Università degli Studi di Napoli "Federico II"

Scuola Politecnica e delle Scienze di Base
Corso di Laurea Magistrale in Ingegneria Informatica

Elaborato Software Architecture Design

LIGHTNING ORDER

Restaurant management application

Antonio Emmanuele - M6300
Giuseppe Francesco Di Cecio - M63001211
Giuseppe De Rosa - 00119123
Nicola D'Ambra - M63001223

A.A. 2020/2021

Indice

Capitolo 1

Introduzione

Il seguente documento descrive il processo di realizzazione di un applicazione gestionale per un locale di ristorazione.
Il sistema realizzato ha come obiettivo quello di semplificare il lavoro dei dipendenti e la gestione del locale da parte del proprietario. Le principali figure che traggono vantaggio dall'applicazione sono :

- Camerieri : possono prendere le ordinazioni dei clienti in modo del tutto automatizzato sfruttando un dispositivo mobile;
- Proprietario : può controllare la sua attività in tempo reale e ricavare statistiche dai dati memorizzati (i.e. vendite, merce consumata etc.) .
- Addetto all'accoglienza : può gestire lo stato dei tavoli.
- Chef/Pizzaiolo : può visualizzare gli ordini da preparare e segnalare quando ha terminato il suo compito.

Pertanto, i camerieri disporranno di una applicazione da eseguire su smartphone/-tablet mentre i restanti dipendenti del locale avranno un'applicazione desktop con interfacce diverse in base al ruolo che essi ricoprono. In seguito saranno descritti il processo di sviluppo adottato dal team, l'analisi e specifica dei requisiti, le scelte di progetto ed i dettagli implementativi .

Capitolo 2

Processo di sviluppo

Il processo di sviluppo utilizzato per la realizzazione del software è **UP** (Unified Process). Un processo di tipo agile ci ha permesso quindi in brevi tempi di avere un'implementazione funzionante del software, anche se non soddisfa tutti i casi d'uso richiesti.
Data la grande dimensione dell'applicazione e i tempi ridotti di consegna, l'utilizzo di un processo *guidato da piani* avrebbe generato non pochi problemi.

2.1 Github

In un primo approccio alla realizzazione del sistema si è deciso di utilizzare **Github** come piattaforma di condivisione di codice, modello e documentazione. Fin dall'inizio il lavoro è stato diviso in tre repository indipendenti:

- Una repository che contiene tutti i file di documentazione e di modellazione del software.
- Una repository che contiene solo la parte di software *backend*.
- Una repository che contiene solo la parte di software *frontend*.

Il motivo della divisione del codice in più repository è quello di scollegare fisicamente la realizzazione delle due parti. Inoltre il *frontend* è pensato per essere implementato su una piattaforma Android, mentre il *backend* su una piattaforma Desktop, accentuando tale suddivisione.
In ogni repository sono state create delle *Branch* per permettere uno sviluppo parallelo tra i vari componenti del gruppo. Infatti durante la realizzazione ogni componente

si è occupato di un lavoro, al termine dei quali si poteva continuare con un *Merge* per unirli.
All'inizio del lavoro si è anche fatto uso delle fork. In particolare si sono usate due diverse fork per il lato backend. La prima fork riguardava lo sviluppo dei proxy e la seconda lo sviluppo dell'applicazione centrale. In seguito le due fork sono state unite durante l'integrazione.

2.2 UP

Il processo di sviluppo software UP prevede lo sviluppo in iterazioni. Ognuna delle quali comprende *analisi*, *progettazione*, *implementazione* e *test*.
Prima di tutto si è effettuata una fase di **ideazione**, per poi proseguire con la fase di **elaborazione** (che comprende le iterazioni vere e proprie) ed infine una fase di **transazione** per rilasciare una piccola versione ed illustrarne il funzionamento.

2.2.1 Ideazione

Nei primi giorni si è deciso *cosa* il sistema dovesse fare. In particolare sono stati identificati gli attori che avrebbero preso parte all'applicativo e per ogni attore gli obiettivi da raggiungere. Sono stati valutati i requisiti considerando anche il tempo che si aveva a disposizione per non sfociare in un progetto troppo complesso da realizzare.
In seguito si è effettuato uno studio di fattibilità, volto a capire se il sistema poteva essere realizzato con gli strumenti che si avevano a disposizione.
Nel dettaglio nella fase di ideazione sono state eseguite le seguenti azioni:

- Workshop dei requisiti, attraverso il confronto con gli stackholder (alcuni componenti del gruppo hanno lavorato e/o lavorano ancora nell'ambito della ristorazione).
- Identificazione degli attori.
- Descrizione breve dei primi casi d'uso ricercati.
- Descrizione dettagliata dei casi d'uso più importanti, quelli che rappresentavano il "cuore" del sistema e che racchiudevano le funzionalità di principale interesse.
- Architettura di alto livello

2.2.2 Elaborazione

La fase di elaborazione è stata divisa in tre iterazioni di all'incirca 1-2 settimane ciascuna.

1. Nella prima iterazione sono stati analizzati i requisti più nello specifico focalizzando l'attenzione sulla logica centrale del sistema, costruendo:
    - Analisi
        - Modello di dominio
        - Sequence Diagram di sistema
        - Comunicazione tra Business Logic e Database
    - Progetto
        - Progetto dell'architettura del sistema
        - Progetto della base di dati
        - Progetto dell'architettura del Main System
        - Class Diagram di dettaglio della Business Logic
        - Sequence Diagram di dettaglio
    - Implementazione
        - Implementazione Business Logic (BL)
        - Implementazione del database (DB)
    - Test
        - Test di unità della BL
        - Test di integrazione tra BL e DB
2. Nella seconda iterazione:
    - Analisi
        - Comunicazione tra Business Logic, Broker e Request Generator
        - Comunicazione tra Proxy e Main System
    - Progetto
        - Class diagram di dettaglio del Broker
        - Class diagram di dettaglio del Request Generator
        - Class diagram di dettaglio dei Proxy
    - Implementazione

    - Implementazione del Broker
    - Implementazione del Request Generator
    - Implementazione dei Proxy
  - Test
    - Test di integrazione tra Proxy, Request Generator e Broker

3. Nella terza iterazione ci si è focalizzati sul lato *frontend* per rilasciare un applicativo accessibile dall'utente.
  - Analisi
    - Comunicazione tra Proxy e Applicazione Android
    - Formato dei messaggi da scambiare
  - Progetto
    - Progetto dell'architettura Android
  - Implementazione
    - Raffinamento Main System
    - Implementazione dell'applicazione Android
  - Test
    - Test di comunicazione tra Android e Sistema

Le fasi Costruzione e Transazione non sono state applicate a causa delle tempistiche, anche se sono previste secondo il modello UP. Al termine della fase di elaborazione però viene rilasciato una piccola demo per l'applicazione Android, con compatibilità minima del sistema operativo *Lollipop 5.0*, per testare il corretto funzionamento dei casi d'uso implementati.

2.3 Altre pratiche

Vista la natura fortemente disaccoppiata delle componenti ed il numero delle persone presenti nel team si è potuto andare a dividere facilmente il lavoro. In particolare nelle fasi iniziali si è usato il pair programming per il sistema centrale e per i proxy.

- Antonio Emmanuele e Giuseppe Di Cecio si sono occupati insieme del sistema centrale.

- Giuseppe De Rosa e Nicola D'Ambra si sono occupati dello sviluppo dei Proxy.

In seguito, quando si è passati alla necessità di integrazione, si è continuato ad usare il pair programming per integrare proxy ed applicazione centrale.

- Antonio Emmanuele e Giuseppe De Rosa si sono occupati dell'integrazione tra Proxy e Main System.
- Giuseppe Di Cecio e Nicola D'Ambra si sono occupati dello sviluppo dell'applicazione Android.

Altra pratica che abbiamo integrato è quella, presa dal framework scrum, del daily scrum. Ogni giorno c'è infatti stata una riunione in cui ci si è aggiornati sullo stato di tutti i lavori. Ovviamente, avendo usato il pair programming, il **daily scrum** si è svolto tra i due gruppi e non per ogni singola persona del gruppo.

Capitolo 3

Documento di visione

3.1 Introduzione

Si vuole realizzare un sistema per la gestione degli ordini e dei pagamenti di un locale di ristorazione.
L'obiettivo è automatizzare e semplificare la gestione delle attività di routine del personale, a partire dalle ordinazioni dei clienti fino ad arrivare all'organizzazione della cucina, della sala e dei pagamenti.
In particolare si vuole far fronte a tutte quelle problematiche legate alle :

- Ordinazioni dei clienti.
- Amministrazione della sala.
- Sincronizzazione tra le varie attività all'interno del locale (cucina, bar, forno, etc.).
- Gestione delle merci a disposizione.
- Pagamenti dei clienti a fine servizio.
- Storici dei dati del locale.

