzando i campi dati al valore del rispettivo parametro, in ordine.
- `virtual ~Order() = default`
  Distruttore di default, dichiarato virtuale.
- `virtual Order *clone() const = 0`
  Metodo di copia "standard", dichiarato virtuale puro per permettere la costruzione di copia polimorfa.
- `unsigned int getTable() const`
  Getter per il campo dati `table`.
- `std::string getItem() const`
  Getter per il campo dati `item`.
- `unsigned int getQuantity() const`
  Getter per il campo dati `quantity`.
- `void setQuantity(unsigned int)`
  Setter per il campo dati `quantity`.
- `virtual std::string getClassName() const`
  Metodo virtuale che ritorna il nome della classe sottoforma di stringa.
- `virtual bool isA(const std::string &) const`
  Metodo virtuale il cui parametro, `type`, è interpretato come il nome di una classe `Type`, e ritorna true se e solo se il tipo dinamico di `*this` è un sottotipo di `Type`.

- `virtual std::vector<std::string> getDetails() const = 0`
  Metodo virtuale puro che ritorna un elenco di ”dettagli” relativi all'ordine, cioè informazioni aggiuntive specifiche associate al tipo di pietanza ordinata.

- `virtual void setDetails(const std::vector<std::string> &) = 0`
  Metodo virtuale puro speculare a `getDetails()`, permette di modificare le informazioni aggiuntive.

- `virtual bool operator==(const Order &) const`
  Operatore di uguaglianza, dichiarato virtuale per tenere in considerazione dei dettagli relativi alle sottoclassi. L'implementazione ”parziale” fornita da `Order` confronta i rispettivi tipi e i campi dati.

- `bool operator!=(const Order &) const`
  Operatore di disuguaglianza, dichiarato non virtuale in quanto si basa sull'operatore di uguaglianza. Ritorna `true` se e solo se l'operatore di uguaglianza ritorna `false`.

Sono inoltre presenti 4 metodi pubblici statici che agiscono da getter per i ”campi dati statici” allocati sullo heap:

- `static const std::vector<std::string> &getAbstracts();`
  Getter per `abstracts()`.

- `static const std::vector<std::string> &getTypes();`
  Getter per `types()`.

- `static const std::multimap<std::string, std::pair<DetailType, std::string>> &getInfo();`
  Getter per `info()`.

- `static const std::map<std::string, std::function<DeepPtr<Order>(unsigned int, const std::string &, unsigned int, const std::vector<std::string> &)>> &getMake();`
  Getter per `make()`.

Classi Annidate È presente una classe annidata protetta, `Empty`, la quale non ha campi dati, ed ha due costruttori:

- `Empty(const std::string &);`
  Il parametro viene interpretato come il nome di una classe astratta derivata da `Order`, e viene aggiunto a `abstracts()`.

- `Empty(const std::string &, const std::vector<std::pair<DetailType, std::string>> &, const std::function<DeepPtr<Order>(unsigned int, const std::string &, unsigned int, const std::vector<std::string> &)> &);`
  Il primo parametro, `type`, viene interpretato come il nome di un tipo derivato da `Order`, e viene aggiunto a `types()`; il secondo, `details`, come un elenco di coppie (forma, nome) associate a dettagli di una classe derivata, e quindi ogni suo elemento viene agiunto a `info()` utilizzando `type` come chiave; infine, il terzo parametro, `constructor`, viene interpretato come una funzione che agisce da costruttore, e quindi viene aggiunto a `make()` utilizzando ancora `type` come chiave.

È inoltre presente una scoped enumeration, `DetailType`, che rappresenta le varie forme che possono assumere i dettagli aggiunti dalle classi derivate:

- `DetailType::Choice`, una scelta fra due alternative.

- `DetailType::SmallText`, un breve testo (ad esempio una dimensione, o una temperatura).

- `DetailType::LargeText`, un testo più lungo (ad esempio una descrizione o un elenco).

