Università degli Studi dell'Aquila

Dipartimento di Ingegneria e Scienze dell'Informazione e Matematica

Corso di laurea Triennale in Ingegneria dell'informazione

PROGETTAZIONE DI SISTEMI DI CASSA AUTOMATICI CON TRATTAZIONE E GESTIONE DEL DENARO ORIENTATI AL MERCATO DEL RETAIL: GESTIONE TAVOLI

Relatore	**Studente**
Prof. Vincenzo Stornelli	Luca Del Signore
Correlatore	**Matricola**
Ing. Daniele Vettori	252362

A.A. 2019/2020

Indice

Capitolo 1: Introduzione

Questo elaborato è incentrato sui processi decisionali e di sviluppo che hanno portato alla realizzazione di una parte dell'applicazione del progetto "GesMenu", congiuntamente con lo studente Alessio Perozzi, per l'azienda Data Project Engineering (DPE).

Per arrivare a creare quest'applicazione abbiamo dovuto percorrere varie fasi: abbiamo iniziato con lo studio dei Framework necessari, del pattern architetturale da utilizzare fino ad arrivare alla progettazione del sistema e alla sua effettiva realizzazione.

Il lavoro di tesi è stato organizzato nel seguente modo:

- Nel presente capitolo, che è quello di introduzione alla problematica esaminata, si vogliono offrire alcune informazioni generali sullo scopo del lavoro e sull'applicazione GesMenu.

- Il secondo capitolo presenta una spiegazione dettagliata dei framework e pattern utilizzati, illustrando le loro caratteristiche principali e il vantaggio del loro utilizzo.

- Il terzo capitolo costituisce l'essenza di tutto il lavoro in quanto illustra come sono stati utilizzati gli strumenti analizzati nel capitolo precedente per realizzare il software e i ragionamenti che hanno caratterizzato le varie fasi della progettazione.

- La conclusione contiene alcune considerazioni finali sull'elaborato e sulla previsione degli sviluppi futuri. Inoltre, in questa sezione,

vengono evidenziati i limiti che sono emersi nell'utilizzo di queste tecnologie.

1.1 La Problematica

L'Azienda aveva avviato, nel 2017, il suddetto progetto denominato "GesMenù".

Il progetto era scaturito in un primo momento nella realizzazione di un software WEB per l'automazione dei pagamenti attraverso l'uso di casse automatiche; i vantaggi derivanti dall'utilizzo del predetto sistema sono risultati i seguenti:

- Impedire il maneggio del denaro da parte del personale e quindi evitare prelievi fraudolenti (poiché è la cassa automatica che raccoglie il pagamento e dà il resto e non chiude la transazione finché non viene inserito tutto l'importo);
- Individuazione immediata di banconote false;
- Operazione di pagamento molto più rapida e veloce (elimina code);
- Assenza di errori che possono verificarsi durante la quadratura manuale della cassa.

Inoltre, nel 2017, in Italia non esistevano ancora Software finalizzati all'automazione dei pagamenti.

Nel corso del tempo il progetto ha iniziato ad ampliarsi ed evolversi: si è arrivati quindi ad un software in grado di gestire la maggior parte delle attività del mercato del retail (gestione magazzino, gestione di eventuali tavoli etc.). È nata quindi la necessità di ristrutturare e ampliare l'App, in modo tale da includere in modo strutturato le nuove idee. In particolare, a me e al mio collega è stato assegnato rispettivamente il compito di

implementare le funzionalità di gestione dei tavoli e dei conti. La mia parte ha preso il nome di “Modulo di gestione tavoli” e in questo elaborato si andrà ad approfondire tutto il processo di sviluppo, tecnologie e strumenti utilizzati per la realizzazione del software.

1.2 Obiettivi

1.2.1 Obiettivi aziendali

Durante questi 3 anni di evoluzione e di raccolta delle informazioni, si è sentita sempre di più la necessità di **standardizzare** le varie funzionalità che, per venire incontro alle esigenze dei clienti e attività commerciali con esigenze maggiori, si erano venute a creare e a mescolarsi nel software; quindi si è pensato di suddividere le funzionalità del software in “moduli” diversificati usufruibili da parte di tutte le categorie di esercenti del mercato del retail.

Per fare un esempio, un tabaccaio per lo svolgimento della sua attività, acquistando l’App, non avrebbe mai fatto ricorso al modulo relativo alla gestione dei tavoli (per ovvie ragioni) e quindi non lo avrebbe comprato. Al contrario un ristoratore si sarebbe sicuramente interessato all’utilizzo di tale modulo, aggiungendolo all’acquisto.

1.2.2 Obiettivi personali

Il mio obiettivo personale è stato quello di imparare a sviluppare App in un contesto aziendale, approfondendo le conoscenze di programmazione strutturata e ampliando le mie personali conoscenze mentre facevo esperienza nel mondo del lavoro. In aggiunta a ciò, ho maturato l’idea che, trattandosi di un lavoro che doveva essere utilizzato da terzi, era necessario rendere tutto il più chiaro e fruibile possibile,

cercando di dare enfasi all’organizzazione e alla struttura pulita piuttosto che alla funzionalità.

Proprio per questo, ho orientato il mio “modus operandi” a:

1. Identificare la problematica;
2. Scinderla in sotto-problematiche;
3. Creare, per ognuna di esse, una funzione atomica il più generale possibile per permettere il suo riutilizzo in più parti del codice (principio della massima coesione e minimo accoppiamento).

In particolare ho dato molta più enfasi al primo e al secondo punto, poiché il codice è la traduzione e il riflesso del ragionamento fatto in precedenza; esplicito inoltre che ho usato un approccio orientato agli oggetti.

Capitolo 2: Analisi delle tecnologie utilizzate

Nel seguente capitolo si illustreranno e analizzeranno le varie tecnologie e Framework utilizzati per lo sviluppo dell'applicazione, evidenziando le loro caratteristiche, i vantaggi derivanti dal loro utilizzo e il fine sotteso ad esse.

2.1 Visual Studio 2019

È stato scelto per lo sviluppo l'ambiente Visual Studio 2019; questo IDE mette a disposizione dello sviluppatore tantissimi strumenti: vi è la possibilità di scaricare direttamente pacchetti aggiuntivi (NuGet) che integrano nuovi controlli visivi, Framework, estensioni e molto altro per facilitare il processo di sviluppo. Si precisa inoltre che i linguaggi consigliati per lo sviluppo di App in UWP sono:

- XAML per la parte front-end;
- C# per la parte back-end.

Nonostante io non fossi a conoscenza di questi linguaggi e la possibilità offertaci di poterne scegliere alternativi, io e il mio collega abbiamo deciso di utilizzare quelli consigliati per allargare la nostra conoscenza e anche perché era quello usato dal resto del team.

2.2 Universal Windows Platform

Visual studio permette di creare Apps in Universal Windows Platform (UWP): quest'ultima è una piattaforma software il cui scopo principale è quello di sviluppare applicazioni che possono essere eseguite su qualsiasi dispositivo che abbia Windows come sistema operativo

(Windows 10, Windows 10 Mobile, Xbox One e Hololens) senza dover riadattare il software.

I vantaggi principali che si hanno sviluppando in UWP sono:

- **Sicurezza:** le applicazioni UWP mettono in chiaro tutte le risorse di cui avranno bisogno (microfono, Webcam, files etc..); inoltre all'utente viene richiesto se è possibile accedere ad esse per garantire il corretto funzionamento dell'applicazione.
- **Un insieme di API comune a tutti i device:** Ogni dispositivo che è in grado di far girare Windows 10, può utilizzare le API core di UWP; è proprio grazie a questa caratteristica che le App create in UWP sono così versatili e riutilizzabili. È comunque possibile usare API specifiche di un device (ad esempio IoT) grazie all'utilizzo di estensioni SDK, limitando però la fruibilità di quell'App solo da quel particolare dispositivo.
- **Capacità di adattamento dell'interfaccia grafica:** le User-Interface (UI) sono in grado di adattarsi a qualsiasi grandezza e risoluzione dello schermo e *DPI*[1] del dispositivo su cui è in funzione l'App, e di conseguenza si avrà automaticamente (per quanto possibile) l'impostazione del layout e della scala; esistono comunque dei tools che permettono di differenziare l'UI nel caso di utilizzo di un dispositivo specifico.
- **Le App sono in grado di ricevere gli input da diverse fonti,** in virtù della possibilità del loro utilizzo su dispositivi di varia natura (da una tastiera, da uno schermo touch o da un controller dell'Xbox).

[1] Con la sigla **DPI** si esprime la quantità di punti stampati o visualizzati su una linea lunga un pollice.

- **Unico Store per tutti i device**; inoltre è facile e versatile pubblicare e monetizzare le Apps nel suddetto Store.
- **Possibilità di utilizzare diversi linguaggi per lo sviluppo**; tra essi troviamo: XAML, HTML, JavaScript, C#, VB, C++.
- **Apps facilmente esportabili per IOS e Android.**

2.2.1 Perché si è scelto Universal Windows Platform

Visto che il Software in questione è destinato a diversi dispositivi (totem, tablet, dispositivi di cassa) appare chiaro come sia molto vantaggioso svilupparlo in ambiente UWP; in aggiunta lo Store di Windows permette di aggiornare l'App ad ogni avvio, così da rendere completamente autonomi i clienti sotto questo punto di vista.

Inoltre, essendo una desktop App, si ha del codice già compilato (e non interpretato) e quindi molto più veloce, diversamente dalla vecchia versione WEB.

2.3 Il Pattern Model-View-ViewModel

Model-View-ViewModel (MVVM) è un pattern architetturale che permette di disaccoppiare la progettazione dell'interfaccia grafica dalla logica di business. In poche parole MVVM non è molto lontano dai vari pattern già conosciuti (come MVC) ma presenta delle piccole differenze essendo specifico per lo sviluppo di applicazioni Windows. Prima di analizzare nel dettaglio MVVM, dedicherò un paragrafo ad approfondire la creazione delle User-Interface in UWP.

2.3.1 L'interfaccia grafica e il "Code-Behind"

Le Universal Windows Apps usano lo XAML (eXtensible Application Markup Language) per generare le interfacce. Lo XAML è un linguaggio di Mark-Up basato su *XML*[2] e quindi molto simile all'HTML. Segni di questa derivazione sono i "tag" (anche detti controlli) impilati secondo una gerarchia ad albero, con i loro attributi che ci permettono di specificarne, ad esempio, la posizione e la grandezza.

Visual Studio ha inoltre un editor grafico per mezzo del quale, invece di scrivere direttamente XAML, è possibile prendere gli oggetti visivi da una lista, trascinarli nell'anteprima generando automaticamente lo XAML corrispondente alle configurazioni richieste, con la possibilità inoltre di fare modifiche direttamente sull'oggetto (es. modificarne la grandezza) o in menù dove vi sono tutti gli strumenti per effettuare le varie trasformazioni (ad esempio, cambiarne il colore o il testo all'interno). Però, durante lo sviluppo, è risultato molto più facile e maneggevole scrivere a mano piuttosto che usare l'editor.