3.2 Parti interessate

1. Proprietario del locale.
2. Dipendenti del locale (camerieri, pizzaioli, chef, etc.).
3. Clienti del locale.

3.3 Utenti

- Dipendti di sala (camerieri, addetto all'accoglienza)
- Dipendenti di cucina/forno (chef, pizzaiolo).
- Proprietario del locale.
- Gestore delle merci (economo).

3.4 Obiettivi delle parti interessate

1. Un servizio più efficiente e automatizzato massimizza il numero di clienti serviti e la loro soddisfazione, minimizzando errori di distrazione da parte dei dipendenti.
2. Il lavoro del dipendente viene semplificato automatizzando la gestione delle ordinazioni al sistema.
3. I clienti avranno un'esperienza migliore perché verranno ridotti i tempi di attesa e possibili errori dovuti da ordinazioni errate.

3.5 Caratteristiche del sistema

Il sistema sarà in esecuzione su un'applicazione "mobile" (su smartphone o tablet) direttamente accessibile dai dipendenti del locale, ma sarà in esecuzione anche su una piattaforma "desktop", o dedicata, per gestire le richieste degli utenti che non hanno la necessità di lasciare la loro postazione durante il servizio.
In primo luogo il dipendente della sala dovrà essere in grado di registrare le ordinazioni dei clienti nel sistema il quale provvederà ad inviarle alle attività del locale interessate (cucina, forno o bar) . In secondo luogo il sistema dovrà essere accessibile

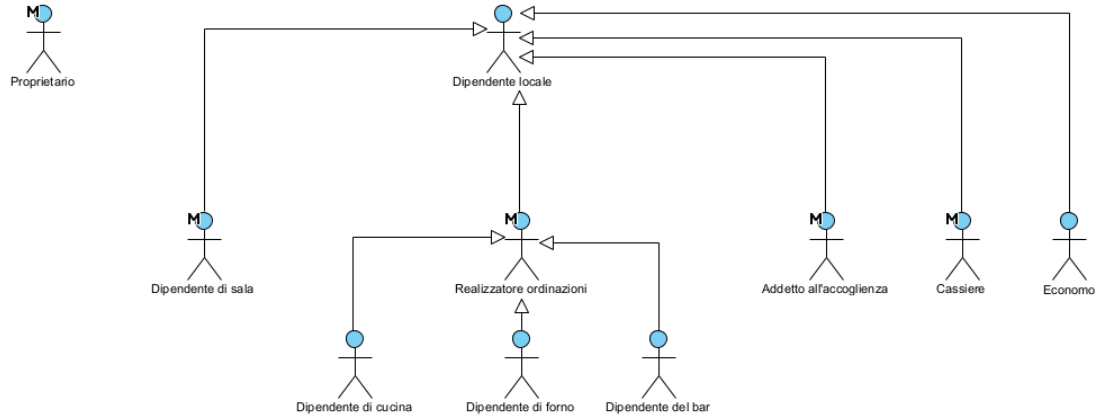
dagli altri dipendenti, come dipendenti della cucina e del bar, per visualizzare le ordinazioni dei clienti registrate recentemente.
Il sistema dovrà permette di gestire il coordinamento tra reparti diversi attraverso l'uso di messaggi di notifica ed inoltre dovrà fornire da supporto per l'assegnazione dei tavoli. Infine grazie alla memorizzazione di tutte le transazioni il proprietario potrà conoscere i guadagni del locale, la merce consumata e ricavarne statistiche.

Capitolo 4

Specifica dei Requisiti

Per avere una visione chiara di ciò che si deve costruire, quindi una descrizione dettagliata dei requisiti, sono stati effettuati dei colloqui con degli stakeholders e delle sessioni di "brainstorming" tra i componenti del gruppo. In seguito si è costruito un diagramma dei casi d'uso i quali sono stati poi descritti dettagliatamente.
Prima di elencare e descrivere i casi d'uso, è necessario identificare gli attori che caratterizzeranno il sistema.

4.1 Attori Primari



- Dipendente di sala: deve interagire con il cliente e prendere le ordinazioni in maniera tale che vengano smistate, tramite il sistema alle varie attività del

locale;

- Dipendenti di cucina/forno/bar: usano il sistema per ricevere i vari ordini e per notificare un completamento di essi;
- Addetto all'accoglienza: usa il sistema per visualizzare i tavoli disponibili ed assegnarli ai clienti;
- Cassiere: richiede al sistema il conto totale di un cliente interessato per completare il pagamento;
- Economo: si occupa della gestione delle merci ovvero l'aggiornamento delle quantità presenti in magazzino (in base alle vendite effettuate dalla sala e dalle richieste dei realizzatori di ordinazioni);
- Proprietario: può usare il sistema per aggiungere e rimuovere dipendenti, consultare dati come vendite, guadagni, quantità di merci.

Nella realtà non tutti gli attori descritti corrisponderanno a persone fisiche differenti, infatti alcuni dipendenti potranno assumere le funzionalità di più attori (ad esempio un dipendente di sala potrebbe ricoprire anche il ruolo di addetto all'accoglienza). A tal proposito quindi si è scelto di intendere gli attori come **ruoli** che un dipendente reale possa ricoprire.

4.2 Attori Finali

- Cliente: pur non interagendo con il sistema, è direttamente interessato in alcuni suoi casi d'uso in quanto gli consentono di usufruire del servizio del locale

4.3 Tabella Attori-Obiettivi

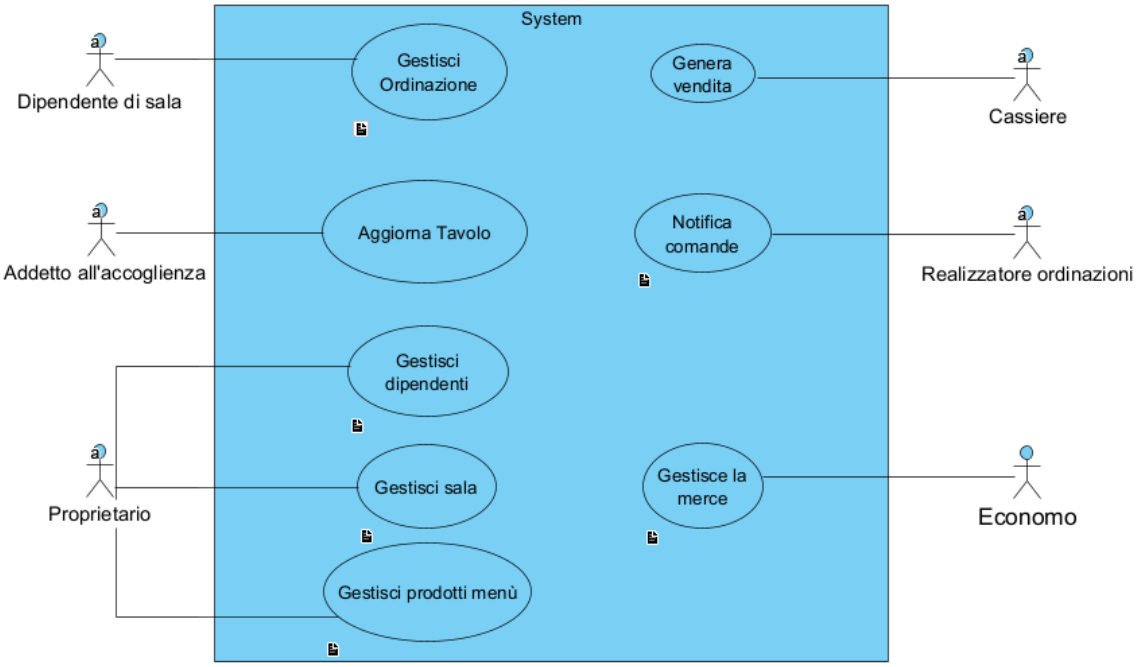
Dopo aver identificato a pieno gli attori che utilizzeranno il sistema, bisogna associare ad ogni attore i suoi obiettivi.

Attore	Obiettivi
Proprietario	Inserimento di un dipendente Rimozione di un dipendente Modifica dei ruolo di un dipendente Visualizzazione dei dipendenti Inserimento di un tavolo Rimozione di un tavolo Modifica di un tavolo Visualizzazione dei tavoli Inserimento di prodotti nel menu Rimozioni di prodotti dal menu Modifica di prodotti del menu Visualizzazione del menu completo
Dipendente di sala	Creazione di un'ordinazione per un cliente Rimozione di un'ordinazione Modifica di un'ordinazione Visualizzazione delle ordinazioni
Accoglienza	Assegnazione di un tavolo a dei clienti Modifica dello stato di un tavolo (libero, occupato, riservato, ecc.) Visualizzazione di tutti i tavoli
Cassiere	Generazione vendita di un tavolo (pagamento dei clienti)
Realizzatore di ordinazione	Visualizzazione degli ordini per area (cucina, bar, forno) Notifica completamento ordinazione
Economo	Inserimento di una merce Modifica di una merce Rimozione di una merce Visualizzazione di una merce

Gran parte degli obiettivi riguardano operazioni *CRUD* (Create, Read, Update, Delete) che possono essere raggruppate in un solo obiettivo.

