Dentist management system

Ingegneria del software

Leonardo Pampaloni, Filippo di Martino

Computer Engineering
University Of Florence
Italy

1 Introduzione

L'idea del progetto nasce come applicazione per la gestione di interventi fatti da dentisti e assistenti su vari clienti. Nell'applicazione sono presenti principalmente:

1. ***Dentista***: Il dentista ha la possibilità di controllare tutte le informazioni riguardanti i clienti, gli assistenti e i set di interventi assegnati agli Assistenti.
2. ***Assistenti***: Ogni assistente ha a sua disposizione un set predefinito di interventi da poter fare, oltre che poter vedere le informazioni sui clienti.
3. ***Clienti***
4. ***Interventi***

L'applicazione ha come obiettivo quello di gestire e notificare l'admin, ovvero il dentista, ti tutti gli interventi effettuati dagli assistenti, inoltre è compresa nell'applicazione il salvataggio dei clienti, degli articoli e degli interventi all'interno di un database. Sono presenti due tipologie di notifica per l'admin, uno all'interno dell'applicazione, che tiene traccia dello storico delle operazioni fatte da tutti gli assistenti mentre l'altro è esterno all'applicazione e viene notificato l'admin tramite email ogni volta che un assistente effettua un intervento su un cliente.

2 Diagrammi UML

Abbiamo scelto di presentare tre diversi diagrammi UML, il diagramma delle classi (*Class Diagram*), il diagramma dei casi d'uso (*Use Case Diagram*), e il diagramma E/R (*Entity Relationship* del progetto.

2.1 Class Diagram

Dal *Class Diagram* possiamo vedere come effettivamente sono legate le varie classi del programma. Si può notare infatti che le operazioni di notifica sono effettuate da un *Observer pattern*, suddiviso in due diverse classi per le due tipologie differenti di notifica. Sono presenti anche altri due pattern: lo *State pattern* e il *Composite pattern*, rispettivamente per la gestione delle classi dei vari menù e per la gestione delle classi per le operazioni/interventi.
Il Dentista (*Admin*,ha il compito di creare Clienti, Articoli, e set di Operazioni/Interventi. Per le operazioni abbiamo incluso la possibilità di raccogliere più interventi in uno (creandolo tramite il *Composite pattern*), oltre ovviamente a poter selezionare l'uso di più articoli e strumenti (E.g. Creazione di un kit-monouso (Guanti,Bicchiere,Tovaglietta), e includerlo in tutte le operazioni).
La classe *Program* è il cuore della gestione dell'applicazione: rappresenta il

nodo centrale del sistema. *Program* è una classe ***Singleone*** in quanto si necessita di avere una singola istanza di essa e deve essere necessariamente reperibile, al suo interno sono presenti inoltre metodi come *load(Connection c* e *upload(Connection c)* che permettono la comunicazione con il DB esterno e *run()* per poter gestire il loop di sistema.