A supporto della pagina grafica, che ad esempio possiamo denominare "TablesPage.xaml", ve ne è un'altra scritta in C# chiamata "TablesPage.xaml.cs" che, come suggerisce il nome, è una vera e propria classe collegata a quella grafica; all'interno di questa classe, denominata dagli gli sviluppatori "Code-behind", è possibile definire proprietà, funzioni e gestori degli eventi a supporto della pagina grafica.

[2] **XML** (acronimo di **eXtensible Markup Language**) è un metalinguaggio per definire la struttura di documenti e dati.

I Gestori degli Eventi sono funzioni che prendono in ingresso due parametri: l'elemento della UI che ha generato l'evento e le informazioni riguardanti l'evento stesso.

Si intuisce subito che a livello funzionale basterebbe soltanto il Code-behind, ma ciò andrebbe contro tutti i principi di programmazione strutturata poiché, man mano che il software cresce, sempre più problemi di manutenzione e chiarezza nascono, con un conseguente forte accoppiamento tra UI e logica di business e un inevitabile aumento nei costi di aggiornamento.

Ed è per evitare questi problemi che il pattern MVVM è stato pensato, poiché, mantenere una chiara separazione tra la logica dell'applicazione e la sua UI rende più facile testarla, evolverla e aggiornarla. Inoltre, MVVM dà la possibilità di riutilizzare il codice e permette agli sviluppatori e designers delle interfacce di collaborare più agilmente.

L'unico compito che rimane al Code-behind è quello di gestire le funzioni di manipolazione grafica e di navigare tra le varie UI. Inoltre è qui che vengono dichiarate le ViewModel contenenti le informazioni e le funzioni per la realizzazione dei Casi D'Uso esposte come proprietà della pagina stessa, e attraverso il *data-binding*[3] è possibile accedere ad esse ed esporle in output.

[3] Il **data-binding** è quel meccanismo che consente di associare e sincronizzare una fonte dati agli elementi dell'interfaccia utente.

2.3.2 Caratteristiche e vantaggi di Model-View-ViewModel

Come il nome suggerisce ci sono tre Layer principali in MVVM:

- il View Layer che contiene la UI e la sua logica;
- il ViewModel Layer che descrive la presentazione dei dati e lo stato dell'applicazione;
- il Model Layer che incapsula la logica di business e i dati stessi.

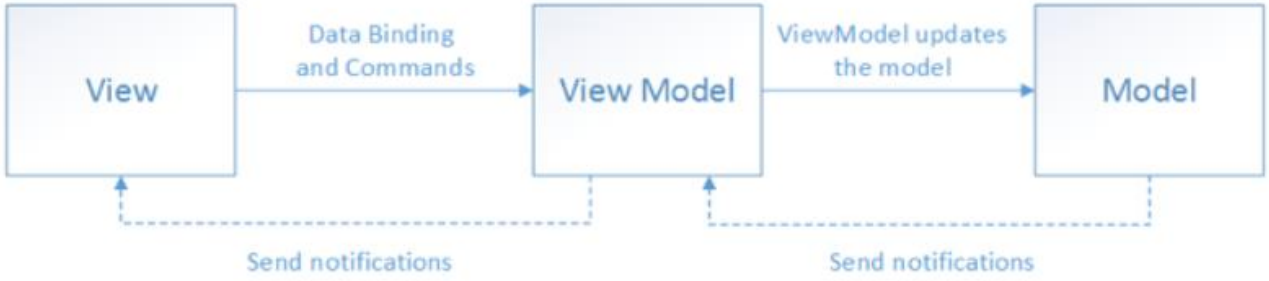

Figura 2.1: Presentazione grafica del pattern MVVM

Analizzando la figura di sopra, notiamo che il Layer View è a conoscenza del ViewModel Layer ma non viceversa, e la stessa relazione esiste tra il ViewModel Layer e il Model Layer.

I benefici nello strutturare l'applicazione in questo modo sono:

- Processo di sviluppo iterativo e incrementale tipico della programmazione strutturata ad oggetti, che non solo permette ai designer delle interfacce e sviluppatori software di lavorare indipendentemente e contemporaneamente, ma anche che questi ultimi possano lavorare su parti diverse del codice e in seguito congiungersi facilmente; infatti non è stato necessario fare alcuna modifica al codice quando il mio modulo ha comunicato con quello progettato dal mio collega.

- Come si intuisce dalla proposizione appena detta, è possibile progettare altre interfacce dell'App senza dover effettuare modifiche agli altri due Layer.
- Nel caso in cui siano già state implementate nell'applicazione le classi del Model e ci siano delle modifiche da effettuare su di esse, potrebbe essere rischioso e complesso per l'integrità di tutta l'applicazione farle direttamente sul Model. In questo scenario, le ViewModel potrebbero assumere il ruolo di adattatore in modo da integrare le modifiche richieste senza modificare il Model.

Di seguito illustrerò nel dettaglio le responsabilità di ciascun Layer:

- Il **View** Layer è responsabile della definizione della struttura, layout e trasposizione di ciò che l'utente andrà a vedere sullo schermo; come già detto prima, esso prende le informazioni da mostrare in output dalle ViewModel attraverso il data-binding.
- Il **ViewModel** Layer gestisce e implementa le varie funzionalità dell'applicazione, decidendo quali dati devono essere mandati in output (lasciando alle View la responsabilità della loro rappresentazione); riceve dalle View i dati in input per elaborarli e integra proprietà aggiuntive (strumenti) che non sono presenti nel Model; implementa proprietà e comandi alle quali le View possono bindarsi e le notifica di eventuali cambiamenti. In un certo senso le classi del ViewModel sono ciò che definiamo "Controllers" negli altri Pattern.
- Il **Model** Layer modella i dati e gli "oggetti" della realtà materia di studio; è indipendente da tutti gli altri Layer e rappresenta soltanto il dominio e non la logica per la gestione dei vari casi d'uso (compito delle classi ViewModel). E' possibile esporre anche

questi dati in output soltanto se sono dichiarati attraverso le classi ViewModel.

Sottolineo nuovamente che, grazie alle classi del ViewModel, il View Layer è completamente indipendente dal Model Layer.

E' indispensabile dire che sono da evitare l'abilitazione e la disabilitazione di elementi UI nel Code-behind poiché queste operazioni sono una conseguenza della mutazione degli stati logici dell'applicazione; pertanto l'abilitizazione/disabilitazione deve essere gestita dalle classi del ViewModel Layer, nel modo più generico possibile senza vincolarsi ad uno specifico elemento grafico (questo è possibile grazie al data-biding).

2.3.3 Il Data-Binding in UWP

Per permettere al View Layer di comunicare con il ViewModel Layer è necessario dirgli da dove deve prendere/scrivere le informazioni e, in alcuni casi, come rimanere sincronizzati con esse; questo avviene attraverso il meccanismo di data-binding, e in questo paragrafo analizzeremo come viene implementato in UWP.

Esistono due tipi di binding in XAML: {x:Bind} e {Binding}, entrambi dichiarati nel linguaggio di Markup (più precisamente nell'attributo di un controllo dell'UI a cui si vuole legare un valore). Senza soffermarci troppo sulle loro differenze, è stato osservato che {x:Bind}, oltre ad essere più recente di {Binding}, è anche migliore dal punto di vista computazionale poiché verifica gli errori al tempo di compilazione e non a run-time, evitandoli durante l'uso del programma; di conseguenza l'applicazione è più veloce, usa meno memoria ed è più facile da *debuggare*.

Facendo un esempio pratico, possiamo dire ad un *"TextBlock"*[4] di far visualizzare su schermo le informazioni contenute in una ViewModel, oppure associare all'evento "Click" di un pulsante una funzione, come in figura 2.2.

```
<TextBlock Text="{x:Bind Table.Number, Mode=OneWay}"
           Foreground="White"
           FontWeight="Bold"
           HorizontalAlignment="Stretch"
           VerticalAlignment="Stretch"
           TextAlignment="Center" />
```

Figura 2.2: Blocco di testo che mette in output il numero di un tavolo contenuto in TableViewModel

```
<components:ButtonIcon Image="../../Images/aggiungi_sala-10.png"
                       Text="Aggiungi Sala"
                       Click="{x:Bind RoomViewModel.StoreRoom}"
                       IsEnabled="{x:Bind RoomViewModel.RoomActionsViewModel.AddRoomEnabled,
                           Mode=OneWay}" />
```

Figura 2.3: Pulsante il cui evento "Click" è Bindato alla funzione "StoreRoom" nella RoomViewModel

Come si intuisce dalla figura 2.3, l'attributo "IsEnabled" può assumere solo valori booleani. È lecito domandarsi cosa sia possibile bindare ai vari attributi e la risposta è semplice: qualsiasi informazione che sia dello stesso tipo dell'attributo del "tag" al quale ci si riferisce. È possibile che sorgano in alcuni casi delle incongruenze tra il tipo con il quale il dato è rappresentato e il modo nel quale lo si vuole mostrare: per esempio ho usato un ENUM per rappresentare lo stato del tavolo (Occupato, Prenotato, In Ritardo etc.) ma nell'interfaccia grafica questa informazione

[4] Un **TextBlock** è un controllo XAML semplice che permette la visualizzazione di piccole quantità di testo.

deve essere notificata da un colore. Nasce quindi il bisogno di **convertire** l'informazione da un formato ad un altro associando un colore (ad esempio il rosso) ad un enumerativo (tavolo occupato). Questa funzione è svolta da classi *converters*, anch'esse da includere nel binding, di cui parlerò nell'omonimo paragrafo.

Manca ancora un'ultima informazione riguardante il binding da analizzare prima di comprendere appieno i converters e tutti gli strumenti che sono dipendenti da esso;

Il binding può essere effettuato in tre modi:

- **One-Time:** Utilizzato quando si vogliono semplicemente mostrare dati che non cambiano durante il ciclo di vita della schermata poiché, nel momento della creazione dell'interfaccia (e solo in questo momento), l'attributo legge il dato a cui è bindato; qualsiasi modifica effettuata sulla vista o sul dato, non verrà notificata.
- **One-Way:** Utilizzato quando si vuole osservare l'informazione, così da aggiornare l'UI in seguito a modifiche dell'informazione stessa; utilizzo tipico dei dati di sola lettura.
- **Two-Way:** Utilizzato quando si vuole sia osservare che modificare l'informazione da un punto di accesso dell'UI; questa modalità è tipica dei dati che necessitano sia di lettura che di scrittura, come il campo di testo di una form.

Durante la descrizione del "One-Way" e "Two-Way" è stato usato il termine "**Osservare**"; si vuole così intendere che va tenuta traccia dei cambiamenti che avvengono sui dati, così che la UI possa sincronizzarsi di conseguenza. Ma essendo questo processo non automatico, è necessario che qualcosa notifichi i vari controlli della UI qualora vi siano

cambiamenti sui dati a cui sono bindati, così che possano allinearsi di conseguenza. Nel prossimo paragrafo verrà descritto nel dettaglio questo processo.

2.3.3.1 La classe “Observable”

Continuando il discorso del paragrafo precedente vediamo adesso cosa vuol dire rendere una fonte di dati “osservabile”. Per essere più precisi, la definizione giusta da usare è “che si fa osservare”, poiché sarà proprio l’oggetto a notificare cosa dentro di sé è cambiato; aggiungo, in riferimento a ciò che è stato detto riguardo al pattern MVVM, che sono le classi del ViewModel a notificare la vista di eventuali cambiamenti e questo è possibile grazie all’ausilio di un evento che noi manualmente evochiamo.