4.4 Diagramma dei Casi D'Uso

Tutti gli obbiettivi CRUD dei vari attori sono stati inglobati in un unico caso d'uso con il prefisso **Gestisci**, per indicare le operazioni che racchiudono.



Nel diagramma non è stato rappresentato un caso d'uso comune a tutti gli attori: **login**.

4.4.1 Descrizione Breve

Gestisci Ordinazione

Come descritto in precedenza questo caso d'uso racchiude le operazioni CRUD riguardo ad un'ordinazione di uno o più clienti. Il cameriere infatti deve essere in grado di creare un'ordinazione per un cliente e poterla in seguito modificare, visualizzare o addirittura rimuovere dal sistema.

Aggiorna Tavolo

L'addetto all'accoglienza si occupa di far accomodare i clienti nella sala. Il suo obiettivo è quindi avere una visione completa dei tavoli comunicando tramite il sistema

quali tavoli hanno clienti in attesa di ordinazione, quali sono liberi e quali sono occupati.

Gestisci Dipendenti

Il proprietario può aggiungere/rimuovere dipendenti dal sistema ed inoltre è in grado di decidere/modificare i ruoli o il ruolo di ogni dipendente durante il servizio.

Gestisci Sala

Il proprietario può indicare l'identificativo di ogni tavolo in relazione alla sala in cui appartiene. Gli altri dipendenti quindi posso accedere tramite l'identificativo a tutte le informazioni che riguardano i tavoli (numero di persone accomodate, lista dei prodotti ordinati, ecc).

Gestisci Prodotti Menù

Il proprietario può effettuare operazioni CRUD per la gestione del menù del locale.

Genera Vendita

Il compito del cassiere è quello di registrare il pagamento dei clienti in relazione a ciò che hanno ordinato. Esso quindi deve poter risalire tramite il tavolo a tutti gli ordini associati ai clienti per ricavare il prezzo complessivo da pagare.

Notifica Ordinazione

Per rendere il sistema quanto più automatizzato possibile, i realizzatori di ordinazioni (cuoco, pizzaiolo, etc.) devono notificare il completamento di un'ordinazione tramite il sistema. In questo modo, inviando una notifica il sistema provvederà ad avvisare altri dipendenti in attesa di quel completamento.

Gestisci Merce

Ogni prodotto del menu è composto da un insieme di merci disponibili in magazzino. Il compito dell'economo è quindi quello di gestire le merci, per avere sotto controllo le merci in esaurimento. L'esaurimento di una merce comporta la non disponibilità di tutti i prodotti del menu che la contenevano.

Login

Per rendere l'applicazione generica, e non dedicata per ogni utente, essa deve fornire una procedura di *login*. Un attore quindi può accedere al sistema con le proprie credenziali e visualizzare la propria area riservata per il lavoro.

4.4.2 Descrizione Dettagliata

La descrizione dettagliata è stata fatta solo per i casi d'uso di rilievo.

Titolo	Gestisci Ordinazione
Livello	Obiettivo Utente
Portata	Applicazione Android/Desktop Lightning Order
Attore Primario	Dipendente di sala
Parti Interessate	–Dipendenti di sala (vogliono registrare/modificare un'ordinazione) –Proprietario (vuole che i clienti siano serviti) –Clienti (vogliono ricevere quanto ordinato) –Realizzatore di ordinazioni (vuole essere aggiornato sulle ordinazioni)
Pre-Condizioni	–Il dipendente di sala è autenticato e ha il ruolo di cameriere –Il tavolo che sta prendendo l'ordinazione non è nello stato di libero
Garanzia di successo	–Un'ordinazione viene creata/modificata/rimossa –Una notifica viene inviata ad un realizzatore di ordinazione –La merce numerabile costituente i prodotti viene riservata o ripristinata in caso di eliminazione/modifica
Scenario Principale	1. Dipendente di sala richiede di visualizzare la lista dei tavoli 2. Il sistema mostra i tavoli in attesa 3. Dipendente di sala seleziona un tavolo 4. Il sistema mostra la lista di ordinazioni del relativo tavolo 5. Il cliente richiede un'operazione (crea/modifica/elimina ordinazione)

	6. Il sistema riceve la modifica e notifica gli interessati.

Estensioni	5.A Dipendente di sala seleziona ordinazione a tavolo 6.A Cliente comunica i prodotti 7.A Il dipendente di sala inserisce prodotti con eventuali merci aggiuntive 8.A Il cliente comunica preferenza di consegna per prodotti 9.A Il dipendente di sala inserisce la preferenza di consegna 10.A Il dipendente di sala aggiunge ordinazione 11.A Il sistema riceve la modifica 12.A Vengono inviate notifiche ai realizzatori di ordinazioni 5.B Il dipendente di sala rimuove l'ordinazione selezionata 6.B Il sistema riceve la modifica 7.B Il sistema notifica l'evento ad eventuali realizzatori 5.D Cliente comunica al dipendente di aggiungere un prodotto all'interno di un'ordinazione 6.D Dipendente aggiunge il prodotto (o i prodotti) all'ordinazione 7.D Il sistema viene aggiornato e invia una notifica ai realizzatori 5.E Cliente comunica al dipendente di rimuovere un prodotto all'interno di un'ordinazione 6.E Dipendente rimuove il prodotto (o i prodotti) all'ordinazione 7.E Il sistema viene aggiornato e invia una notifica ai realizzatori
Scenari Alternativi	11.A.1 L'ordinazione registrata è la prima per quel tavolo 11.A.2 Il sistema cambia lo stato del tavolo da *"in attesa di ordinazione"* in *"occupato"* 5.A.1 Prodotto comunicato non è disponibile 5.A.2 Ripeti 5

	5.C.1 L'ordinazione non è modificabile (poiché in lavorazione) 5.C.2 Ripeti 5 5.D.1 Il prodotto non è disponibile 5.D.2 Ripeti 5 5.E.1 Il prodotto non è rimovibile (poiché in lavorazione) 5.E.2 Ripeti 5

Titolo	Aggiorna Tavolo
Livello	Obiettivo Utente
Portata	Applicazione Android/Desktop Lightning Order
Attore Primario	Addetto all'accoglienza
Parti Interessate	–Dipendenti di sala (vogliono avere una visione aggiornata dello stato dei tavoli) –Cliente (vuole accomodarsi) –Proprietario (vuole che i clienti entrino ed escano in maniera ordinata)
Pre-Condizioni	–Il dipendente ha effettuato il login e ricopre il ruolo di addetto all'accoglienza
Garanzia di successo	–Il cliente occupa il tavolo –Lo stato del tavolo viene aggiornato –I camerieri ricevono la notifica di un nuovo tavolo da servire

Scenario Principale	1. Addetto all'accoglienza richiede di visualizzare i tavoli di una sala 2. Il sistema mostra i tavoli con il relativo stato 3. L'addetto all'accoglienza seleziona un tavolo per cui aggiornare lo stato 4. Il sistema registra la modifica e notifica i camerieri in caso di tavolo appena occupato
Estensioni	3.A L'addetto all'accoglienza seleziona un tavolo nello stato *"occupato"* 4.A L'addetto all'accoglienza seleziona l'aggiornamento dello stato del tavolo in *"libero"* 3.B L'addetto all'accoglienza seleziona un tavolo *libero* 4.B L'addetto all'accoglienza seleziona l'aggiornamento dello stato del tavolo in *"in attesa di ordinazione"*

Titolo	Notifica Ordinazione
Livello	Obiettivo Utente
Portata	Applicazione Android/Desktop Lightning Order
Attore Primario	Realizzatore di ordinazione
Parti Interessate	–Realizzatore di ordinazione (vuole notificare ad altri realizzatori e ai camerieri l'aver completato uno o più lavori). –Cliente (vuole ricevere l'ordinazione). –Proprietario (vuole che cliente riceva l'ordinazione)
Pre-Condizioni	–L'ordinazione da segnare come completata è registrata nel sistema

Garanzia di successo	–Prodotto in ordinazione viene posto in stato di terminato –Qualora i prodotti di tutta l'ordinazione vengano posti come terminati allora anche l'ordinazione stessa viene posta come terminata
Scenario Principale	1. Realizzatore richiede ordinazioni al sistema 2. Il sistema restituisce le ordinazioni che può eseguire 3 Il realizzatore seleziona un prodotto 4 Il sistema restituisce i dati del prodotto 5. Il realizzatore notifica il completamento del prodotto 6. Il sistema si aggiorna 7. Il sistema sblocca prodotti pendenti
Estensioni	5.A Il prodotto completato era l'ultimo 6.A Il sistema aggiorna lo stato dell'ordinazione 7.A Il sistema invia un'ordinazione pendente per altri realizzatori di ordinazione in attesa di quel completamento
Scenari Alternativi	7.B.1 Non ci sono ordini pendenti

4.5 Requisiti non funzionali

Il sistema oltre a realizzare i requisiti descritti precedentemente deve rispettare dei vincoli di funzionamento. Tutti i requisiti non funzionali vengono descritti tramite delle specifiche supplementari di tipo **FURPS+**

4.5.1 Funzionalità

Per tutti i casi d'uso deve esistere una gestione degli errori ottimale. Durante il funzionamento dell'applicazione, il sistema non deve interrompersi a causa degli errori non gestiti.