”Polimorfismo su Stringhe” Per assicurare il corretto popolamento di `abstracts()`, `types()`, `info()`, e `make()` è necessario che ciascuna classe derivata (anche indirettamente) da `Order` abbia un campo dati statico, possibilimente privato, di tipo `Empty`, e che esso sia inizializzato con il costruttore adeguato: per le classi astratte quello ad un parametro, a cui va passato il nome della classe, mentre per le classi istanziabili quello a tre parametri, a cui vanno passati il nome della classe, l'elenco dei dettagli, come coppie (forma, nome), e una funzione che agisca da costruttore. In particolare, i dettagli nell'elenco devono essere nello stesso ordine in cui vengono ritornati dal metodo

`getDetails()`. Questo sistema garantisce che i 4 container siano popolati correttamente all'avvio dell'applicazione grazie al fatto che i campi dati statici sono costruiti prima dell'esecuzione della funzione `main()`, ed essendo completamente automatico esonera l'utente della gerarchia dall'invocazione di un'ipotetica funzione `setup()`. Il tutto si basa sulla garanzia che i 4 container siano costruiti prima di essere popolati, ed è per questo che vengono costruiti sullo heap ed acceduti attraverso puntatori di classe statica locali ai metodi di accesso: essendo di classe statica, le inizializzazioni dei puntatori vengono eseguite solo una volta, la prima volta che le funzioni vengono invocate, cioè la prima volta che un campo statico di tipo `Empty` viene costruito. D'altro canto, essendo allocati sullo heap andrebbero deallocati con `delete`, cosa che non avviene, ma siccome andrebbero deallocati dopo l'esecuzione della `main` questo non consiste in un memory leak, perchè in questo momento la memoria viene comunque reclamata dal sistema operativo.

RTTI su stringhe I metodi `getClassName()` e `isA()` forniscono una versione basata su stringhe dei meccanismi di RTTI. Considerando una reference valida `o` di tipo `Order &` (oppure, per un ragionamento analogo, un puntatore valido di tipo `Order *`), l'invocazione `o.getClassName()` ritorna una stringa analoga a quella ritornata da `typeid(o).name()`, con la particolarità che la prima è fissata dal programmatore, mentre la seconda è implementation defined. In modo simile, se `type` è una stringa contenente il nome di una classe `Type`, l'invocazione `o.isA(type)` è analoga a `dynamic_cast<Type *>(&o)!=nullptr`, ma in questo caso la differenza, più marcata, è che `isA()` "agisce" su stringhe, mentre il `dynamic_cast` "agisce" su tipi.

2.1.2 Food

La classe astratta `Food`, derivata da `Order`, rappresenta ordini di piatti. Presenta un campo dati aggiuntivo `without`, dotato di getter e setter, che rappresenta parti del piatto che il cliente ha chiesto di escludere; per tenere in considerazione il nuovo campo dati è presente un costruttore adeguato. Il metodo `clone()`, pur rimanendo virtuale puro, subisce overriding per modificare il tipo di ritorno (che è quindi covariante). Gli altri metodi virtuali sono reimplementati come da aspettative.

2.1.3 MeatBased

La classe astratta `MeatBased`, derivata da `Food`, rappresenta ordini di piatti il cui ingrediente principale è un tipo di carne, e per questo ha un ulteriore campo dati `temperature`, dotato di getter e setter, che rappresenta la temperatura di cottura della carne; è quindi fornito un costruttore adeguato. Il metodo `clone()` subisce ancora override, ma non viene ancora implementato. Gli altri metodi virtuali sono reimplementati come da aspettative.

2.1.4 Steak, Hamburger

Le classi `Steak` e `Hamburger`, derivate da `MeatBased`, rappresentano ordini di piatti il cui ingrediente principale è rispettivamente una bistecca o un hamburger. Non presentano campi dati aggiuntivi, ma `Steak` non fa uso del campo dati `without` di `Food`; di conseguenza, `Hamburger` utilizza il costruttore di `MeatBased`, mentre `Steak` ne fornisce uno ad hoc. Per lo stesso motivo, solo `Steak` reimplementa `getDetails()`, `setDetails()` e l'operatore di uguaglianza. Gli altri metodi virtuali sono reimplementati come da aspettative.

2.1.5 Sandwich

La classe `Sandwich`, derivata da `Food`, rappresenta ordini di piatti basati su sandwich. Non presenta campi dati aggiuntivi, e di conseguenza utilizza il costruttore di `Food`. Per lo stesso motivo, non reimplementa `getDetails()`, `setDetails()` e l'operatore di uguaglianza. Gli altri metodi virtuali sono reimplementati come da aspettative.