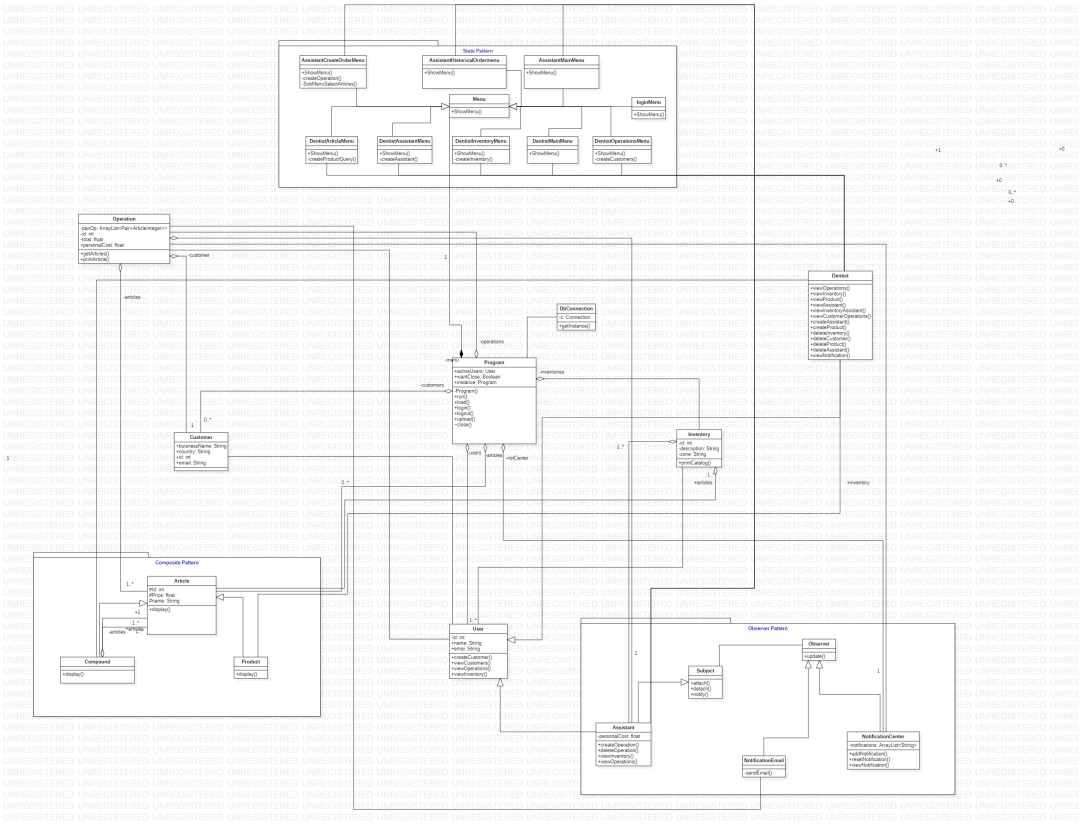

Figure 1: Diagramma delle classi

2.1.1 State Pattern

Si tratta di un pattern comportamentale basato su oggetti che viene utilizzato quando il comportamento di un oggetto deve cambiare in base al suo stato. Questo pattern è spesso utilizzato per le macchine a stati finiti, il nostro caso è molto simile a quello scenario, infatti il menù passa da uno stato all'altro in base alla scelta dell'utente che lo sta utilizzando.

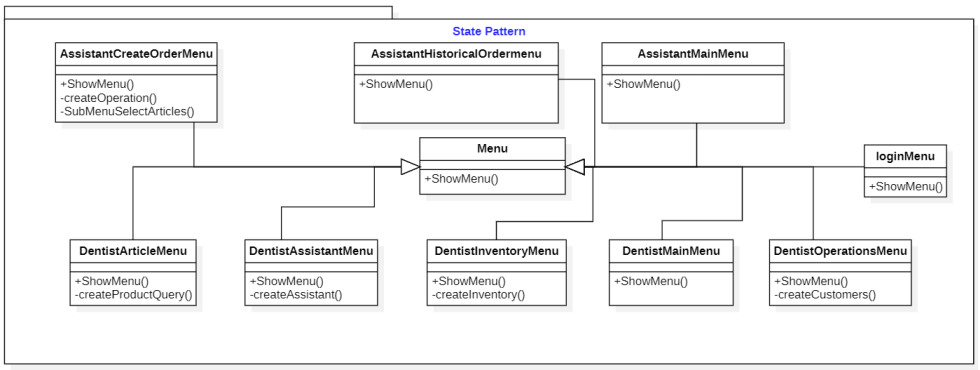

Figure 2: Diagramma delle classi (State pattern)

2.1.2 Observer Pattern

Questo pattern permette di definire una dipendenza 1⟶N fra oggetti, il suo compito è quello di notificare gli N oggetti ogni volta che un oggetto (Subject) cambia stato. Nel progetto sono inseriti due tipologie di Observer: uno che notifica internamente all'applicazione (*NotificationCenter*) mentre l'altro che manda una mail all'assistente desiderato e all'Admin (*NotificationEmail*).