4.5.2 Usabilità

L'interfaccia utente deve essere semplice e intuitiva permettendo così al dipendente di interagire con il sistema nel modo più veloce possibile.
Per quanto riguarda i camerieri, l'applicativo può essere installato sul proprio dispositivo mobile (ad esempio il proprio smartphone) in modo tale da ridurre al minimo i costi per dispositivi dedicati. Per i dipendenti i quali dispongono di una postazione fissa l'applicativo sarà invece in esecuzione su un computer fisso con monitor touch screen in modo da rendere l'interazione con l'applicazone semplice e veloce.

4.5.3 Affidabilità

Il sistema deve garantire un corretto coordinamento tra i dipendenti della sala e una prevenzione degli errori impedendo ad esempio che :

- una stessa ordinazione venga presa in carico da due realizzatori contemporaneamente;
- un articolo terminato (unità finite nel caso di una bevanda o per assenza di materie prime nel caso di una pietanza) possa essere ordinato da un cliente.

Ogni ordine registrato deve poter essere modificato, in particolare deve essere possibile eliminare uno degli articoli che ne fanno parte o l'intero ordine. Inoltre il proprietario deve avere la possibilità di gestire tutti gli utenti iscritti all'applicazione. In altre parole nel momento in cui un dipendente non farà più parte dell'azienda questo deve poter essere eliminato dalla *user list* in modo da impedirgli l'accesso all'applicazione con le sue vecchie credenziali.

4.5.4 Prestazioni

Il sistema deve funzionare in tempo reale. La gestione delle ordinazioni, soprattutto, deve poter rispondere immediatamente a variazioni, modifiche e aggiunte di eventuali ordinazioni nel sistema, per notificare velocemente coloro che dovranno realizzarle ed evitare così possibili incomprensioni tra reparti.

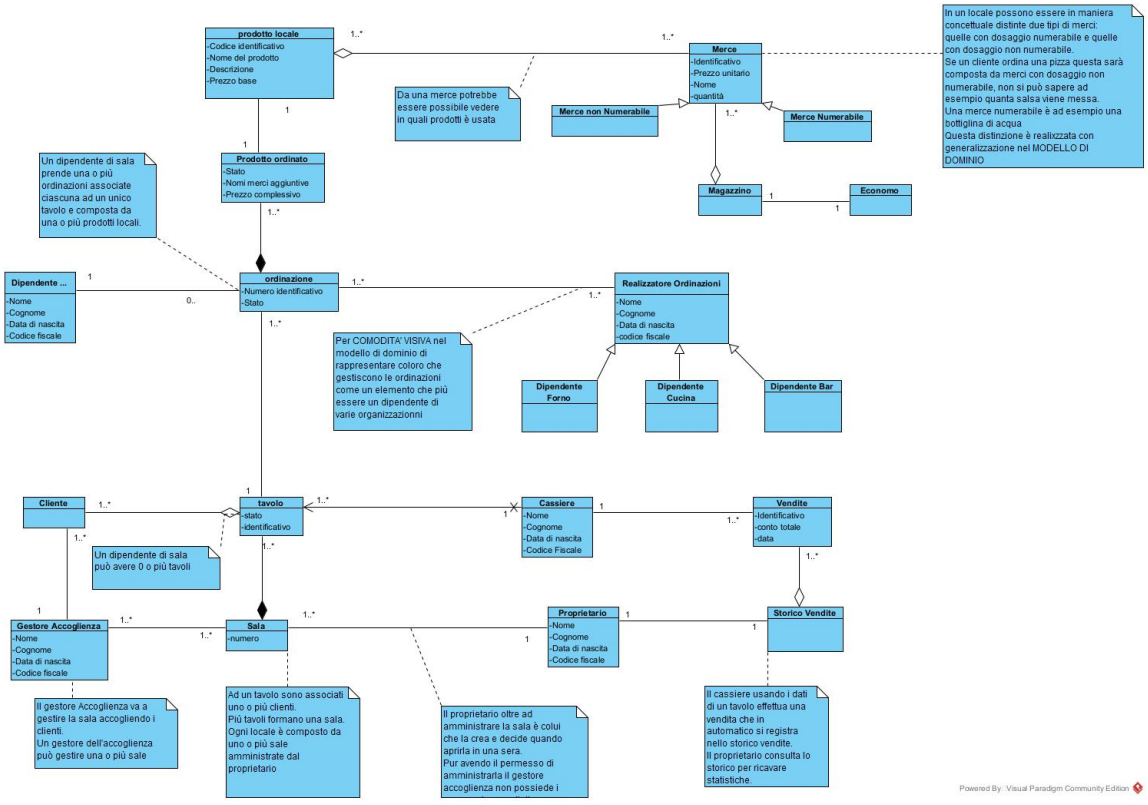
4.5.5 Sostenibilità

Nel lato "mobile" il sistema deve adattarsi alla maggior parte degli smartphone in commercio, onde evitare problemi di compatibilità.
Nel lato "desktop" il sistema deve essere portabile per poter essere configurato in un qualunque ambiente.

Capitolo 5

Analisi dei requisiti

5.1 Modello di Dominio



Con questo primo diagramma di analisi si sono fatte le prime ipotesi di funzionamento (descritte dai commenti), mettendo in evidenza le relazioni tra le varie entità del sistema.

1. In un locale con più camerieri si può verificare la situazione in cui diversi camerieri possono prendere le ordinazioni ad uno stesso tavolo. Con questa ipotesi quindi il cameriere non viene associato direttamente al tavolo, ma all'ordinazione che ha creato per esso.

2. Il sistema gestisce le merci in modo automatizzato. Esse sono divise in *numerabili* e *non numerabili*. A tal proposito, quando il cameriere crea un'ordinazione che contiene delle merci numerabili, esse vengono immediatamente scalate dal magazzino.

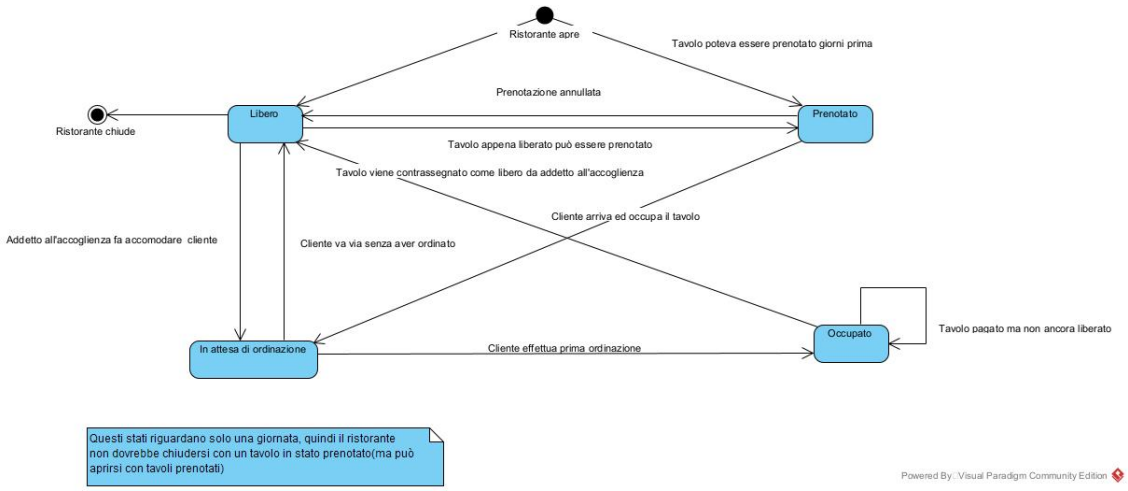
3. Tutte le vendite completate vengono memorizzate in uno storico.

4. Il prodotto che viene aggiunto all'ordinazione non equivale al prodotto presente nel menu. Esso infatti può subire delle modifiche come aggiunta o rimozioni di merce dalla sua versione di base. Viene indicato quindi come l'entità *Prodotto Ordinato*.