La classe ViewModel (e in alcuni casi anche una classe del Model) che intende notificare questi cambiamenti deve implementare l’interfaccia *INotifyPropertyChanged* la quale ha un gestore di questi particolari eventi; e adesso, per far funzionare tutto, dobbiamo invocare questo gestore e dirgli quale proprietà vogliamo notificare agli elementi in ascolto che si aggiorneranno di conseguenza.

Appare evidente che ogni classe del ViewModel Layer interessata da modifiche a run-time dovrà implementare quest’interfaccia e, visto che praticamente la quasi totalità delle classi lo sono, per una migliore programmazione si crea una classe padre strutturata con funzioni a supporto per questo compito e poi ogni classe che necessita di essere osservabile eredita da essa. Ne esistono di già fatte (e addirittura Windows ne mette a disposizione una di default), ma noi ne abbiamo fatta una personale con solo quello che ci serviva, chiamata “Observable”. Il codice di detta classe è riportato in appendice A (6.1).

Inoltre C# mette a disposizione delle collezioni Osservabili, in modo da notificare se viene aggiunto o rimosso qualcosa dalla collezione; una di queste, che è la più usata da noi, è *ObservableCollection*.

2.3.3.2 I Converters

È già stato detto a cosa servono i Converters; possiamo intuire come nel pattern MVVM essi permettono di rendere ancor più indipendente il ViewModel Layer da quello View; adesso vediamo la loro struttura e le caratteristiche principali.

I Converters sono delle classi che implementano l'interfaccia *IValueConverter* che ha come metodi Convert e ConvertBack. Il binding, quando vengono passate informazioni dalle classi del ViewModel alla View, richiama la funzione Convert che esegue la conversione logica (per esempio dallo stato del tavolo al colore associato ad esso); ConvertBack invece viene richiamata quando una View invia degli input alle ViewModel ed effettua la conversione inversa (per i Two-Way binding).

In appendice A (6.3) sono riportati il codice del Converter il cui compito è quello di convertire, nel momento del binding, lo stato del tavolo nel colore associato e anche *l'IDictionary* [5] che contiene la logica di associazione (la chiave è lo stato del tavolo, il valore è il colore) (6.2).

Esistono molti Converters già implementati per i problemi più comuni che possono essere aggiunti attraverso il download di pacchetti NuGet.

[5] Un **IDictionary** rappresenta una raccolta non generica di coppie chiave/valore.

2.4 Entity Framework

In passato nelle applicazioni Windows gli sviluppatori dovevano scrivere manualmente molte righe di codice per accedere alla base dati sottostante e comunicare con essa. Bisognava aprire una connessione, creare e configurare le tabelle, trasformare i dati recuperati dalle tabelle in oggetti dell'applicazione e viceversa. Ovviamente tutto questo portava a lavoro aggiuntivo e il tasso di errore aumentava. A questo proposito Microsoft ha automatizzato tutti questi comandi raggruppandoli in un Framework open-source chiamato "Entity Framework" (EF).

Definizione ufficiale:

> ***"Entity Framework is an object-relational mapper (O/RM) [6]that enables .NET [7] developers to work with a database using .NET objects. It eliminates the need for most of the data-access code that developers usually need to write."***

In riferimento al pattern MVVM, possiamo pensare ad un ulteriore Layer in fondo alla pila gestito completamente da EF.

[6] **(O/RM)** è uno strumento che fornisce tutti i servizi inerenti alla persistenza dei dati in un Database relazionale mediante un'interfaccia orientata agli oggetti.

[7] **Microsoft .NET** è una piattaforma di sviluppo general purpose che mette a disposizione varie funzionalità come il supporto per più linguaggi di programmazione coprendo tutti i paradigmi di programmazione, consentendo l'esecuzione su più piattaforme ed in presenza di scenari applicativi variegati.

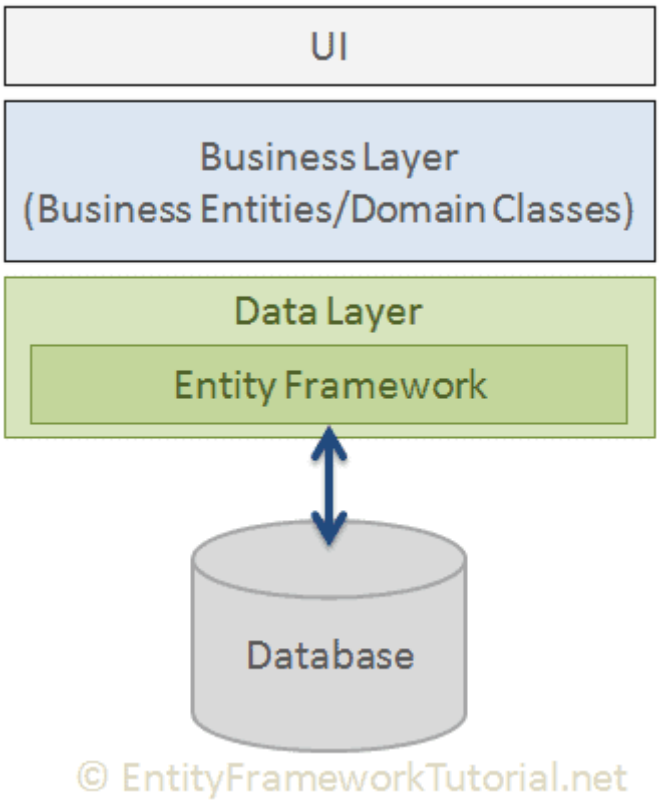

Figura 2.4: EF tra i Layer di business (ViewModel e Model) e il Database

Microsoft ha introdotto EF nel 2008. Esso si è sviluppato fino al 2013 con l'ultima versione (la 6) stabile e priva di bug. Dopo il 2013, invece di continuare l'evoluzione di questa versione già ottimizzata, è iniziato lo sviluppo di "EF Core", il quale è più versatile, in continua evoluzione e supportato da più sistemi operativi.

Le differenze tra EF e EF Core sono davvero poche, pertanto molte delle guide che si trovano su EF sono valide anche per EF Core. Nella nostra applicazione si è usato EF Core poiché UWP lo supporta meglio.

Prima di proseguire, bisogna aprire una parentesi sul significato di "Entity" per EF e "Context Class", poiché nell'esposizione delle caratteristiche del Framework, nonché del suo utilizzo, tutto girerà intorno a questi termini.

2.4.1 La Context Class

La “Context Class” è la classe che permette di utilizzare i comandi di EF attraverso i suoi metodi. Rappresenta una sessione con il Database attraverso la quale è possibile eseguire i metodi CRUD; essa deve essere figlia della classe DBContext che contiene tutti i suddetti metodi base. Inoltre in questa classe è possibile sovrascrivere i metodi per configurare alcune impostazioni del Database manualmente.

Poiché volevamo utilizzare un’unica sessione coincidente con il ciclo di vita dell’applicazione, invece di istanziare ogni volta un nuovo oggetto della context class nel momento in cui era necessario accedere al Database, abbiamo realizzato un approccio Singleton così da riusare sempre e soltanto la stessa sessione e contemporaneamente mantenere il tracking degli oggetti dell’applicazione: se si recupera dal Database un oggetto e conseguentemente si fanno delle modifiche su questa istanza, non avendo chiuso la sessione in cui esso è stata recuperata, le modifiche verranno automaticamente applicate anche alle tabelle nel Database. Questo tipo di configurazione è chiamato “Connected Scenario”. In appendice A (6.3) è riportata l’implementazione di questa context class.

2.4.1.1 Come comunicare con il Database

Una volta aperta la sessione con il database attraverso l’istanziazione della context class, è possibile fare modifiche e query verso le n-uple nel database semplicemente lavorando sugli oggetti del Model, sarà poi EF a tradurre tutte le richieste in modo che siano comprensibili dal Database. Vedremo più avanti come questo avviene.

Il commit viene effettuato quando viene richiamato il metodo della context class *“SaveChanges()”*. Esiste anche la funziona per il rollback.

2.4.2 Definizione di “Entità” in Entity Framework

In EF, è chiamata “Entity” (o entità) una generica classe che viene associata ad una tabella nel Database e ogni suo attributo ad una colonna. Per permettere questa associazione, questa classe deve essere inclusa come proprietà del tipo *“DbSet<class>”* nella context class.

Un Entity ha due tipi di proprietà, ossia “Proprietà scalari” (come int o string) e “Proprietà di navigazione”, ricordando che per proprietà si intende l’attributo di una classe (poiché si ribadisce che un’entità è una classe).

Le “Proprietà scalari” sono tutti i tipi primitivi e vengono mappate come una colonna nel Database.

Le “Proprietà di navigazione” rappresentano le relazioni tra le varie Entità ed esse non sono di tipo primitivo. Ci sono due tipi di “Proprietà di navigazione”:

- Quando un’entità è in relazione con un’istanza di un’altra entità, questa è chiamata “Reference navigation property” e rappresenta la molteplicità (1). EF crea all’interno della tabella nel Database una colonna ForeignKey (FK) che punta alla PrimaryKey (PK) della tabella dell’altra Entity con cui è in relazione.
- Se un’entità possiede una collezione di Entity, questa è chiamata “Collection navigation property” e rappresenta la molteplicità (*). In questo caso, come avviene anche in molti Database relazionali, EF crea una FK nella tabella delle entità interessate nella collezione che punta alla PK del suo contenitore.

2.4.3 Come Lavora Entity Framework

Il primo compito di EF è quello di creare un “Entity Data Model” (EDM). EDM contiene informazioni per manipolare e gestire tutti i dati in qualsiasi forma essi siano, sia che siano oggetti dell’applicazione sia che siano n-uple provenienti dalle tabelle del Database; inoltre EDM possiede le informazioni per è passare da una rappresentazione all’altra.

Da ciò che è stato appena detto, si intuisce che l’EDM è formato da tre componenti:

- **Conceptual Model:** questo è un modello creato sulla base di informazioni elaborate sulle classi dell’applicazione e le loro relazioni, di configurazioni scritte manualmente nella context class e di convenzioni seguite. È indipendente dalla progettazione delle tabelle nel Database.
- **Storage Model:** altro non è che lo schema relazionale del Database sottostante; può essere modellato partendo dalle classi di dominio (approccio Code-First) oppure da un Database già esistente (approccio Database-First). Questo Model dipende principalmente dal *Database provider* [8] utilizzato.
- **Mappings:** qui risiedono le informazioni di mappatura necessarie per creare connessioni tra Conceptual Model e Storage Model, così da trasformare il dato da una rappresentazione all’altra.

[8] Un **Database provider** è una libreria software che implementa un particolare DBMS e tutti gli strumenti per poiterci comunicare

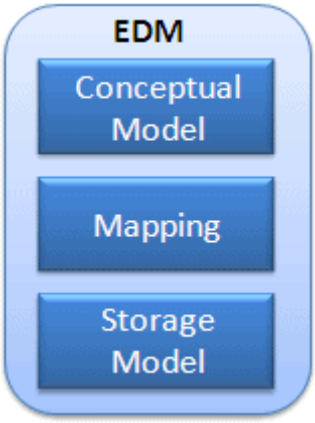

Figura 2.5: Entity Data Model

Ovviamente, nel caso in cui si cambiasse il Database Provider, soltanto alcune informazioni nello Storage Model (e conseguentemente le informazioni sulla mappatura) dovrebbe cambiare, lasciando inalterato il Conceptual Model.