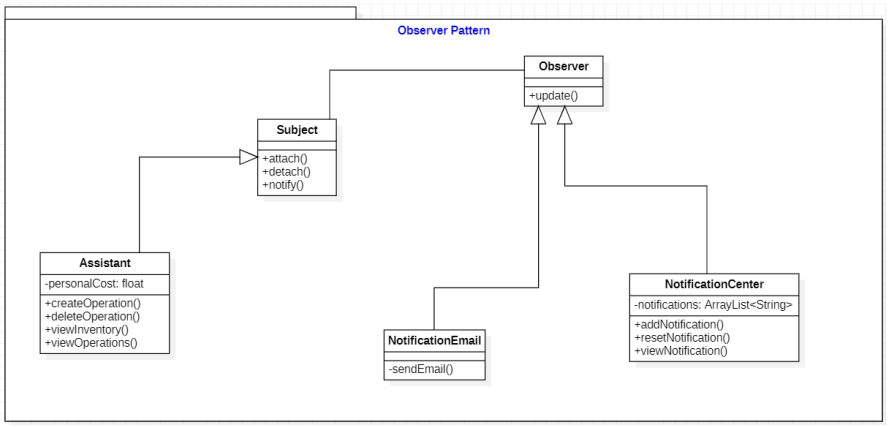

Figure 3: Diagramma delle classi (Observer pattern)

2.1.3 Composite Pattern

Il pattern serve per poter trattare un gruppo di oggetti come istanza di un oggetto singolo. Solitamente questo raggruppamento si può vedere come una struttura ad albero, nel progetto però il pattern è stato leggermente modificato per permettere l'annidamento delle classi composte, in questo caso infatti il grafico del pattern potrebbe essere riassunto con un grafo anzichè un albero.

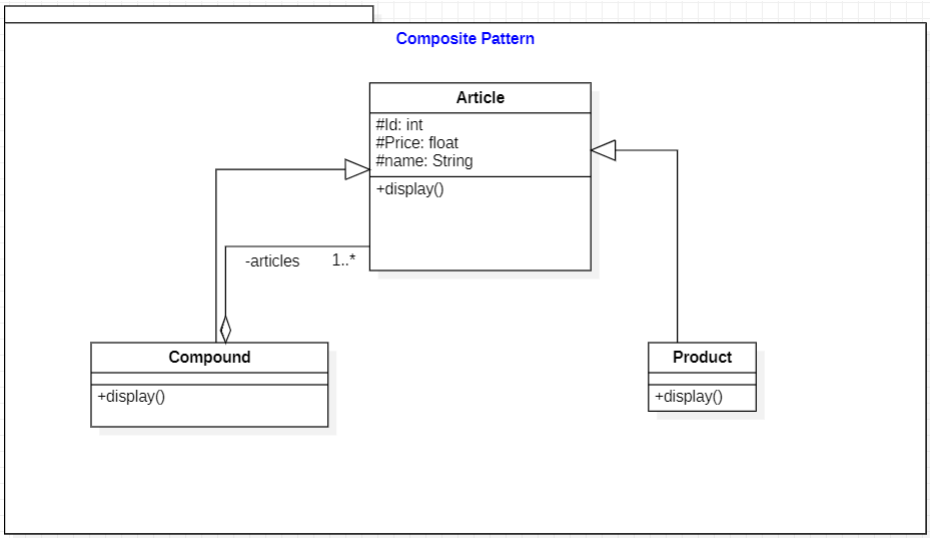

Figure 4: Diagramma delle classi (Composite pattern)

2.2 Use Case Diagram

Nello *Use Case Diagram* sono presenti due attori che interagiscono con il sistema, il Dentista e l'Assistente. I due attori hanno un caso d'uso comune, in quanto entrambi sono classi derivate di User.
Di seguito lo Use Case completo del progetto.

2.2.1 Dentist's Use Case

Si distinguono quattro macro gruppi di casi d'uso:

1. ***Articoli***: Il Dentista ha la possibilità di gestire gli articoli, può infatti scegliere se creare, eliminare o semplicemente visualizzare gli Articoli.