2.1.6 Appetizer, Dessert

Le classi `Appetizer` e `Dessert`, derivate da `Food`, rappresentano ordini di antipasti e dolci, rispettivamente. Ciascuna ha un campo dati aggiuntivo, `sauces` e `with` nell'ordine, con getter e setter, ed entrambe non fanno uso del campo dati `without` di `Food`. Per questo forniscono entrambe costruttori, ed entrambe reimplementano `getDetails()`, `setDetails()` e l'operatore di uguaglianza. Gli altri metodi virtuali sono reimplementati come da aspettative.

2.1.7 Drink

La classe astratta `Drink`, derivata da `Order`, rappresenta ordini di bevande; non presenta nessun campo dati aggiuntivo, e quindi utilizza il costruttore di `Order`. Il metodo `clone()`, subisce overriding per modificare il tipo di ritorno ma non viene implementato, come non vengono implementati `getDetails()`, `setDetails()` e l'operatore di uguaglianza. Gli altri metodi virtuali sono reimplementati come da aspettative.

2.1.8 Alcoholic

La classe astratta `Alcoholic`, derivata da `Drink`, rappresenta ordini di bevande alcoliche; non presenta nessun campo dati aggiuntivo, e quindi utilizza il costruttore di `Drink`. Il metodo `clone()`, subisce overriding per modificare il tipo di ritorno ma non viene implementato, come non vengono implementati `getDetails()`, `setDetails()` e l'operatore di uguaglianza. Gli altri metodi virtuali sono reimplementati come da aspettative.

2.1.9 Beer, Cocktail, Wine, Liquor

Le classi `Beer`, `Cocktail`, `Wine`, e `Liquor`, derivate da `Alcoholic`, rappresentano ordini di birre, cocktail, vini, e liquori, rispettivamente. Ciascuna ha un campo dati aggiuntivo, `size`, `garnish`, `vintage` e `ice` nell'ordine, con getter e setter, e per questo forniscono tutte costruttori. I metodi virtuali sono reimplementati come da aspettative.

2.1.10 Coffee, Analcoholic

Le classi `Coffee` e `Analcoholic`, derivate da `Drink`, rappresentano ordini di caffè e bibite analcoliche, rispettivamente. Ciascuna ha un campo dati aggiuntivo, `notes` e `ice` nell'ordine, con getter e setter, e per questo forniscono entrambe costruttori. I metodi virtuali sono reimplementati come da aspettative.

2.2 Template Container

Le istanze del template di classe `Container<T>` sono contenitori che permettono di gestire collezioni di oggetti polimorfi (in caso non lo siano il template è comunque istanziabile, ma è necessario fornire specializzazioni ai template di funzione contenuti nel namespace `UniformInterface`, e il contenitore risultante è inutilmente appesantito). Siccome l'applicazione richiede che si possano eseguire inserimenti e rimozioni in posizione arbitraria nel container, si è scelto in fase di progettazione di utilizzare una lista doppiamente concatenata; si è inoltre scelto di fornire un'interfaccia pubblica il più simile possibile a quella del template `std::list<T>`.

2.2.1 Classi annidate

Essendo una lista concatenata, il template di classe si appoggia su un template di struttura annidata `Node`, che presenta diversi costruttori adatti ai diversi modi di passare gli oggetti da aggiungere al container come parametri, il distruttore, ed un overloading per l'operatore di uguaglianza, che esegue il confronto fra i campi info dei nodi considerati e dei successivi. Sono inoltre forniti due template di classe annidati `iterator<T>` e `const_iterator<T>`; in realtà, questi sono specializzazioni parziali del template di classe annidata `temp_iterator<T,constness>`, il cui parametro non-tipo `constness` è `true` per `const_iterator<T>` e `false` per `iterator<T>`. Questo, combinato con i template `reference<T,constness>` e `pointer<T,constness>` contenuti nel namespace `ReferenceTypes`, permette di evitare definire due classi separate, ma fortemente accoppiate, per gli iteratori e gli iteratori costanti, al costo di non permettere il cast da `iterator<T>` a `const_iterator<T>`. È fornito un costruttore privato ad un parametro, e per questo il template `Container` è dichiarato come template di classe friend associato, mentre il costruttore di default (che costruisce un iteratore non dereferenziabile) è pubblico; inoltre, sono presenti (e pubblici) i costruttori di copia e di move forniti dal compilatore, come anche i rispettivi operatori di assegnazione e il distruttore. Gli iteratori forniscono overloading per gli operatori di dereferenziazione, dereferenziazione e selezione, incremento e decremento prefisso e postfisso, e confronto. Il confronto è superficiale: l'operatore di uguaglianza ritorna `true` se e solo se i due iteratori puntano allo stesso nodo. In caso di comportamenti anomali, i metodi lanciano delle appropriate eccezioni.