Grazie a EDM, EF è in grado di fare query e salvataggi convertendo automaticamente i dati e le richieste nel formato giusto, ed è inoltre possibile accedere ai dati direttamente utilizzando gli oggetti del Model Layer, tenendo traccia delle informazioni che si stanno toccando nel Database.

2.4.4 Caratteristiche principali e vantaggi di Entity Framework Core

Le caratteristiche principali di EF Core, che dopo la descrizione dinanzi svolta adesso possiamo capire appieno, sono:

- **Cross-Platform:** EF Core è un Framework che può essere utilizzato su Windows, Mac e Linux.
- **Modelling:** EF crea un Entity Data Model basato sulle entità dichiarate nella context class e sul Database provider che si vuole utilizzare. Usa questo modello per comunicare con il database sottostante attraverso le classi del Model Layer.

- **Querying:** EF ci permette di scrivere query in *LINQ*[9] in modo da essere generali e concettuali senza legarsi ad un particolare DBMS, con la possibilità di poter cambiare quest'ultimo senza dover fare modifiche al codice; sarà il Database provider che tradurrà le query in LINQ nel linguaggio specifico del DBMS utilizzato.
- **Change Tracking:** EF mantiene traccia dei cambiamenti fatti sugli oggetti dell'applicazione, che verranno automaticamente applicati alle tabelle del Database una volta eseguito il commit.
- **Saving:** EF esegue comandi di INSERT, UPDATE, and DELETE verso il database basandosi sul lavoro che è stato fatto sugli oggetti del Model Layer (dichiarati come Entity) quando viene richiamato il metodo SaveChanges().
- **Caching:** Query ripetute restituiranno valori che sono nella cache invece di re-interrogare il Database.
- **Built-in Conventions:** EF ha un set di regole ben definite che automaticamente configurano un Conceptual Model.
- **Configurations:** EF permette di svincolarsi da queste convenzioni e configurare a mano il Conceptual Model usando "Data Annotation Attributes" o "Fluent API"

2.4.5 Creazione dell'EDM

In EF Core esistono due approcci per sviluppare l'EDM: l'approccio Code-First e quello DB-First. Come si capisce già dal nome, il secondo approccio presuppone che vi sia già un Database progettato da cui partire (creando prima lo Storage Model) per costruire le entità

[9] **Language Integrated Query** (**LINQ**) è un componente del .NET Framework di Microsoft che aggiunge ai linguaggi .NET la possibilità di effettuare interrogazioni su oggetti utilizzando una sintassi simile a SQL.

dell’applicazione; ma, visto che non c’è molto supporto da parte del Framework (cosa diversa per EF 6), viene per lo più usato l’approccio Code-First; quest’ultimo parte dall’applicazione, ossia dalle classi già esistenti (creando prima il Conceptual Model) e si sente il bisogno di salvare le loro istanze (dati). È l’approccio più usato poiché gli sviluppatori preferiscono seguire i principi “Domain-Driven Design” (DDD) e generare in seguito un Database a supporto; grazie a EF, quest’ultima operazione è praticamente automatica. Vediamo, a seguire, come questo avviene.

2.4.5.1 Convenzioni Code-First per la creazione del Model Layer

Abbiamo detto che, grazie ad EF, allo sviluppatore basta scrivere le classi del Model Layer e automaticamente viene creato lo schema del Database con attributi e relazioni. Ma potrebbero esserci delle particolari caratteristiche che vogliamo trasmettere al Database sottostante e questo è possibile grazie alle convenzioni Code-First, una collezione di regole che automaticamente configura un Conceptual Model interpretando le nostre classi e contenente le informazioni che devono essere trasmesse per la progettazione del Database. Questo vuol anche dire che è possibile scrivere inizialmente le nostre classi del Model Layer senza essere costretti a pensare ad un eventuale Database, ma che comunque potremmo dover modificare qualche proprietà in modo tale da rispettare quelle convenzioni che implementino la traduzione nello schema relazionale voluta, perché seguendole sappiamo con certezza quale sarà l’effetto finale nella configurazione delle tabelle del Database.

Ad esempio noi sappiamo che una tabella deve sempre avere una chiave primaria; pertanto bisogna identificare un attributo nella classe

che può rappresentare univocamente un'istanza; se non è presente un attributo con questa caratteristica, la convenzione prevede l'aggiunta di un nuovo attributo che dovrà chiamarsi *"Id"* o *"<Nome della classe>"+Id*, in modo che EF lo identifichi come chiave primaria nella tabella; se vogliamo invece che un altro attributo diventi chiave primaria, bisogna specificarlo usando i "Data Annotations Attributes" o "Fluent API", scavalcando così la convenzione.

Tralasciando le convenzioni più scontate (come la traduzione dei tipi primitivi nei tipi corrispondenti nel Database, i nomi delle colonne o i nomi delle tabelle) un occhio di riguardo va soprattutto alle convenzioni che gestiscono le relazioni tra classi e la loro traduzione in uno schema relazionale. Per questa problematica esistono varie convenzioni, ma la linea di principio è sempre la stessa: leggere la molteplicità dei riferimenti ad altri oggetti; ad esempio, se una classe ha una collezione di oggetti, essa sarà implicata in una relazione uno a molti, con la conseguente traduzione nello schema relazionale.

Capitolo 3: Sviluppo del modulo “Gestione Tavoli”

In questo capitolo entriamo nel vivo dello studio della problematica: esaminiamo il processo di sviluppo del software ripercorrendo i ragionamenti più importanti che hanno portato a determinate scelte e alla creazione dei tre Layer del pattern MVVM.

3.1 Casi d’uso

Il primo passo è stato quello di capire quali erano le funzionalità richieste e di modellare in seguito le classi del Model e ViewModel Layer. Tra queste annoveriamo:

1. **Gestire le stanze**
    1.1. **Creare/modificare una stanza,** in particolare il nome
    1.2. **Eliminare una stanza**
    1.3. **Scegliere la stanza da visualizzare**
    1.4. **Selezionare la data in cui visualizzare una stanza,**
        permettendo di visualizzare le informazioni di ogni tavolo in quella particolare data

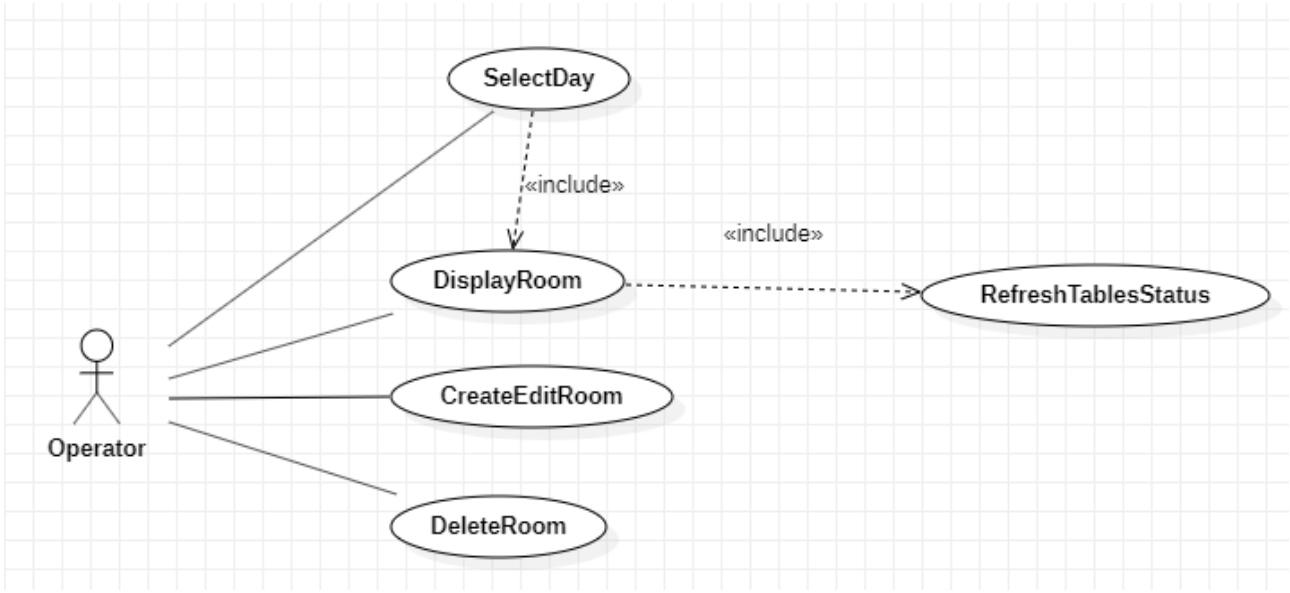

Figura 3.1: Funzionalità da effettuare sulle stanze

2. **Gestire i tavoli**
   2.1. **Creare/Modificare le informazioni di un tavolo,** come il suo numero o i posti
   2.2. **Disabilitare un tavolo,** rendendolo non più occupabile ma comunque presente nel caso si voglia visualizzare il suo storico
   2.3. **Spostare un tavolo nella stanza,** modificandone l'orientamento e la posizione

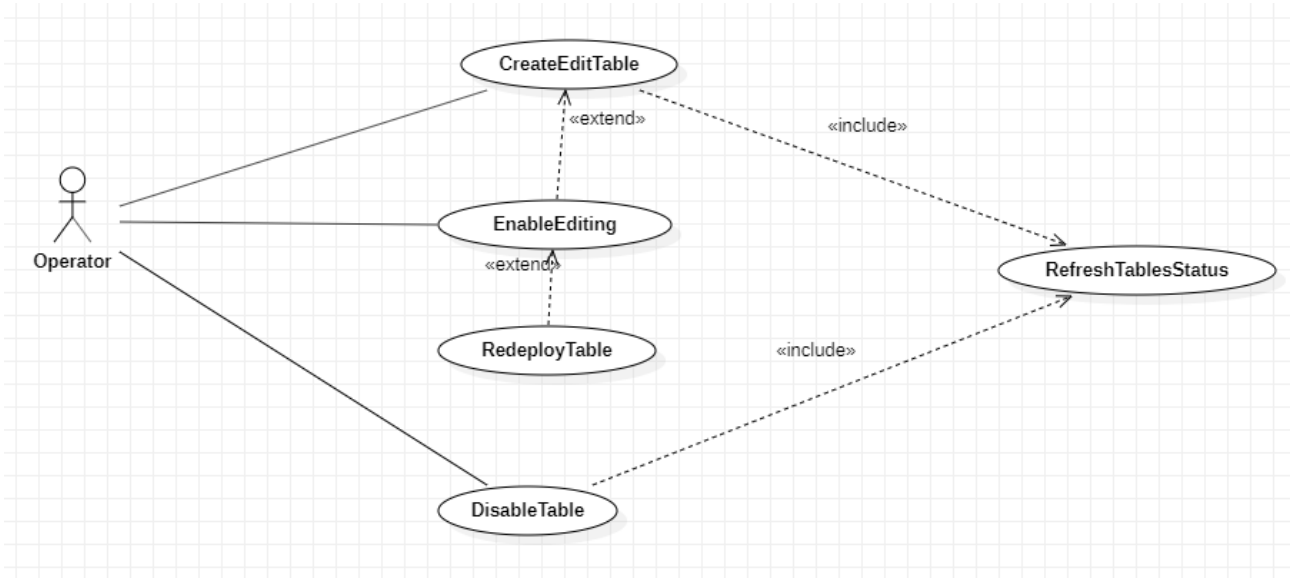

Figura 3.2: Funzionalità da effettuare sui tavoli

3. **Gestire le occupazioni**
   3.1. **Occupare un tavolo**
   3.2. **Visualizzare lo storico di un tavolo/di tutti i tavoli**
   3.3. **Richiedere il preconto**
   3.4. **Richiedere il conto**
   3.5. **Cambiare il numero di coperti**
   3.6. **Spostare l'occupazione da un tavolo ad un altro**
   3.7. **Unire due o più tavoli, di cui almeno uno occupato**
   3.8. **Cancellare un'occupazione**