5.2 State Chart Diagram

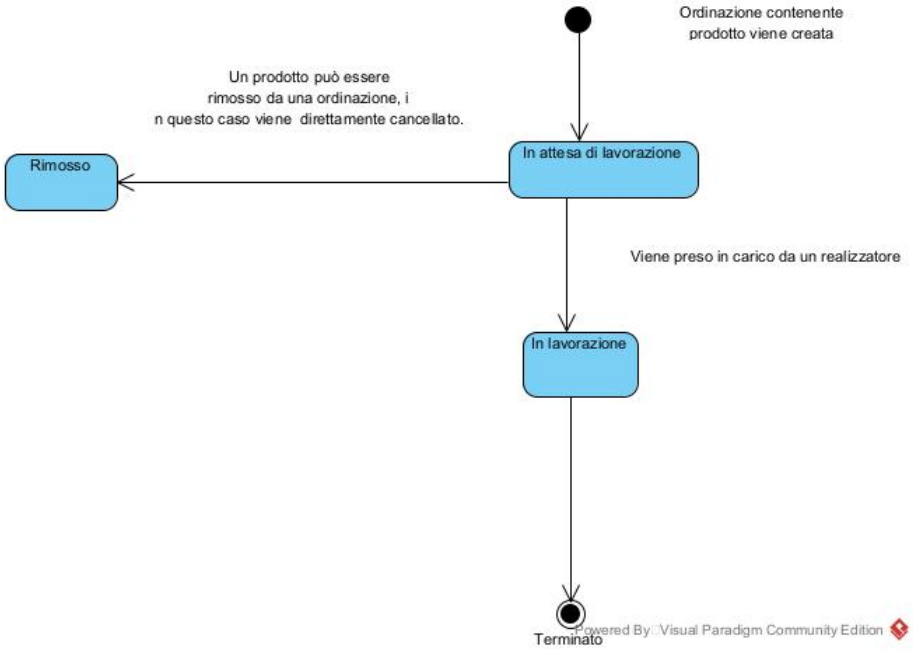
Alcune entità del sistema hanno delle funzionalità diverse in relazione allo stato in cui si trovano.

5.2.1 Tavolo



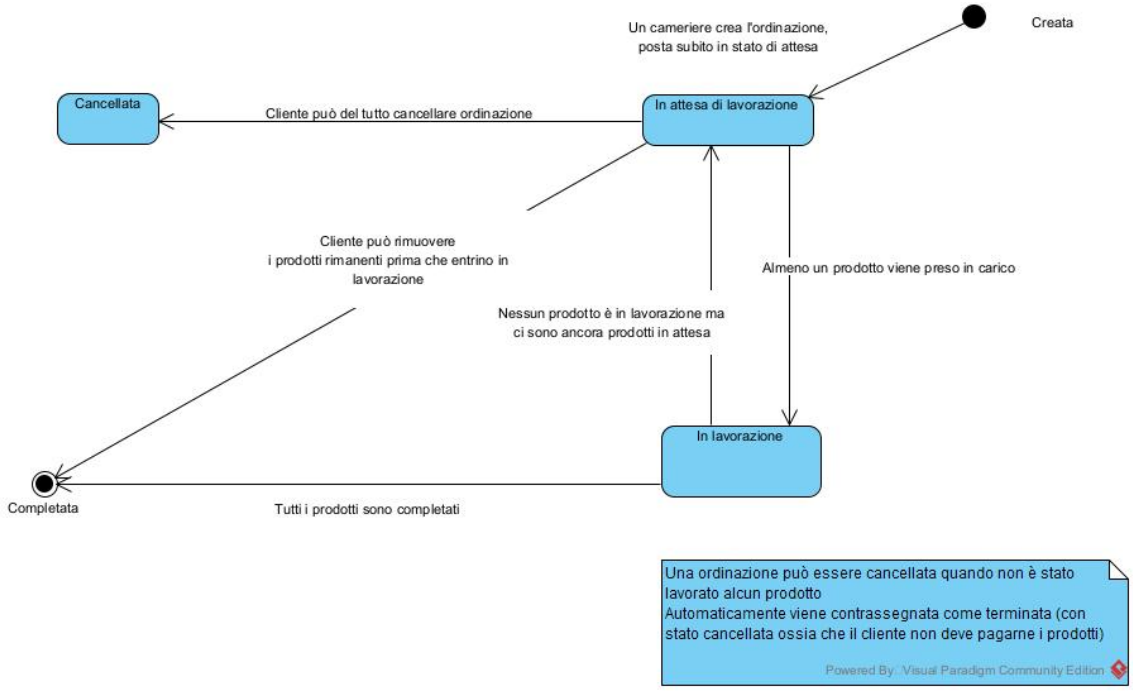
Questi stati valgono solo durante l'esecuzione del sistema. In teoria il locale non può chiudere con tavoli occupati o in attesa di ordinazione. Attraverso questi stati esiste anche più controllo per evitare errori umani. Ad esempio un cameriere non può prendere le ordinazioni di un tavolo libero, ovvero non occupato da nessun cliente.

5.2.2 Prodotto Ordinato



Gli stati vengono aggiornati tramite dei feedback di chi realizza i relativi prodotti.

5.2.3 Ordine



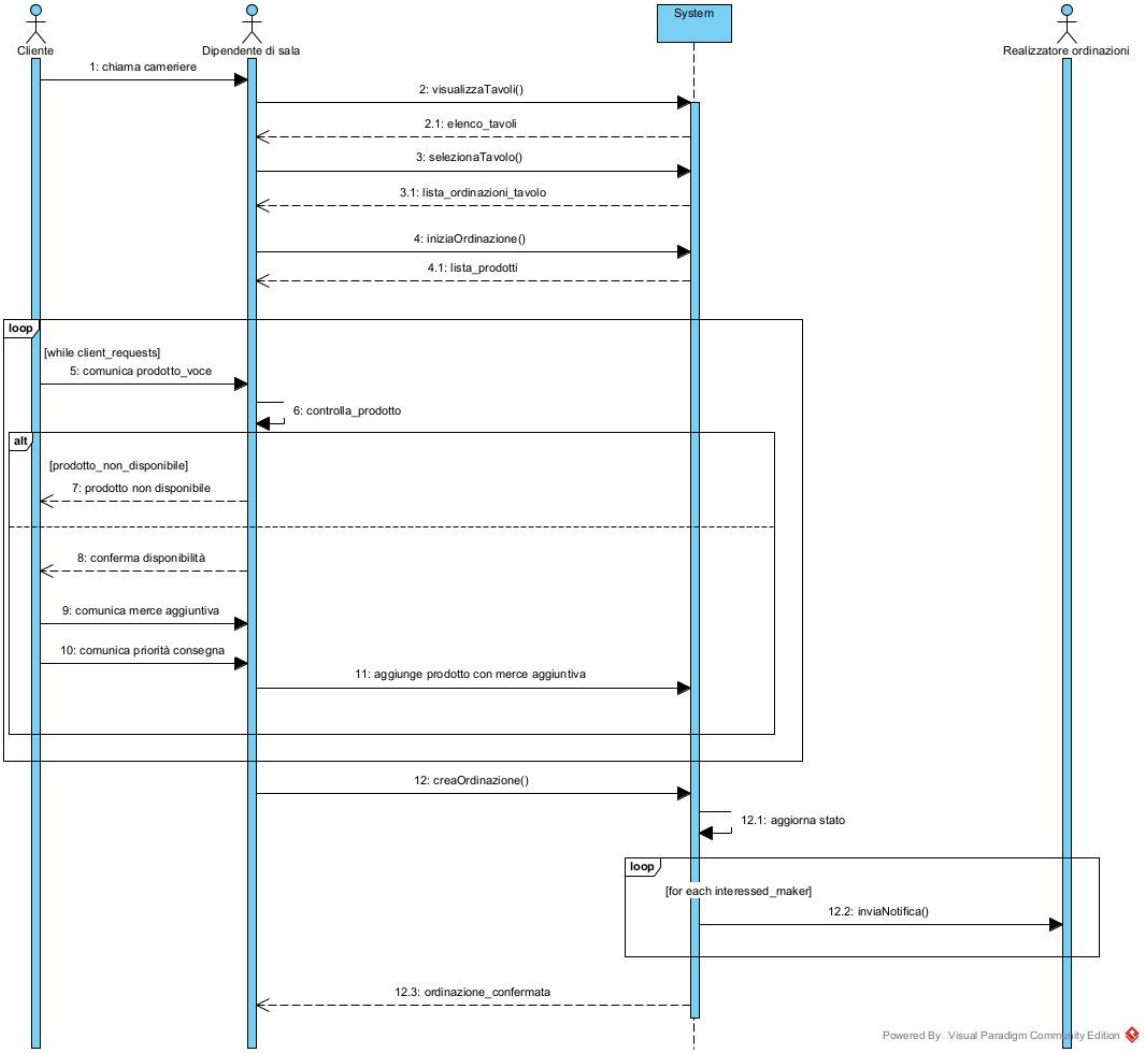
Gli stati dell'ordine dipendono molto dagli stati dei prodotti ordinati che lo compongono, essendo infatti un'aggregazione di prodotti.