2.2.2 Metodi

Costruttori, Distuttore, Assegnazione Il container fornisce 6 costruttori, di cui due sono il costruttore di copia e quello di move, i quali sono accompagnati dal distruttore e dagli operatori di assegnazione di copia e di move per la rule of five, che dallo standard C++11 ha sostituito la rule of three. I rimanenti costruttori permettono

di costruire un container vuoto, un container di una data dimensione con nodi tutti uguali, un container a partire da un range, oppure un container a partire da una initializer list.

Iterazione Per l'iterazione sono forniti 6 metodi, completamente analoghi ai metodi `begin()`, `begin() const`, `cbegin() const`, `end()`, `end() const` e `cend() const` forniti dai container della STL. È inoltre fornito un metodo statico `toConstIter()` per convertire da `iterator<T>` a `const_iterator<T>`.

Dimensione, Accesso, Inserimento, Rimozione Sono forniti un metodo `size()` che ritorna la dimensione della lista, e un metodo `empty()` che ritorna `true` se e solo se la lista è vuota, e metodi `front()` e `back()`, `const` e non, per l'accesso al primo e all'ultimo elemento. Sono forniti metodi `push_back()`, `push_front()`, `pop_back()`, e `pop_front()` per l'inserimento e la rimozione in testa e in coda alla lista (ciascuno dotato di diversi overloading),ed inoltre metodi `insert()` ed `erase()` per l'inserimento e la rimozione in posizione arbitraria, basati su iteratori. È fornito un metodo `clear()` per svuotare il container. Infine, sono forniti metodi `swap()` (in due versioni, per scambiare il contenuto di due container o di due nodi) e `give` (per spostare un nodo da un container ad un altro, o da un punto ad un altro nello stesso container);

Ricerca, Confronto Sono forniti metodi `find()` e `find_if`, `const` e non, per la ricerca di elementi nel container (`find()` utilizza l'operatore di confronto fornito da `DeepPtr<T>`, mentre `find_if()` utilizza una funzione passata come parametro, la quale deve avere due parametri di tipo `const T &` e tipo di ritorno `bool`). Sono forniti inoltre operatori di confronto; l'operatore di uguaglianza ritorna `true` se e solo se i due container hanno gli stessi elementi nello stesso ordine.

2.3 Altre Classi

Oltre alla gerarchia radicata in `Order` e al template `Container`, sono state definite altre classi.

2.3.1 Template DeepPtr

Il template di classe `DeepPtr<T>` è un template per puntatori smart. Gli oggetti delle classi istanziate da questo template utilizzano un campo dati di tipo `T *` per la gestione di oggetti anche polimorfi. La gestione profonda della memoria è garantita grazie a costruttore di copia, costruttore di move, operatore di assegnazione di copia, operatore di assegnazione di move, e distruttore. I rimanenti costruttori generano puntatori smart ad una copia dell'oggetto passato come parametro. Sono forniti operatori di dereferenziazione e di dereferenziazione e selezione, ciascuno in due versioni, `const` e non-`const`, ed operatori di confronto, che eseguono il confronto fra gli oggetti puntati. Sono inoltre forniti i metodi `swap()`, per scambiare i contenuti di due puntatori smart, e `takeResponsibility()`, per associare il puntatore smart ad un oggetto preesistente, invece di costruirne uno di copia. Per la copia e il confronto vengono utilizzati dei template di funzione contenuti nel namespace `UniformInterface`, che nella loro versione di default chiamano un metodo `clone()` e l'operatore di uguaglianza, ma che in caso di tipi privi di questi metodi permettono di essere adattati attraverso la specializzazione.