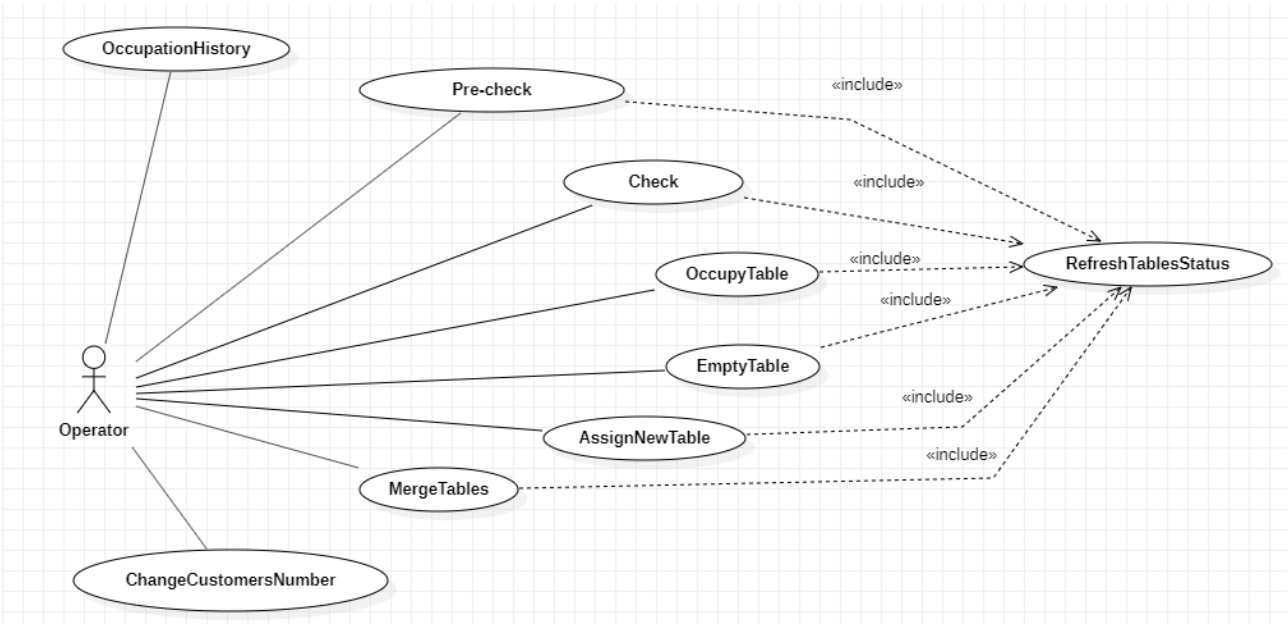

Figura 3.3: Funzionalità da effettuare sulle occupazioni

4. **Gestire le prenotazioni**
   4.1. **Prenotare un tavolo**
   4.2. **Visualizzare lo storico di un tavolo/di tutti i tavoli**
   4.3. **Modificare una prenotazione,** cambiandone l'orario o il numero di coperti
   4.4. **Spostare la prenotazione da un tavolo ad un altro**
   4.5. **Unire due Tavoli o più tavoli, di cui almeno uno prenotato**

4.6. **Eliminare una prenotazione**

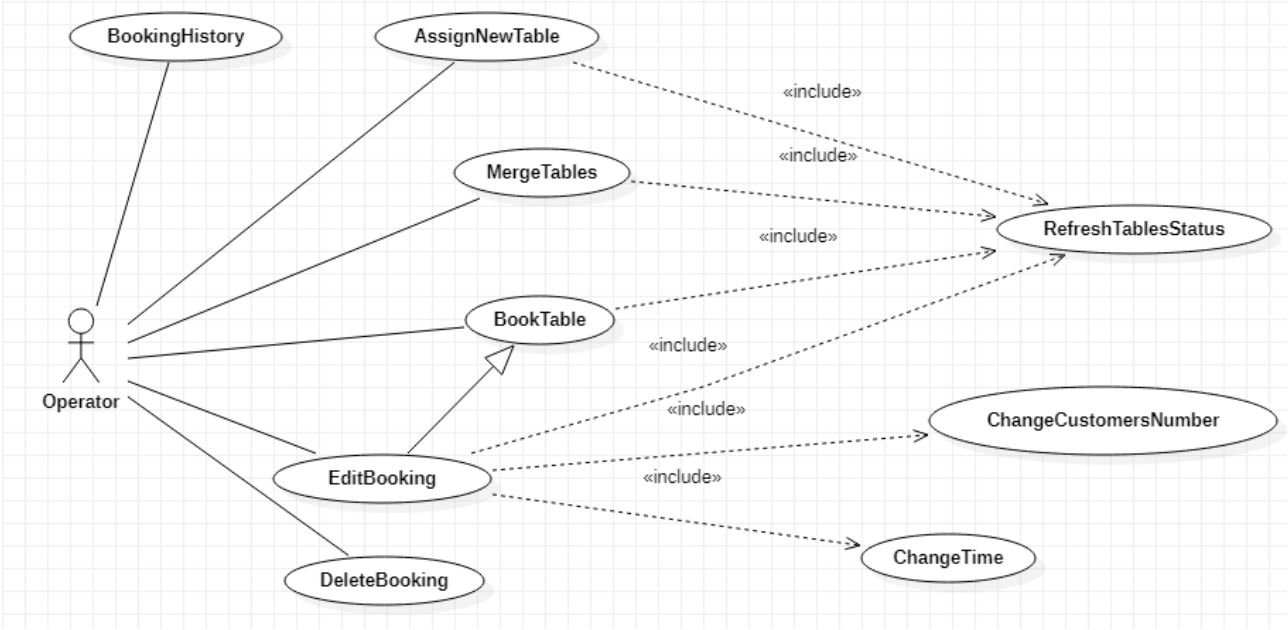

Figura 3.4: Funzionalità da effettuare sulle prenotazioni

Come possiamo vedere dai grafici, molti dei casi d'uso appena descritti ne includono uno implicito denominato **"Refresh Table Status"**, il quale descrive l'aggiornamento dello stato dei tavoli a seguito di una modifica.

3.2 Implementazione del View Layer

Dopo che sono stati analizzati i casi d'uso, i grafici dell'azienda sono stati incaricati di progettare delle interfacce che permettessero di implementare tutte le funzionalità. Il passo successivo quindi è stato quello di realizzare le interfacce progettate utilizzando lo XAML. Era fondamentale fare in modo che fossero anche adattive, ossia scalabili e ridimensionabili a seconda della grandezza dello schermo, vista la versatilità che si voleva dare all'applicazione.

Si riporta di seguito la schermata principale del Modulo progettata dai grafici che si doveva realizzare:

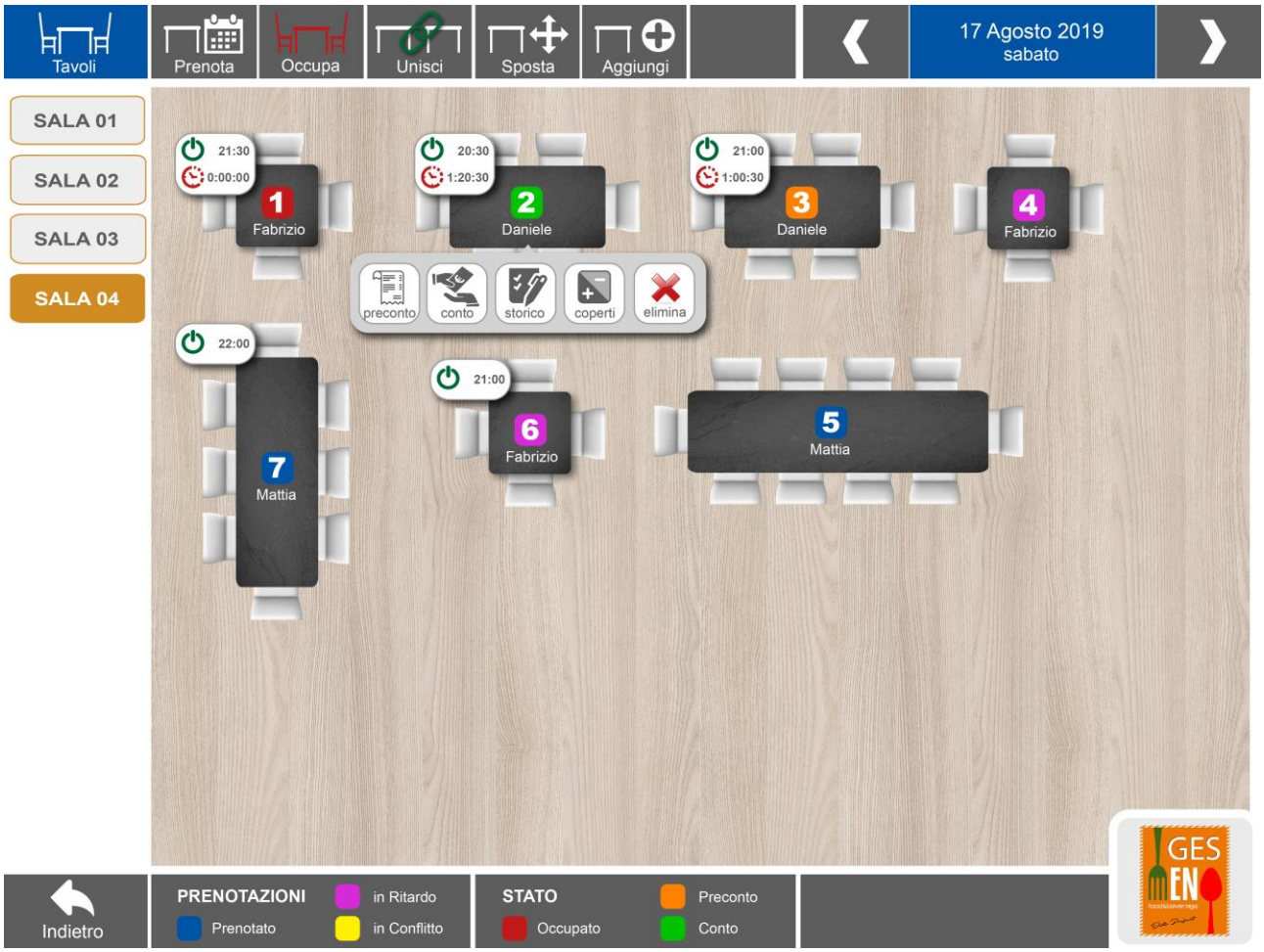

Figura 3.5: Prima prototipo della schermata del modulo progettata dai grafici

Durante lo sviluppo, per esigenze tecnologiche e progettuali, questa schermata ha subito di volta in volta piccole modifiche (esempio lampante è che gli sviluppatori, durante la realizzazione, hanno evidenziato che una delle deficienze era rappresentata dall'assenza di un comando per l'abilitazione della modifica delle stanze e dei tavoli) arrivando infine alla versione finale:

Figura 3.6: Schermata finale del modulo nell'applicazione

Quindi, dopo aver realizzato almeno lo scheletro (processo che è stato molto utile per comprendere ancora meglio la problematica e di cosa si aveva effettivamente bisogno) si è passati alla progettazione del Model Layer, ovvero alla realizzazione del diagramma delle classi che rappresentano il dominio del modulo.