2. ***Inventari***: Solo il Dentista ha la possibilità di aggiungere e rimuovere eventuali Inventari contenenti i Set di Operazioni assegnati ai vari assistenti.

3. ***Assistenti***: Il Dentista ha la possibilità di gestire anchegli Assistenti, può infatti scegliere se aggiungere, eliminare o visualizzare gli Inveentari dei vari Assistenti.

4. ***Clienti***: Il Dentista ha la possibilità di gestire i Clienti, può infatti creare, eliminare o visualizzare lo storico delle operazioni avvenute su quel Cliente.

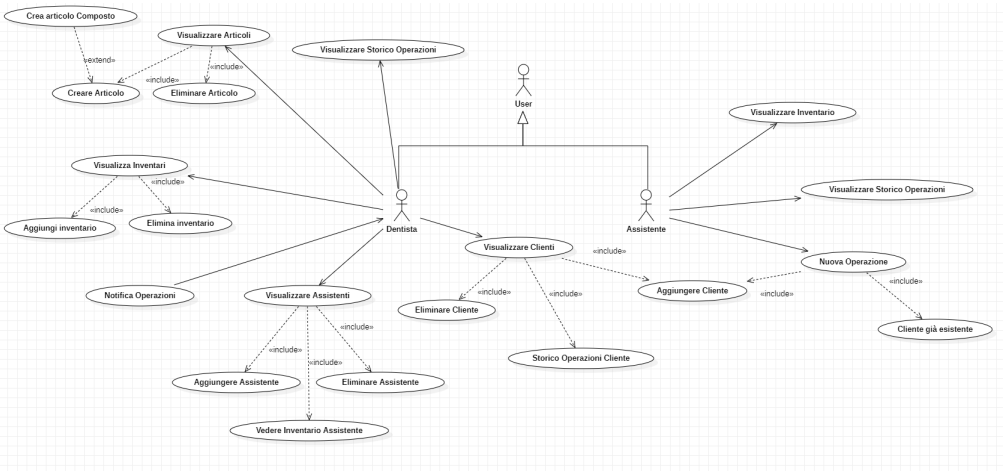

Figure 5: Diagramma dei casi d'uso

2.2.2 Assistent's Use Case

Nello *Use Case Diagram* dell'Assistente invece sono presenti meno funzionalità:

1. ***Operazioni***: Tramite la creazione di una nuova operazione è possibile creare o selezionare il cliente alla quale verrà fatto l'intervento. Inoltre è possibile vedere lo storico delle operazioni fatte dall'Assistente stesso.

2. ***Inventario***: L'Assistente può vedere quali operazioni ha nel suo inventario.

2.3 E/R Diagram

Per avere più chiarezza su come è strutturato il DB abbiamo fatto il diagramma *Entity Relationship* della struttura dati.

2.3.1 Compound DB Structure

3 Implementazione e approfondimento

Di seguito riportiamo alcuni dei più importanti metodi utilizzati che necessitano di una spiegazione più approfondita.

3.1 Observer

```
1 public final class NotificationCenter implements Observer {
2
3     private ArrayList<String> notification;
```