5.3 Sequence Diagram di Dominio

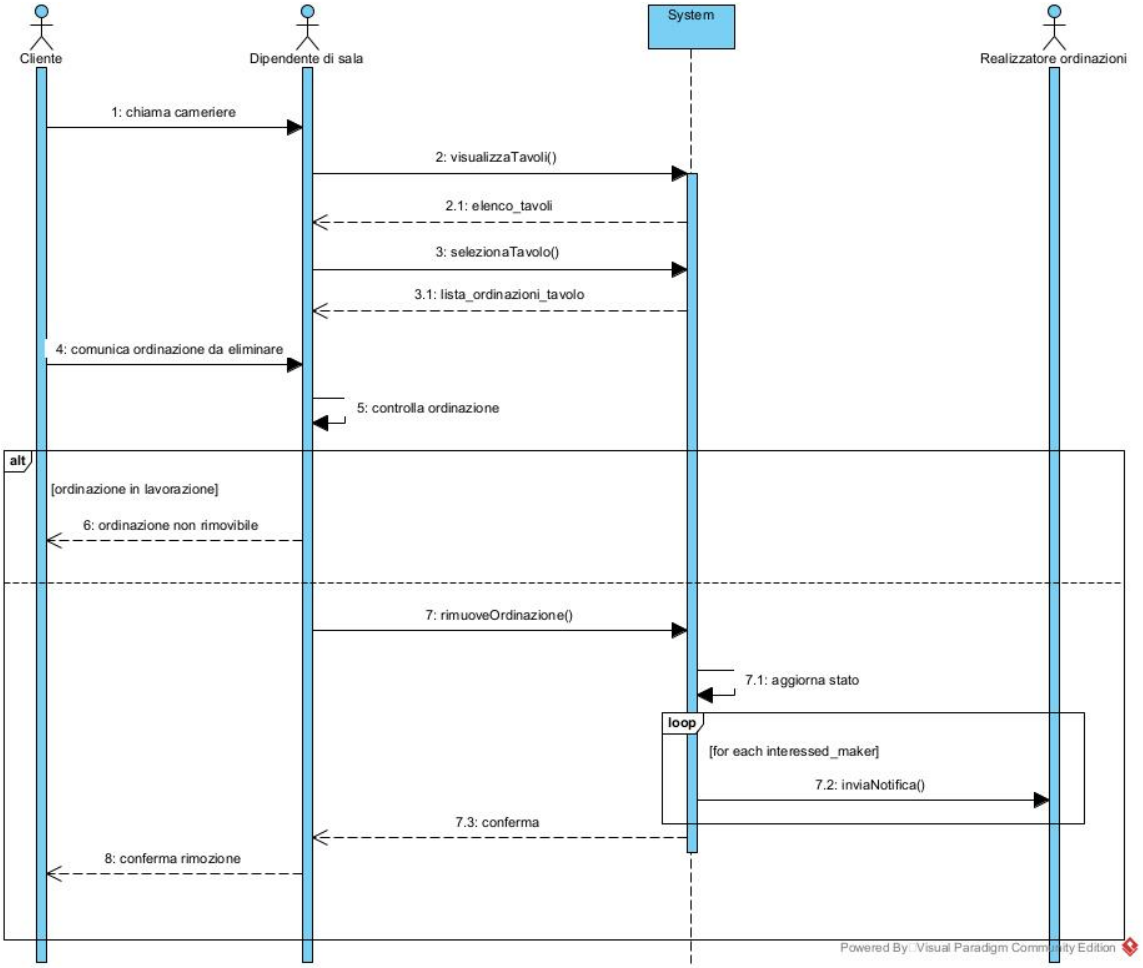
I seguenti sequence diagram sono realizzati solo per indicare le interazioni tra un utente e il sistema.