3.2.1 Suddivisione della schermata principale

Per rendere il codice di questo Layer un po' più comprensibile e far sì che sia meno dispendioso effettuare revisioni e modifiche, mi è stato detto di suddividere la schermata principale in sezioni, evitando così un blocco monolitico di codice XAML in cui è meno intuitivo identificare un particolare comando. Per questo scopo sono stati utilizzati gli *UserControl* che permettono anche di creare nuovi controlli (ovvero i

“tag” dello XAML) come combinazione di controlli già esistenti e di aggiungere una nuova logica di gestione.

Infatti, gli UserControl sono stati utilizzati non solo per la suddivisione ma anche per creare dei nuovi controlli che più si avvicinavano ai nostri bisogni denominati da noi “Components”: ad esempio, come è possibile vedere in figura 3.5, i pulsanti sono molto più elaborati di quelli che l’IDE mette a disposizione poiché normalmente essi possono ospitare non più di un argomento (immagine, testo o un altro controllo), e pertanto è stato creato un nuovo controllo che potesse contenere sia un’immagine che un testo in modo da risultare come un unico argomento denominato da noi “TextBlockIcon”. In appendice A (6.5 e 6.6) vi è la sua implementazione per una migliore comprensione del suo contenuto.

3.3 Implementazione del Model Layer

Lavorando secondo un approccio ad oggetti, si è passati alle fasi di analisi e design cercando di ragionare sulla realtà che si voleva rappresentare, i casi d’uso e le informazioni necessarie per integrare tutte le funzionalità. Questo Layer ha subito numerose modifiche nel corso dello sviluppo, alcune con lo scopo di rendere il modello più solido e coerente, altre a causa delle convenzioni di Entity Framework.

In Figura 3.7 è riportato il diagramma finale delle classi, seguito dai ragionamenti che hanno portato alla scelta di questo modello:

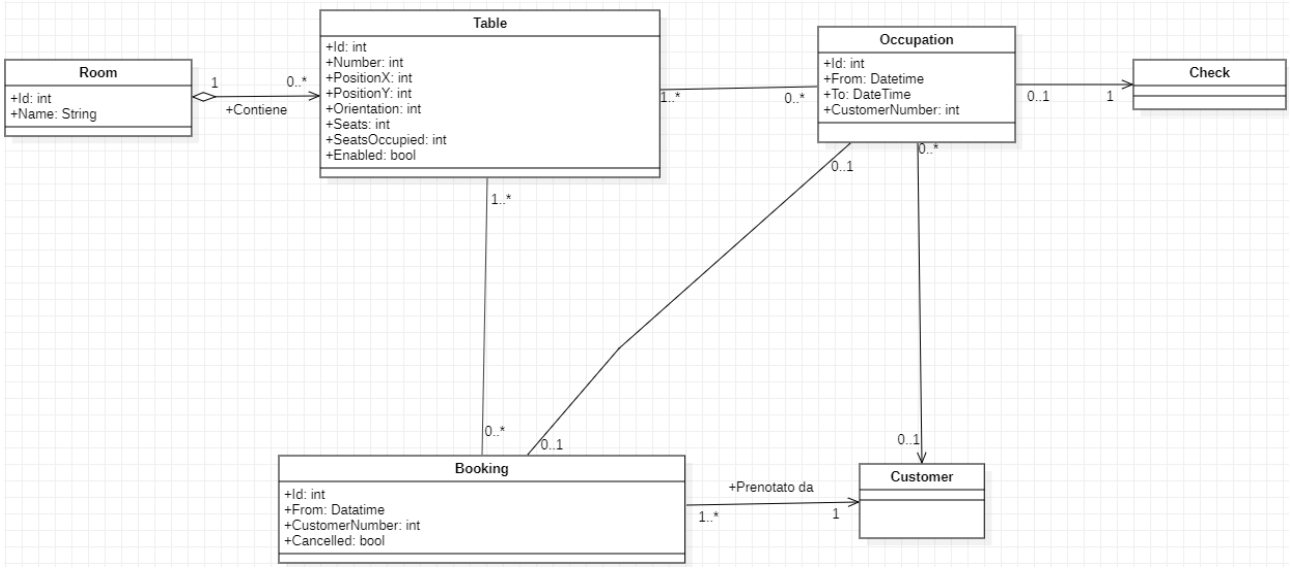

Figura 3.7: Diagramma delle classi del modulo

- **Classe Room** (rappresenta la stanza dove sono collocati i tavoli):
  - **Name:** il nome che viene assegnato ad una stanza. Esso non è univoco, in modo da permettere all'utilizzatore di creare più stanze con lo stesso nome; pertanto si è resa necessaria l'aggiunta di un ID secondo le convenzioni di EF;
  - **Relazioni:** la classe Room è una aggregazione di tavoli poiché essi non sono oggetti costitutivi della stanza e possono essere spostati da una stanza all'altra; inoltre una stanza ha senso di esistere anche in mancanza di tavoli.
- **Classe Table** (rappresenta il tavolo e i suoi attributi più intrinseci):
  - **Number:** il numero del tavolo assegnato dal locale; esso non è univoco poiché si vuole lasciare più libertà possibile all'utilizzatore permettendo, ad esempio, di creare due tavoli con lo stesso numero ma in stanze diverse; pertanto si è reso

necessario l’aggiunta di un ID secondo le convenzioni dell’EF;

- **PositionX, PositionY, Orientation:** visto che era richiesta la funzionalità di spostamento dei tavoli nella stanza e di modifica del loro orientamento, è stato necessario aggiungere questi tre attributi, due dei quali descrivono la posizione nella stanza a partire da un punto di riferimento e l’altro rappresenta l’angolo di rotazione;
- **Seats:** il numero di sedie fisicamente disposte intorno al tavolo;
- **SeatsOccupied:** sedie di un tavolo attualmente occupate, dato essenziale poiché certi casi d’uso hanno bisogno di questa informazione per permettere spostamenti di clienti o l’unione di più tavoli senza superare il numero massimo di posti a disposizione;
- **Enabled:** indica se durante l’uso del programma un tavolo è stato disabilitato; in questo caso il tavolo non è più disponibile per nuove prenotazioni o occupazioni ma comunque il sistema ne continua a tenere traccia e ne rende disponibile la visualizzazione dello storico;
- **Relazioni:** un tavolo può aver avuto diverse occupazioni e prenotazioni nel tempo (visibili nello storico) e un’occupazione (o prenotazione) può essere assegnata a più tavoli (ad esempio in un’unione di tavoli). Per questo motivo ho pensato di modellare queste relazioni come associazioni molti a molti bidirezionali tra le classi Table e Occupation (e tra Table e Booking), in modo che partendo da un tavolo si potesse risalire a tutte le sue occupazioni (prenotazioni) e

partendo da un’occupazione (prenotazione) risalire ai tavoli interessati da essa.

- **Classe Occupation** (rappresenta l’occupazione dei tavoli da parte dei clienti):
    - **From,To:** un’occupazione è caratterizzata da una data e un orario di inizio e di fine; qualora il campo relativo al termine fosse vuoto vorrebbe dire che l’occupazione è ancora attiva e quindi non è necessario mettere un altro campo per indicare il suo stato;
    - **CustomerNumber:** durante lo sviluppo è stata aggiunta una proprietà che, attraverso una somma delle sedie occupate dei tavoli con cui un’occupazione è in relazione, è possibile avere il totale dei coperti;
    - **Relazioni:** già si è parlato della relazione esistente tra le classi Table e Occupation, pertanto non si rifaranno le stesse considerazioni; un’occupazione ha un conto associato, e quest’ultimo può essere riferito ad una sola occupazione; vantaggio di questa decisione è stato quello di rendere indipendente il tavolo dal conto, facilitando spostamenti di occupazioni e operazioni di unione di tavoli, poiché basta soltanto spostare l’occupazione su nuovi tavoli. Un cliente può essere associato ad un’occupazione. Un’occupazione può essere preceduta da una prenotazione.

  Infine, nonostante esistessero già degli attributi che insieme avrebbero reso univoco un oggetto Occupation, ho deciso di continuare a seguire le convenzioni dell’EF inserendo un ID;
- **Classe Booking** (rappresenta la prenotazione dei tavoli da parte dei clienti). Molte considerazioni fatte per le occupazioni valgono

anche per le prenotazioni poiché speculari (come l'esistenza di un ID):

- **From:** data e orario di prenotazione;
- **CustomerNumber:** numero di coperti;
- **Cancelled:** indica se una prenotazione è stata disdetta;
- **Relazioni:** le considerazioni fatte per la relazione esistente tra le classi Table e Occupation descritta in precedenza valgono anche per la relazione tra le classi Table e Booking; un cliente può effettuare una o più prenotazioni.

Non è servito aggiungere attributi per descrivere lo stato di un tavolo poiché basta soltanto fare delle analisi sull'orario attuale e gli orari delle prenotazioni e occupazioni associati ad un tavolo, considerando anche le varie combinazioni che si vengono a creare (compito di TableViewModel come vedremo più avanti nel dettaglio).

Per quanto riguarda la modellazione delle classi **Customer** e **Check,** che rappresentano rispettivamente il cliente e il conto dell'occupazione, mi sono limitato principalmente a dichiararne la molteplicità con cui sono in relazione con le classi del mio modulo mentre la progettazione è stata lasciata al mio collega che si è occupato del modulo che più si avvicinava a queste.

3.4 Implementazione del ViewModel Layer

L'ultimo tassello mancante del Pattern MVVM sono le classi del ViewModel Layer (ossia le classi che regolano la logica di business), il cui compito è quello di coadiuvare e coordinare tutti i vari oggetti e processi in modo da implementare i casi d'uso.

È importante precisare fin da subito che il modulo può trovarsi in tre diverse **modalità operative**:

- **Operational:** è la modalità standard del modulo e da qui è possibile attivare una **funzionalità** (occupare o prenotare un tavolo, attivare la modalità **Edit** etc.).
- **Edit:** in questa modalità si possono apportare modifiche alle proprietà dei tavoli o delle stanze.
- **Selection:** dopo aver attivato la **funzionalità** scelta, questa modalità consente di selezionare un tavolo ed eseguire su di esso la funzionalità attivata.