2.3.2 Eccezioni

Sono state definite le classi `EmptyContainer`, `InvalidFile`, `InvalidIterator`, `NullPrtExcept`, e `UnavailableFile` per sollevare eccezioni. `InvalidFile` e `UnavailableFile` derivano da `std::invalid_argument`, mentre le restanti derivano da `std::logic_error`. Comunque tutte le classi sono parte della gerarchia radicata in `std::exception`.

2.3.3 Finestre di Dialogo

Le classi `AddOrderDialog`, `EditOrderDialog`, e `SearchDialog` forniscono all'utente della GUI la possibilità di inserire informazioni, nello specifico per inserire nuovi ordini, modificare ordini esistenti, e selezionare ordini che rispettano specifiche caratteristiche. Sono tutte derivate da `QDialog`, in modo da rendere modali le finestre associate ai rispettivi oggetti. Tutte sfruttano `Order::info()` per rappresentare i dettagli specifici del tipo selezionato nel modo appropriato. In `AddOrderDialog` e `EditOrderDialog` le informazioni inserite dall'utente sono comunicate all'applicazione attraverso dei metodi essenzialmente analoghi a dei getter, mentre `SearchDialog` sfrutta due getter per comunicare se includere gli ordini in attesa e/o quelli completati, ed un metodo che ritorna un oggetto di tipo `std::function` che racchiude un predicato ad un parametro di tipo `const Order &` e ritorna `true` se e solo se il parametro rispetta le condizioni inserite dall'utente.

2.3.4 OrderWidget

La classe `OrderWidget`, derivata da `QFrame`, permette di rappresentare gli ordini nella GUI. Contiene dei pulsanti che permettono all'utente di modificare l'ordine, completarlo o rimuoverlo, e sfrutta dei segnali per comunicare queste informazioni al resto dell'applicazione. Ogni oggetto di tipo `OrderWidget` contiene un indice (cioè un campo dati di tipo `Model::Index`, descritto in seguito) che punta al corrispettivo ordine. Come le finestre di dialogo, la classe sfrutta `Order::info()` per rappresentare i dettagli specifici dell'ordine nel modo appropriato. Ogni `OrderWidget` mostra un'icona appropriata al tipo dell'ordine associato, ed in caso di estensione della gerarchia è necessario aggiungere le icone necessarie attraverso il resource system di Qt. In particolare, l'immagine relativa al tipo `example` deve essere accessibile attraverso il prefisso `/type` e l'alias `example`: il path completo, indipendentemente dal nome del file, deve essere `:/type/example`.

2.3.5 Model, View, SearchView

Le classi `Model` e `View` costituiscono il fulcro dell'applicazione. `Model` racchiude la logica del programma, gestisce i due container di ordini in attesa e completati, permette di aggiungere ordini, rimuoverli, spostarli da un container all'altro, salvare su file e caricare da file, cercare all'interno dei due container ordini che rispettano certe caratteristiche, e ottenere l'elenco degli ordini incompleti. Espone il tipo `Model::Index`, che coincide con il tipo iteratore di `Container<Order>`, per permettere alla GUI di mantenere un collegamento diretto con gli ordini. `View`, classe derivata da `QMainWindow`, rappresenta la finestra principale dell'applicazione. È dotata di una toolbar per le operazioni eseguibili, ed utilizza una `QScrollArea` per visualizzare l'elenco degli ordini, ciascuno rappresentato da un `OrderWidget`. La classe `SearchView`, derivata da `QDialog` perchè sia modale, permette di visualizzare l'elenco degli ordini risultante da una ricerca. Inoltre, permette di rimuoverli o completarli tutti, chiedendo conferma all'utente attraverso un message box.