5.3.1 Gestisci Ordinazioni

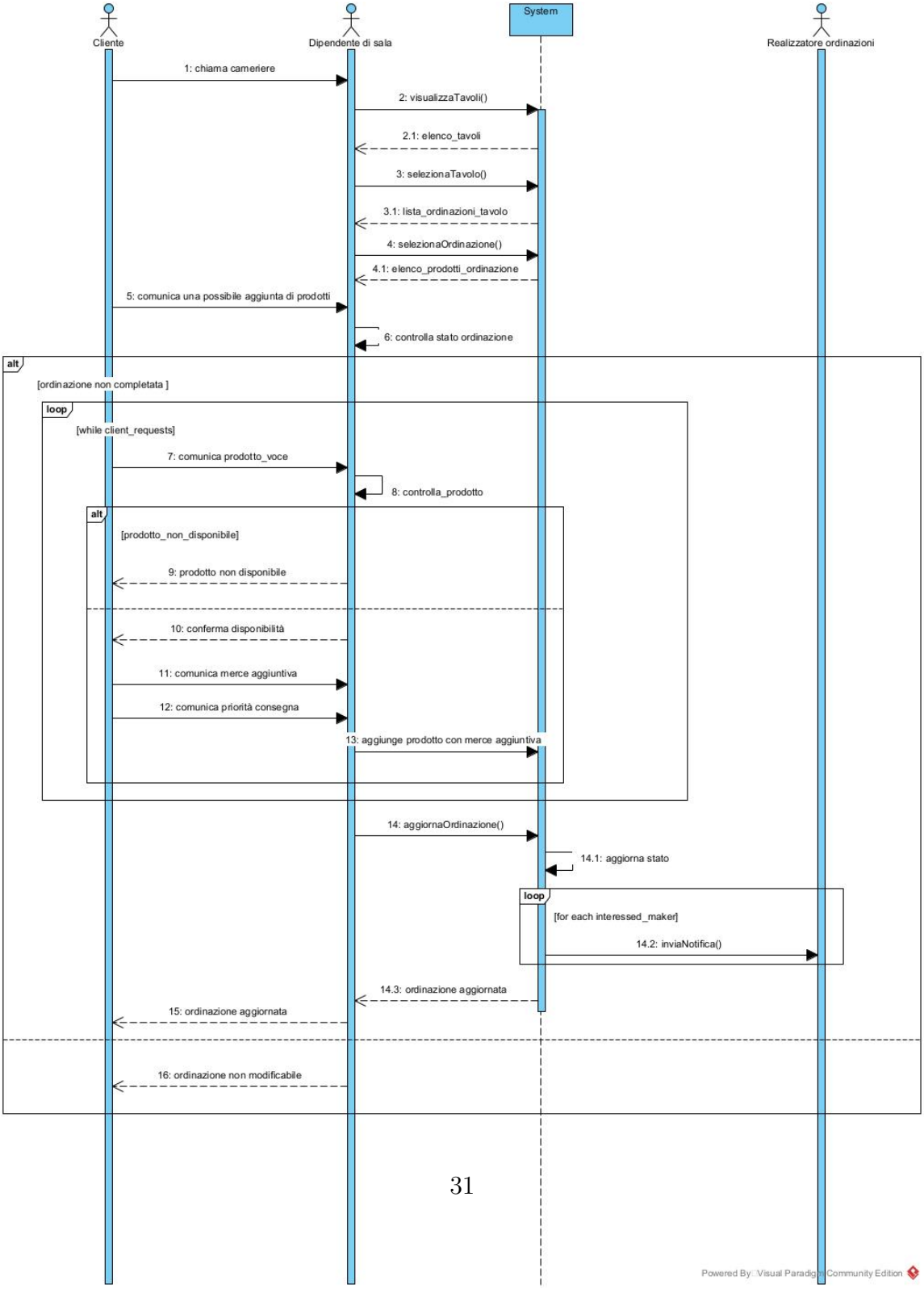
Estensione A



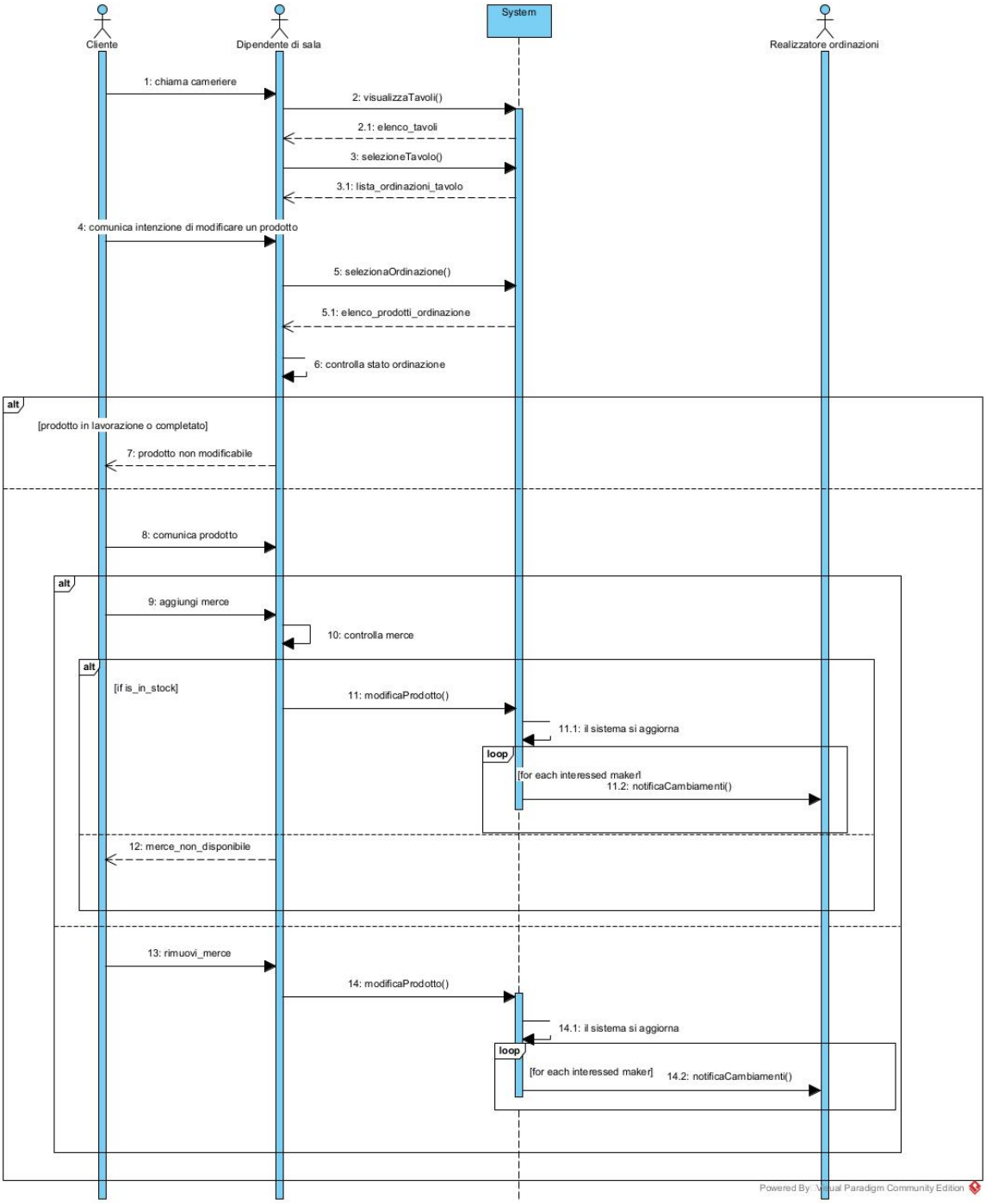
Estensione B



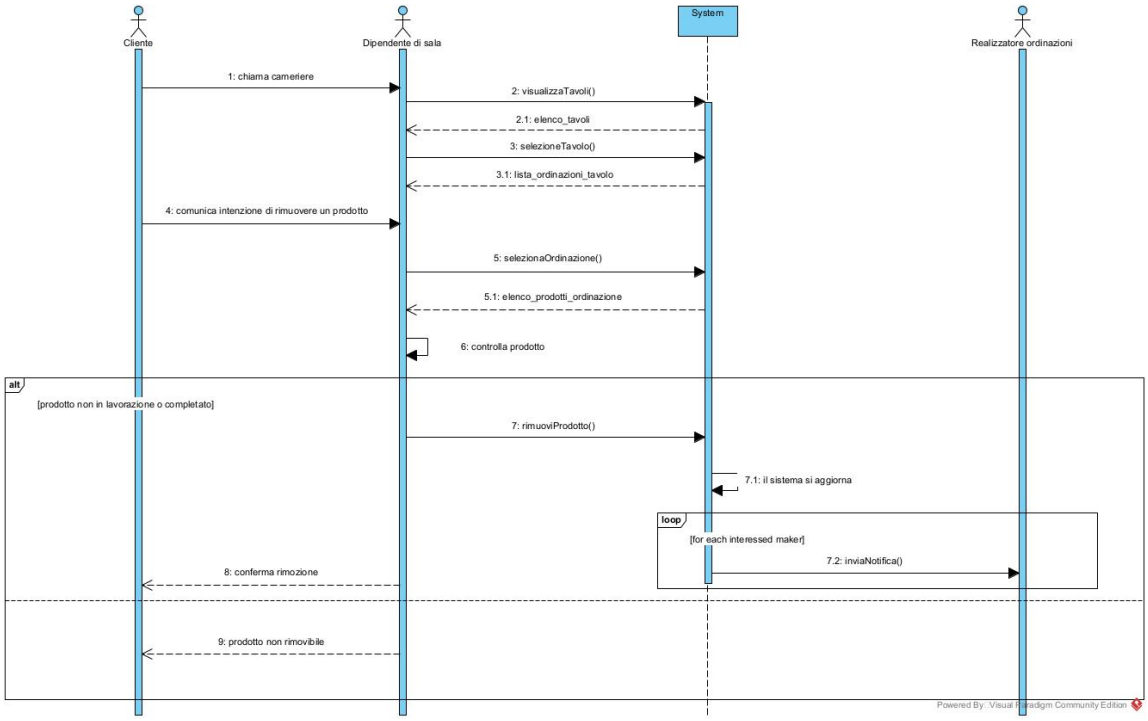
Estensione C



Estensione D

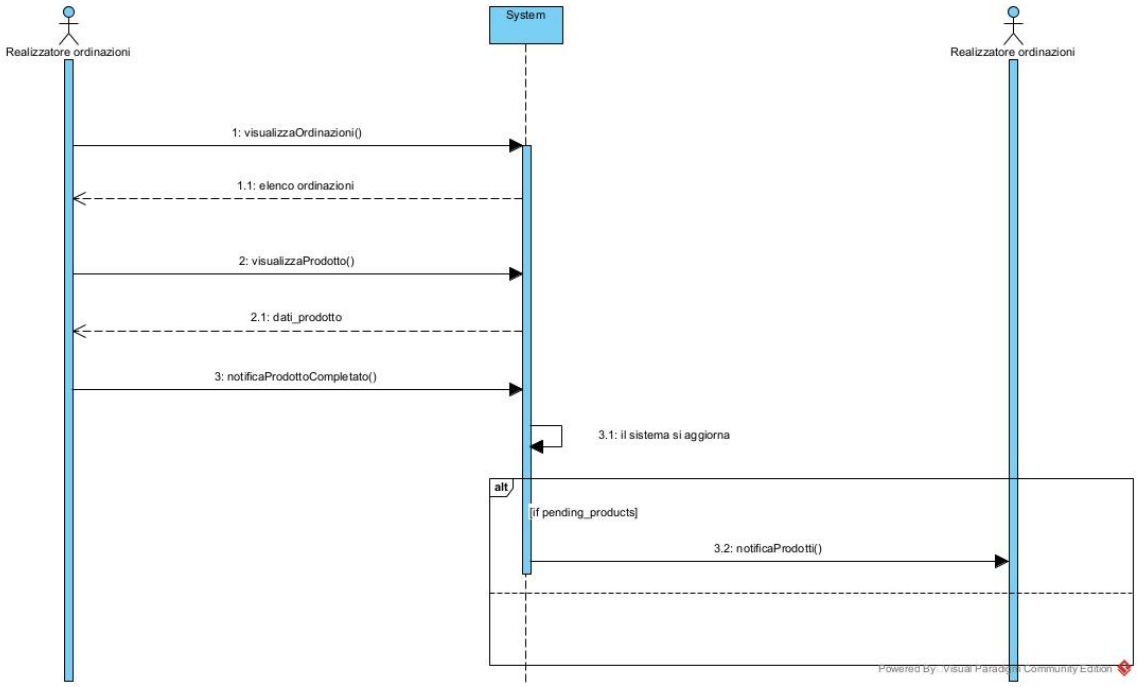


Estensione E



5.3.2 Notifica Ordinazione

Estensione A



Capitolo 6

Progettazione

La fase di progettazione prevede principalmente la scelta dell'architettura da usare, su cui costruire il sistema.
Dato il contesto di utilizzo dell'applicativo, la necessità principale è la sincronizzazione del lavoro dei dipendenti. L'obiettivo è quello di informare ogni dipendente del avvenimento di eventi a cui esso è interessato. Ogni azione deve essere memorizzata nel sistema, in conseguenza della quale esso provvede ad avvisare il gruppo di dipendenti relativo.
Ad esempio, al momento della registrazione di un'ordinazione da parte di un cameriere, il sistema provvede a notificare lo chef/pizzaiolo (realizzatore in generale) per informalo della nuova pietanza da preparare.
Per identificare l'architettura ideale bisogna elencare quali sono i componenti che ne fanno parte e in che modo sono interconnessi.

- I componenti sono di due tipologie:
  - Dispositivi utente, rappresentano i dispositivi fisici concessi ai dipendenti (smartphone o applicazione desktop)
  - Sistema centrale, racchiude la logica di gestione e a cui i dispositivi utente possono fare accesso.
- I connettori sono principalmente protocolli di rete, in quanto il sistema è pensato anche per dipendenti che hanno la possibilità di spostarsi all'interno del locale e accedere al sistema centrale in modalità remota.

Alla base di ciò i componenti devono ricevere dei cambiamenti del sistema tramite opportune *notifiche*, evitando quindi attese attive.

6.1 Architettura

Sulla base delle necessità descritte in precedenza, si è scelto di utilizzare uno stile architetturale del tipo **Publish-Subscribe**. Per la realizzazione dell'applicativo infatti vengono ripresi tutti i vantaggi dello stile a *invocazione implicita*:

- *Disaccoppiamento spaziale*: tutti i componenti sono altamente disaccoppiati, favorendo la scalabilità del sistema. Il proprietario può quindi assumere un numero di dipendenti variabile senza conseguenze.
- *Disaccoppiamento di sincronizzazione*: tutti i componenti non lavorano in "polling", ovvero in attese attive che possono bloccare il sistema.
- *Disaccoppiamento temporale*: tutti i componenti possono essere volatili nel sistema.