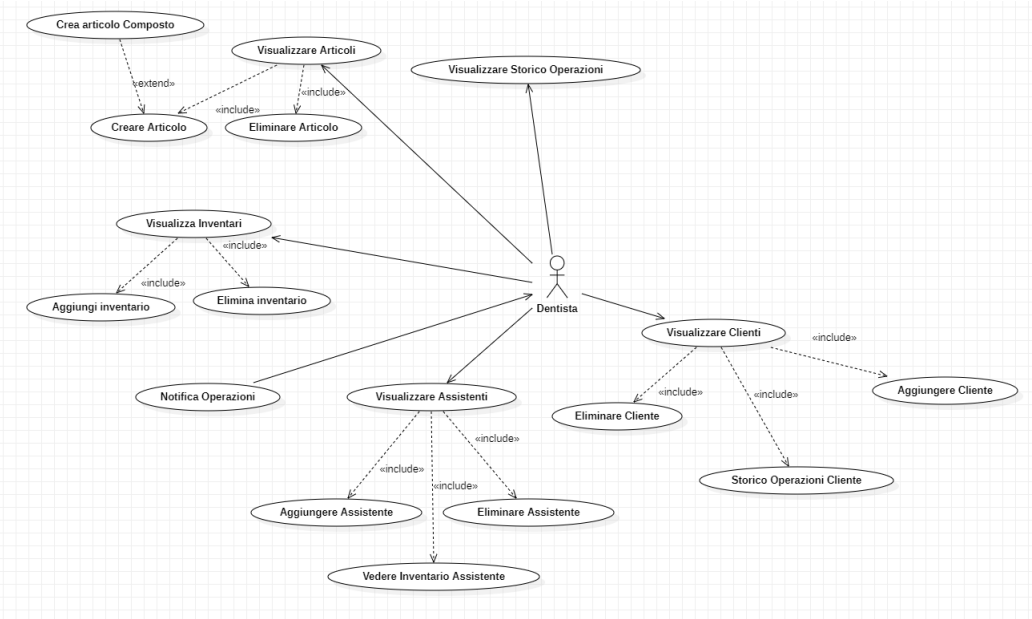

Figure 6: Diagramma dei casi d'uso del Dentista

```
4
5   @Override
6       public void update(Object obj) {
7           Operation operation = (Operation)obj;
8           this. notification .add("Una nuova operazione per " +
                operation.getCustomer().getBusinessName() + "  stata fatta da
                " + operation.getAssistant().getName());
9       }
10
11  }
```