Prima di addentrarci nella spiegazione del funzionamento del modulo, risulta necessario spiegare in breve le varie ViewModel che io ho progettato, la suddivisione effettuata e le responsabilità che esse possiedono.

3.4.1 TableViewModel

La prima ViewModel che ho progettato è stata TableViewModel; al suo interno vi è un oggetto Table dichiarato come proprietà e il suo ruolo è quello di aggiungere informazioni al tavolo in questione: espone le proprietà non solo intrinseche del tavolo ma anche quelle legate ad esso da relazioni con altri oggetti, come le occupazioni di quel tavolo (storico), l'occupazione attuale e la prossima prenotazione, permettendo inoltre di

eseguire dei calcoli su queste informazioni per definire in che stato è il tavolo. Implementa anche i metodi per gestire o modificare le proprietà dell'oggetto tavolo (come l'angolo di rotazione, l'abilitazione ecc.).

In TableViewModel troviamo anche queste proprietà:

- Un campo booleano che indica se, a seguito di varie analisi, il tavolo può essere abilitato per la selezione nella funzionalità richiesta;
- Un campo booleano per sapere se il tavolo è stato selezionato durante una funzionalità.

Poiché esiste una TableViewModel per ogni tavolo, ho deciso di raggrupparle, all'interno di una collezione, in un'altra ViewModel, la quale dovrà gestire l'aggiunta o la rimozione di elementi nella collezione stessa.

3.4.2 RoomViewModel

La seconda è RoomViewModel che è il punto di congiunzione di tutto il modulo poiché riunisce in sé tutte le altre ViewModel.

Essa contiene le proprietà e i comandi essenziali del modulo:

- Un'ObservableCollection contenente tutti gli oggetti Room che permette di notificare facilmente l'aggiunta o rimozione di stanze; questa viene riempita dal database nel momento in cui viene aperta la pagina dei tavoli.
- Un campo contenente la stanza selezionata, le cui informazioni e tavoli vogliamo visualizzare su schermo.
- Un'ObservableCollection di TableViewModel, per ogni tavolo all'interno della stanza selezionata, che viene svuotata e riempita ogni volta che viene selezionata un'altra stanza;

Ho sviluppato questa ViewModel secondo un approccio Singleton, poiché, costituendo essa il cuore del modulo (nonché il contenitore delle altre ViewModel) ed essendo il suo ciclo di vita uguale al tempo di utilizzo della pagina della gestione tavoli, serve un'unica istanza ed è necessario accedere in qualunque momento ai dati in essa contenuti.

Vista l'importanza di questa ViewModel, in appendice A (6.7) è riportato il suo codice.

3.4.3 RoomActionsViewModel

La terza ViewModel progettata è RoomActionsViewModel, dichiarata come proprietà di RoomViewModel, e si occupa di configurare le altre ViewModel in maniera opportuna in seguito all'attivazione/disattivazione di una funzionalità e della conseguente abilitazione/disabilitazione dei comandi presenti nella UI, poiché, per evitare errori e situazioni incoerenti durante una funzionalità, alcune di esse non devono poter essere attivate in contemporanea: quindi le funzionalità sono bloccanti e, finché non vengono disattivate o completate, non è possibile fare altro nell'applicazione. È necessario quindi che ci sia un campo booleano per ogni pulsante all'interno della schermata principale (ad eccezione per i pulsanti di selezione della stanza, i quali sono tutti legati ad un unico campo) e, se essi devono essere disabilitati, il campo ad essi associato attraverso il binding viene impostato a false. Ribadisco che attraverso i metodi di attivazione/disattivazione delle varie funzionalità questi campi vengono configurati opportunamente.

3.4.4 ModeViewModel

Vista la complessità di quattro particolari casi d'uso (occupare o prenotare un tavolo, unire dei tavoli e spostare dei clienti), ho deciso di creare quattro ViewModel dedicate per una migliore programmazione

strutturata dichiarate come proprietà di RoomActionsViewModel. Queste possiedono la logica di gestione delle funzionalità ad esse associate alleggerendo così il compito di tutte le altre ViewModel. Inoltre, viste le molteplici proprietà in comune, si è creata una classe padre generale chiamata “ModeViewModel”, che ha un metodo astratto *“OnSelection”* da attivare ogni qualvolta è selezionato un tavolo e che ogni ViewModel implementa in modo diverso a seconda del caso d’uso che gestisce.

3.4.5 Meccanismo del modulo

Infine vediamo come tutte queste ViewModel comunicano tra di loro.

Quando viene attivata una funzionalità, RoomActionsViewModel configura, come già detto, lo stato logico dei comandi ed assegna ad ogni TableViewModel in RoomViewModel (grazie al Singleton posso sempre accedervici) la ModeViewModel che gestisce la funzionalità richiesta. Subito dopo viene attivata la modalità “selection” in cui possiamo selezionare un tavolo e passare il comando alla ModeViewModel che in quel momento è assegnata alla TableViewModel dove il tavolo è contenuto e che eseguirà la funzione OnSelection per intervenire sul tavolo selezionato.

Capitolo 4: Conclusioni

Il lavoro e il ragionamento svolto da noi studenti è stato inglobato, con gli opportuni adattamenti, nella versione beta dell'applicazione che l'azienda ha attualmente concluso e che metterà in commercio nel Windows Store a partire dal 2021; i moduli attualmente progettati sono quelli della gestione dei tavoli, dei conti e delle stampanti fiscali; si prevede che in futuro verranno progettati e aggiunti altri moduli in maniera agile e veloce grazie ai concetti di programmazione strutturata acquisiti e realizzati, allineandosi alla richiesta del mercato del retail.

Il lavoro svolto, mirato allo sviluppo dell'applicazione, ha riscontrato esiti positivi, così come dimostra la dichiarazione proposta di seguito dell' Ing. Daniele Vettori, membro dell'azienda e correlatore nella realizzazione di questa tesi:

"L'applicazione è stata progettata e sviluppata ponendo la giusta attenzione agli aspetti essenziali di ogni fase del progetto. Il perfetto connubio tra le parti coinvolte: studenti e figure professionali aziendali, ha consentito di centrare i maggiori obiettivi nelle tempistiche preventivate, ottenendo come risultato un'applicazione che rispecchia i canoni di innovazione, semplicità, velocità e scalabilità richieste. Il codice è ben strutturato, facilmente manutenibile ed aperto ad integrazioni future"

Nello svolgimento del presente lavoro ho maturato la personale consapevolezza che:

- l'utilizzo e lo studio di Entity Framework sono risultati per me molto intuitivi: infatti l'accesso al database, il recupero di informazioni e la loro manipolazione sono stati semplici e immediati;
- l'interfaccia è stata progettata e creata indipendentemente dalla composizione dei due Layer "back-end", confermando ciò che è

stato detto teoricamente sul pattern MVVM e come sia poco dispendioso effettuare modifiche; inoltre il pattern si presta molto bene anche ad uno sviluppo incrementale di tipo verticale dei tre Layer;

- per quanto lo XAML permetta di creare interfacce molto accattivanti, è molto meno versatile e flessibile del classico HTML: il processo di data-binding per alcuni fattori è molto limitante e rigido così come lo è l'assegnazione degli stili ai controlli.

Capitolo 5: Bibliografia

- Documentazione UWP: https://docs.microsoft.com/it-it/windows/uwp/get-started/universal-application-platform-guide
- Documentazione MVVM: https://docs.microsoft.com/IT-IT/XAMARIN/XAMARIN-FORMS/ENTERPRISE-APPLICATION-PATTERNS/MVVM

  https://docs.microsoft.com/it-it/windows/uwp/data-binding/data-binding-and-mvvm
- Documentazione data-binding: https://docs.microsoft.com/it-it/windows/uwp/data-binding/data-binding-in-depth
- Documentazione {x:Bind}: https://docs.microsoft.com/it-it/windows/uwp/xaml-platform/x-bind-markup-extension
- Documentazione Entity Framework Core: https://www.entityframeworktutorial.net/ https://ef.readthedocs.io/en/staging/intro.html

- Immagine 2.1: https://docs.microsoft.com/IT-IT/XAMARIN/XAMARIN-FORMS/ENTERPRISE-APPLICATION-PATTERNS/mvvm-images/mvvm.png

- Immagine 2.4: https://www.entityframeworktutorial.net/images/basics/ef-in-app-architecture.png
- Immagine 2.5: https://www.entityframeworktutorial.net/Images/ef-architecture.PNG

Capitolo 6: Appendice A

Citazione 6.1: La classe Observable utilizzata

```
1.  public class Observable : INotifyPropertyChanged
2.      {
3.          public event PropertyChangedEventHandler PropertyChanged;
4.
5.          protected void Set<T>(ref T storage, T value, [CallerMemberName] string propertyName = null)
6.          {
7.              if (Equals(storage, value))
8.              {
9.                  return;
10.             }
11.
12.             storage = value;
13.             OnPropertyChanged(propertyName);
14.         }
15.
16.         public void OnPropertyChanged(string propertyName)
17.         {
18.             PropertyChanged?.Invoke(this, new PropertyChangedEventArgs(propertyName));
19.
20.         }
21.     }
```

Citazione 6.2: IDictionary che associa ad ogni possibile stato che un tavolo può avere un determinato colore

```
1.  public class TableStatusColorDictionary
2.      {
3.          private static TableStatusColorDictionary instance = null;
4.
5.          public IDictionary<TableStatus,SolidColorBrush> Dictionary=new Dictionary<TableStatus,SolidColorBrus
    h>();
6.
7.          public TableStatusColorDictionary()
8.          {
9.              Dictionary.Add(TableStatus.Conto, new SolidColorBrush(Windows.UI.Colors.Green));
10.             Dictionary.Add(TableStatus.Preconto, new SolidColorBrush(Windows.UI.Colors.Orange));
11.             Dictionary.Add(TableStatus.InConflitto, new SolidColorBrush(Windows.UI.Colors.Gold));
12.             Dictionary.Add(TableStatus.Occupato, new SolidColorBrush(Windows.UI.Colors.Red));
13.             Dictionary.Add(TableStatus.InRitardo, new SolidColorBrush(Windows.UI.Colors.Violet));
14.             Dictionary.Add(TableStatus.Prenotato, new SolidColorBrush(Windows.UI.Colors.Blue));
15.             Dictionary.Add(TableStatus.Libero, new SolidColorBrush(Windows.UI.Colors.Transparent));
16.         }
17.
18.         public static TableStatusColorDictionary Instance
19.         {
20.             get
21.             {
22.                 if (instance == null)
23.                 {
24.                     instance = new TableStatusColorDictionary();
25.                 }
26.                 return instance;
27.             }
28.         }
29.     }
```

Citazione 6.3: Converter utilizzato per associare allo stato del tavolo il colore corrispondente nel momento del binding

```
class TableStatusToColorConverter:IValueConverter
    {
        public object Convert(object value, Type targetType, object parameter, string language)
        {
            TableStatus ActualStatus= (TableStatus)value;
            return TableStatusColorDictionary.Instance.Dictionary[ActualStatus];
        }

        //Non è necessario il ConvertBack
        public object ConvertBack(object value, Type targetType, object parameter, string language)
        {
            return false;
        }
    }
```