2.4 Chiamate polimorfe

La gerarchia radicata in `Order` fornisce tre funzionalità diverse in modo polimorfico:

- Clonazione
- RTTI su stringhe
- Gestione dei dettagli di un ordine

Queste funzionalità vengono sfruttate in parti diverse del progetto:

- La clonazione viene utilizzata dai puntatori smart `DeepPtr`, che si appoggiano sulle funzioni racchiuse nel namespace `UniformInterface`. Le due funzioni clone invocano il metodo `clone()` nella loro implementazione di default.
- Il meccanismo di RTTI su stringhe viene utilizzato dalla classe `SearchDialog` per confrontare un ordine con i parametri della ricerca selezionati dall'utente, in particolare per verificare che l'ordine sia di un tipo selezionato, e che sia di un tipo che è derivato da almeno una delle classi astratte selezionate. Inoltre, la possibilità di ricavare il tipo di un oggetto in forma di stringa è utilizzata dalla classe `Model` per il salvataggio dei dati su file e dalla classe `OrderWidget` per mostrare il tipo delle ordinazioni nella GUI.
- La gestione dei dettagli viene utilizzata:
  - "In lettura" (invocando `getDetails()`) da `Model` per il salvataggio dei dati, da `OrderWidget` per mostrare le informazioni nella GUI, e da `SearchDialog` per verificare che ciascun dettaglio di un ordine contenga i parametri di ricerca richiesti dall'utente.
  - "In scrittura" (invocando `setDetails()`) da `OrderWidget` per sovrascrivere i dettagli in seguito ad una richiesta di modifica da parte dell'utente.

2.5 Formato dei file di salvataggio e caricamento

Formato I file di salvataggio sono in formato XML 1.0. All'interno dell'elemento `root` sono presenti tre figli:

- `valid_save`, elemento vuoto con un attributo `application`, il cui valore è `Qontainer`, che specifica la validità del file come file di salvataggio per l'applicazione.

- `to_do`, elemento che contiene come figli tutti gli ordini non ancora completati.
- `completed`, elemento che contiene come figli tutti gli ordini completati.

I figli di `to_do` e `completed` sono elementi `order`, ciascuno con attributi `type`, corrispondente al tipo dell'oggetto che rappresenta l'ordine all'interno dell'applicazione, e `table`, `item` e `quantity`, corrispondenti ciascuno all'omonimo campo dati della classe `Order`. Ogni elemento `order` ha inoltre un elenco di figli, i quali sono elementi vuoti `detail` e rappresentano un dettaglio aggiunto da una sottoclasse di `Order`, il cui valore è contenuto nell'attributo `value`.

Implementazione Per implementare il salvataggio e il caricamento dei dati su e da file si sono utilizzate le classi fornite dalla libreria Qt: `QSaveFile` e `QFile` per la gestione dei file e `QXmlStreamWriter` e `QXmlStreamReader` per la scrittura e lettura del codice XML.

3 Note tecniche

3.1 Istruzioni di compilazione

Per la generazione automatica del Makefile con qmake è fornito il file `progetto.pro`, in quanto il progetto sfrutta ampiamente concetti introdotti dallo standard C++11.

3.2 Ambiente di sviluppo

- Sistema operativo di sviluppo: Windows 10 Home 64-bit
- Compilatore: MinGW-W64 g++ 5.3.0
- Qt framework: Qt 5.12.0
- IDE di sviluppo: Qt Creator 4.8.1

3.3 Ripartizione ore

- Analisi preliminare del problema: 1
- Progettazione modello: 5
  - Progettazione template di classe `Container`: 2.5
  - Progettazione gerarchia di classi: 2.5
- Progettazione GUI: 1.5
- Apprendimento libreria Qt: 2.5
- Codifica modello: 22
  - Codifica template di classe `Container`: 10.5
  - Codifica gerarchia di classi: 11.5
- Codifica GUI: 20
- Debugging: 2
- Testing: 1

Nota. Le ore di apprendimento della libreria Qt si riferiscono al tempo speso per imparare ad utilizzare i sistemi e meccanismi propri della libreria (nello specifico la parent/child relationship, il meccanismo di slot e signal, il project file, e il resource system). Il tempo speso per prendere familiarità con le classi che la libreria fornisce per costruire la GUI è stato considerato come parte del tempo di codifica della stessa, in quanto difficilmente separabile da esso, ma nel complesso è non inferiore alle 5 ore.