Listing 1: Implementazione di Load

3.2 User

3.3 Dentist

3.4 Operation

3.5 Program

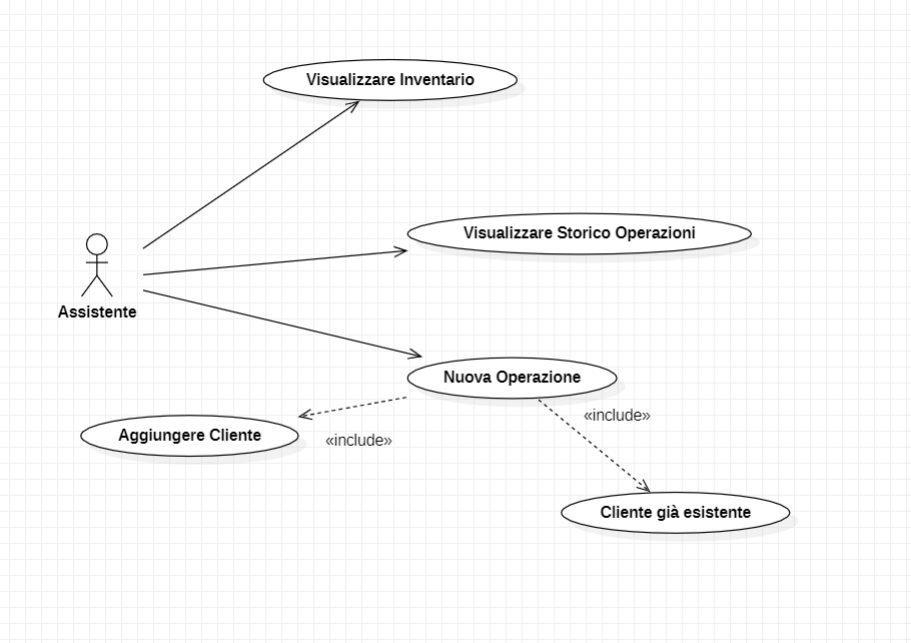

Figure 7: Diagramma dei casi d'uso dell'Assistente

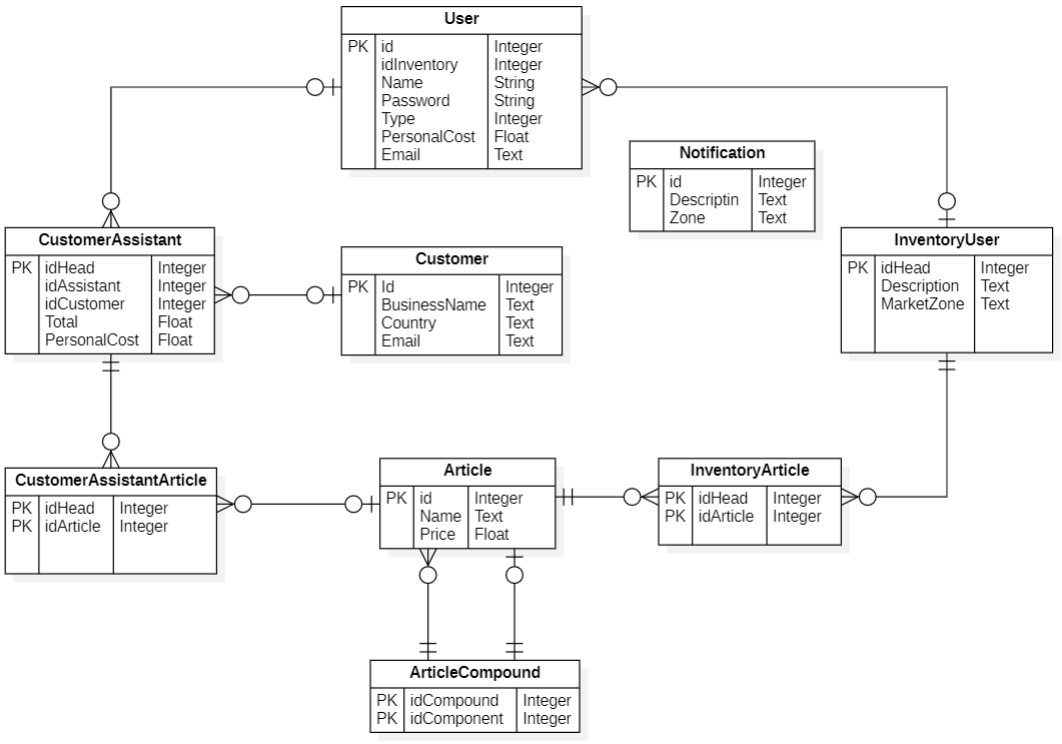

Figure 8: Diagramma E/R della struttura dati

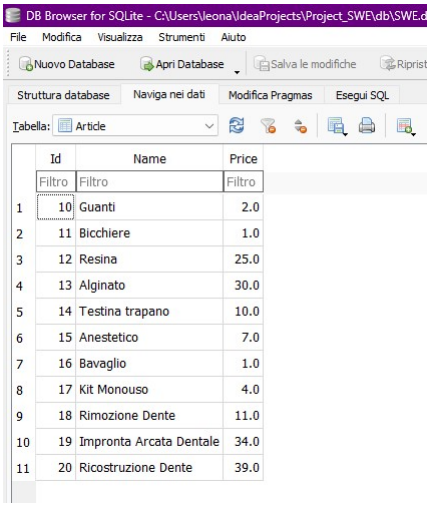

Figure 9: Struttura Article

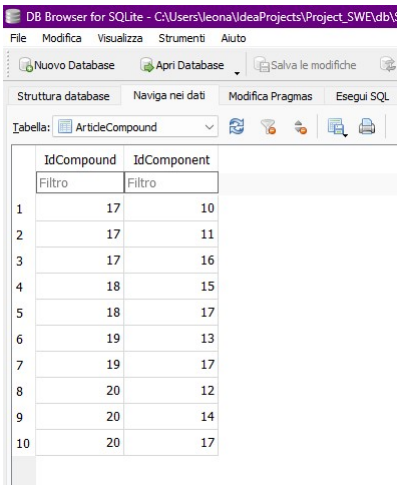

Figure 10: Struttura Article Compound