Citazione 6.4: DatabaseContext implementato secondo un approccio Singleton

```
public class DatabaseContext : DbContext
   {

       private static DatabaseContext instance = null;

       public static DatabaseContext Instance
       {
           get
           {
               if (instance == null)
               {
                   instance = new DatabaseContext();
               }
               return instance;
           }
       }

       public DatabaseContext()
       {
           if (! (Database.CanConnect()))
           {
               Database.EnsureDeleted();
               Database.EnsureCreated();

               Seeder.LimitedSeed(this);

               CustomersSeeder.Seed(this);
               ProductsSeeder.Seed(this);
               PriceListsSeeder.Seed(this);
           }
           else
           {
               Database.EnsureCreated();
           }
       }


   protected override void OnConfiguring(DbContextOptionsBuilder optionsBuilder)
       {
           optionsBuilder.UseSqlite("Filename=TesiTest.db");
           base.OnConfiguring(optionsBuilder);
       }

```

```
protected override void OnModelCreating(ModelBuilder modelBuilder)
        {
            //Configurazione relazione molti a molti tra tavoli e Occupazioni
            modelBuilder.Entity<TableOccupation>()
                .HasKey(t => new { t.TableId, t.OccupationId });

            modelBuilder.Entity<TableOccupation>()
                .HasOne(pt => pt.Table)
                .WithMany(p => p.TableOccupations)
                .HasForeignKey(pt => pt.TableId);

            modelBuilder.Entity<TableOccupation>()
                .HasOne(pt => pt.Occupation)
                .WithMany(t => t.TableOccupations)
                .HasForeignKey(pt => pt.OccupationId);

            //Configurazione relazione molti a molti tra tavoli e Prenotazioni
            modelBuilder.Entity<TableBooking>()
                .HasKey(t => new { t.TableId, t.BookingId });

            modelBuilder.Entity<TableBooking>()
                .HasOne(pt => pt.Table)
                .WithMany(p => p.TableBookings)
                .HasForeignKey(pt => pt.TableId);

            modelBuilder.Entity<TableBooking>()
                .HasOne(pt => pt.Booking)
                .WithMany(t => t.TableBookings)
                .HasForeignKey(pt => pt.BookingId);
        }

        public DbSet<Category> Categories { get; set; }

        public DbSet<Product> Products { get; set; }

        public DbSet<PriceList> PriceLists { get; set; }

        public DbSet<Service> Services { get; set; }

        public DbSet<PriceListItem> PriceListItems { get; set; }

        public DbSet<Customer> Customers { get; set; }

        public DbSet<Check> Checks { get; set; }

        public DbSet<CheckItem> CheckItems { get; set; }

        public DbSet<Variant> Variants { get; set; }

        public DbSet<Room> ListOfRoom { get; set; }

        public DbSet<Table> Tables { get; set; }

        public DbSet<Booking> Bookings { get; set; }

        public DbSet<Occupation> Occupations { get; set; }
    }
```

Citazione 6.5: Implementazione XAML dello User Control *TextBlockIcon*

```
<UserControl
    x:Class="GesmenuCassa.Components.TextBlockIcon"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    xmlns:local="using:GesmenuCassa.Components"
    xmlns:d="http://schemas.microsoft.com/expression/blend/2008"
    xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"
    mc:Ignorable="d"
    d:DesignHeight="70"
    d:DesignWidth="90">

    <StackPanel
        VerticalAlignment="Stretch"
        HorizontalAlignment="Stretch"
        Margin="1"
        Padding="4"
        BorderThickness="0"
```

```
        Orientation="Vertical"
        Background="{x:Bind Background}">

        <Viewbox
            Margin="0"
            Height="{x:Bind ImageHeight}">
            <Image
                x:Name="imageTag"
                Source="{x:Bind Image, Mode=OneWay}"
                Margin="3,0">
            </Image>

        </Viewbox>

        <TextBlock
            x:Name="textTag"
            Text="{x:Bind Text}"
            FontSize="{x:Bind FontSize}"
            Foreground="{x:Bind Foreground}"
            HorizontalTextAlignment="Center"
            VerticalAlignment="Center"
            Margin="3,0"/>
    </StackPanel>

</UserControl>
```

Citazione 6.6: Implementazione C# dello User Control *TextBlockIcon*

```
public sealed partial class TextBlockIcon : UserControl
    {
        public TextBlockIcon()
        {
            this.InitializeComponent();
            this.Background = new SolidColorBrush(Colors.DimGray);
            this.Foreground = new SolidColorBrush(Colors.White);
            this.ImageHeight = 30;
        }

        public String Text { get; set; }

        public ImageSource Image { get; set; }

        public int ImageHeight { get; set; }
    }
```

Citazione 6.7: Implementazione della classe RoomViewModel

```
1.  public class RoomViewModel : Observable
2.      {
3.          public event NotificationReady OnNotificationReady;
4.          public delegate void NotificationReady(String NotificationText);
5.
6.          private ObservableCollection<Room> _rooms = new ObservableCollection<Room>();
7.          private Room _selectedRoom = new Room();
8.          public ObservableCollection<TableViewModel> Tables { get; } = new ObservableCollection<TableViewMode
    l>();
9.          public RoomActionsViewModel RoomActionsViewModel { get; set; }
10.
11.
12.         #region Realizzazione del singleton
13.         private static RoomViewModel instance = null;
14.         public static RoomViewModel Instance
15.         {
16.             get
17.             {
18.                 if (instance == null)
19.                 {
20.                     instance = new RoomViewModel();
21.                 }
22.                 return instance;
23.             }
24.         }
25.         #endregion
26.
27.         public RoomViewModel()
28.         {
29.             this.RoomActionsViewModel = new RoomActionsViewModel();
30.             Rooms = getRooms();
31.         }
32.
33.         #region Usecases dei pulsanti
34.
35.
36.         //Funzione per la creazione di un nuovo tavolo
37.         public async void StoreTable(Boolean error = false)
38.         {
39.             AddTableDialog Dialog = new AddTableDialog();
40.             if (error)
41.             {
42.                 //Dialog.TableNumber = insertedTableNumber;
43.                 Dialog.Error = true;
44.                 error = false;
45.             }
46.
47.             ContentDialogResult result = await Dialog.ShowAsync();
48.
49.             if (result == ContentDialogResult.Secondary)
50.             {
51.                 foreach(TableViewModel tableViewModel in Tables )
52.                 {
53.                     if(tableViewModel.Table.Enabled)
54.                     {
55.                         if(tableViewModel.Table.Number==Dialog.TableNumber)
56.                         {
57.                             error = true;
58.                             StoreTable(error);
59.                         }
60.                     }
61.                 }
62.                 if (!error)
63.                 {
64.                     Table table = new Table()
65.                     {
66.                         Number = Dialog.TableNumber,
67.                         Seats = Dialog.SeatsNumber,
68.                         PositionX = 100,
69.                         PositionY = 100,
70.                         Orientation = 0,
71.                         Enabled = true
72.                     };
73.                     this.SelectedRoom.Tables.Add(table);
74.                     DatabaseContext.Instance.SaveChanges();
75.                     Tables.Add(new TableViewModel(table)); ;
```

```
                    OnPropertyChanged("Tables");
                }
            }
        }

        //Funzione per la creazione di una nuova stanza
        public async void StoreRoom()
        {
            AddRoomDialog Dialog = new AddRoomDialog();
            ContentDialogResult result = await Dialog.ShowAsync();

            if(result== ContentDialogResult.Secondary)
            {
                Room NewRoom = new Room() { Name = Dialog.RoomName};
                Rooms.Add(NewRoom);
                DatabaseContext.Instance.Add(NewRoom);
                DatabaseContext.Instance.SaveChanges();
            }
        }
        #endregion

        #region Proprietà e funzioni per il caricamento delle informazioni necessarie

        public ObservableCollection<Room> Rooms
        {
            get { return _rooms; }
            set
            {
                _rooms = value;
                SelectedRoom = _rooms.FirstOrDefault<Room>();
            }
        }


        public Room SelectedRoom
        {
            get { return _selectedRoom; }
            set
            {
                if (value != null)
                {

                    Set<Room>(ref _selectedRoom, value);
                    getTables();

                }
                else
                    Tables.Clear();
            }
        }


        private ObservableCollection<Room> getRooms()
        {
            ObservableCollection<Room> list = new ObservableCollection<Room>();

            //carico tutto
            var temp = DatabaseContext.Instance.ListOfRoom
                .Include("Tables.TableOccupations.Occupation.Check")
                .Include("Tables.TableBookings.Booking")
                .ToList();
            foreach (Room room in temp)
            {
                list.Add(room);
            }


            return list;
        }


        private void getTables()
        {

            Tables.Clear();

            foreach (Table table in SelectedRoom.Tables)
            {
                Tables.Add(new TableViewModel(table) );
            }

        }
        #endregion
```

```
159.
160.     }
```

Ringraziamenti

Vorrei ringraziare il Professore Vincenzo Stornelli per avermi dato l'opportunità di svolgere tale lavoro, che ritengo di grande utilità per il mio arricchimento personale e per gli sviluppi professionali futuri, e per essere sempre stato disponibile e comprensivo.

Ringrazio l'Ingegnere Daniele Vettori, correlatore di questa tesi di laurea, che, sempre cortese e disponibile, ha seguito con meticolosità me e il mio collega Alessio Perozzi in ogni step del lavoro; ha messo a nostra disposizione la sua esperienza e conoscenza professionale insegnandoci davvero molto. Quello intrapreso al suo fianco è stato un percorso di grande arricchimento a tutto tondo: sento di doverlo ringraziare davvero tanto per tutto l'impegno da lui profuso nel fornirci ogni possibile conoscenza.

Un ringraziamento va ai miei familiari che, con il loro sostegno, mi hanno permesso di arrivare fin qui, contribuendo alla mia formazione personale e professionale.

Ringrazio Mamma per essere sempre stata al mio fianco cercando di alleviare le angosce e le incertezze che mi hanno assalito nell'affrontare questo lavoro che per me è stato abbastanza impegnativo.

Un particolare ringraziamento rivolgo ad Alessio, compagno di questo percorso nonché grande amico, a cui devo molto in quanto mi è stato di grande aiuto nell'offrirmi un consistente contributo, sia durante il percorso accademico sia durante lo sviluppo e la stesura della tesi. Sento, inoltre, di doverlo ringraziare, con un sentimento di profonda

amicizia, per essermi sempre stato accanto sostenendomi anche nei momenti di cedimento emotivo.

Ringrazio Miriana per il tempo che mi ha dedicato, sottraendolo ai suoi impegni giornalieri, dimostrandomi grande amicizia e fratellanza. Il suo contributo è stato rilevante per gli utili suggerimenti dal punto di vista linguistico e grafico.

Ringrazio tutti i miei amici, in particolare Angelo, per avermi sempre sostenuto nell'affrontare le difficoltà incontrate, regalandomi anche momenti di ilarità che hanno contribuito ad alleggerirmi l'impegno.

Ringrazio Giuseppe Valerio che ha corrisposto ad ogni mia richiesta di aiuto dandomi opportuni suggerimenti. A lui ho potuto fare ricorso ogniqualvolta ho avuto bisogno di sciogliere un dubbio o di risolvere un problema presentatosi durante il mio percorso universitario. Inoltre, è stato una grande risorsa nel fornire a tutti gli studenti ogni possibile materiale di studio.