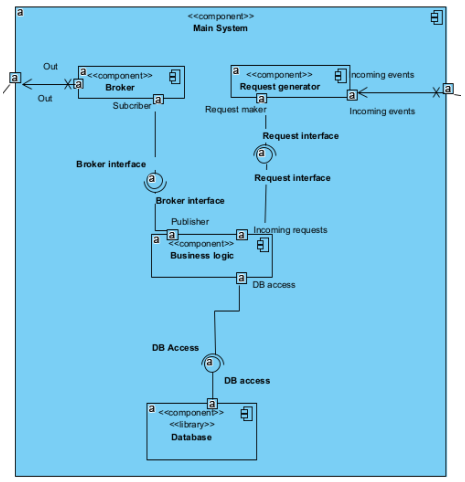
In relazione ai componenti descritti, tutti i dipendenti sono dei *Subscribers* mentre il sistema centrale è l'unico *Publisher*.
I dipendenti, effettuando l'accesso, si iscrivono in automatico al Publisher. Dato che il sistema può contenere un numero notevole di dipendenti, il Publisher fa uso di **Proxy** per smistare i messaggi solamente a specifici Subscribers.
Essi infatti all'atto del login non conoscono i Proxy del sistema, ma è il sistema stesso che li racchiude in gruppi (con lo stesso ruolo) e li associa a determinati Proxy. Tutto ciò si traduce in una distribuzione più efficiente del calcolo computazionale.
Riassumendo quindi:

Componenti	Publisher, Subscribers, Proxy
Connettori	Protocolli di rete
Dati	Notifiche, Richieste di informazioni, Informazioni pubblicate
Topologia	Subscribers sono connessi al Publisher indirettamente, ricevendo e inviando notifiche attraverso i Proxy
Vantaggi	Subscribers sono tutti disaccoppiati lavorando in parallelo. Le notifiche vengono ben distribuite grazie ai Proxy
Svantaggi	Il controllo computazionale è caricato tutto sul sistema centrale. Nessuna conoscenza di quali componenti risponderanno ad un eventi

6.1.1 Main System

Il sistema principale è l'unico Publisher dello stile utilizzato. Studiando i vari stili architetturali, tra quelli visti durante il corso, e prendendo spunto dallo stile a livelli,

l'architettura del sistema non si rispecchia in uno stile architetturale ben preciso, ma è caratterizzata nel seguente modo:

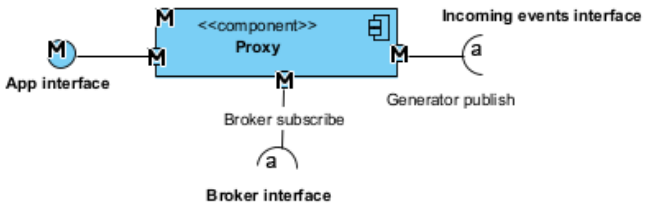


- **Database**: Al livello più basso, il componente, racchiude tutte le funzionalità per accedere e modificare i dati all'interno della base di dati del sistema. La business logic accede al database solo attraverso all'interfaccia che tale livello offre, mascherando la reale implementazione.
- **Business Logic**: Al livello centrale, il componente, racchiude tutte le funzionalità per manipolare le richieste e i dati del sistema. Effettua tutti i controlli necessari prima di inviare una risposta. Si occupa principalmente di implementare i casi d'uso.
- **Broker**: Al livello più alto il Broker si occupa solo di inviare i le risposte (prodotte dalla Business Logic) ai diretti interessati.
- **Request Generator**: Al livello più alto esso riceve, e identifica, le richieste dall'esterno e le inoltra alla Business Logic che si occupa di elaborarle e inviare le risposte al Broker.

La Business Logic deve esporre un'interfaccia al Request Generator che contiene le funzioni in relazione al caso d'uso da realizzare.
Tutto il componente viene visto dall'esterno (e quindi dai Proxy) come una blackbox con due sole interfacce: una per ricevere delle richieste e una per pubblicare gli eventi.

6.1.2 Proxy

Il proxy contiene un'insieme di utenti connessi al sistema tramite applicazione. Esistono più proxy, ognuno associato ad una categoria di utenti (i.e. Proxy Camerieri è associato a tutti i camerieri). Così facendo, all'atto dell'invio di un messaggio il Broker lo invia ad un relativo Proxy che provvede a smistarlo correttamente.



Oltre a inviare i messaggi a tutti i dispositivi associati, esso riceve anche informazioni solo da essi.
Esso è quindi dotato di tre interfacce:

- Un'interfaccia per collegarsi al Broker. Il Broker invia un evento da pubblicare, il proxy lo pubblica a tutti i suoi dispositivi.
- Un'interfaccia per collegarsi al Request Generator. Il proxy riceve dai dispositivi eventi di richiesta, il Request Generator riceve da esso l'evento da elaborare.
- Un'interfaccia per collegarsi all'applicazione mobile.

Attraverso l'analisi degli attori precedentemente descritta si hanno sei tipi di proxy:

- Proxy Cameriere associato a tutti i camerieri in servizio.
- Proxy Realizzatori associato a tutti i realizzatori di ordinazioni (chef, pizzaiolo, barista).
- Proxy Accoglienza associato all'addetto all'accoglienza.
- Proxy Cossiere associato al cassiere in servizio.
- Proxy Gestione associato al proprietario e all'economo.
- Proxy Login è un proxy generale a cui vengono associati tutti gli utenti che non hanno ancora effettuato l'accesso.

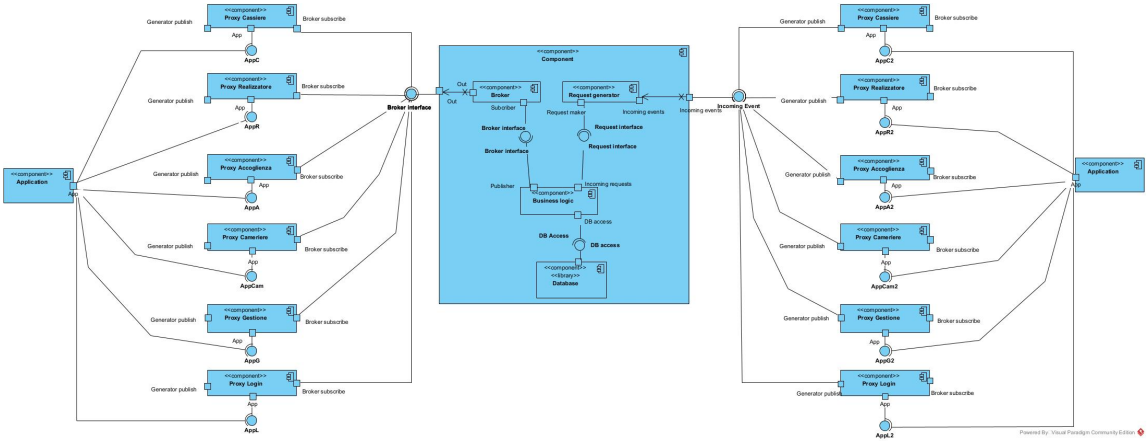
L'idea di base è che ogni utente, prima di effettuare l'accesso, può contattare solo il *Proxy Login*, il quale provvede in base alle credenziali ad associare il dispositivo al relativo Proxy.

6.1.3 Applicazione Mobile

Tutto il calcolo computazionale è a carico del *Main System*, mentre lo smistamento dei messaggi è a carico dei *Proxy*. L'applicazione deve solo ricevere e inviare eventi al proxy (o ai proxy, se più di uno) associato. La costruzione del messaggio avviene tramite un'interfaccia grafica per ogni contesto, il resto è gestito tutto dal Main System.
A tal proposito si è utilizzato il pattern Architetturale *MVC (Model View Control)* per la realizzazione dell'applicazione.

- *Modello* contiene tutti i dati necessari per il messaggio da inviare o ricevuto dall'esterno. Nella pratica il modello è una copia del package *Areas* del Main System. I dati infatti vengono scambiati tramite appositi messaggi utilizzato una serializzazione JSON.
- *Controllo* contiene tutte le funzionalità per interfacciarsi all'esterno tramite opportuni protocolli di rete per inviare e ricevere messaggi.
- *View* è la rappresentazione grafica dei dati.

6.2 Component Diagram



Per rendere più chiara la lettura del diagramma, i proxy sono stati replicati solo ad uno scopo grafico.
L'applicazione inoltre non è collegata realmente a tutti i proxy ma solo a quelli a cui corrispondono le sue credenziali di accesso; potenzialmente può essere collegata a tutti i proxy.

6.3 Class Diagram

Dopo aver illustrato il component diagram, si deve illustrare il modello ad oggetti di ogni componente presente nel sistema.

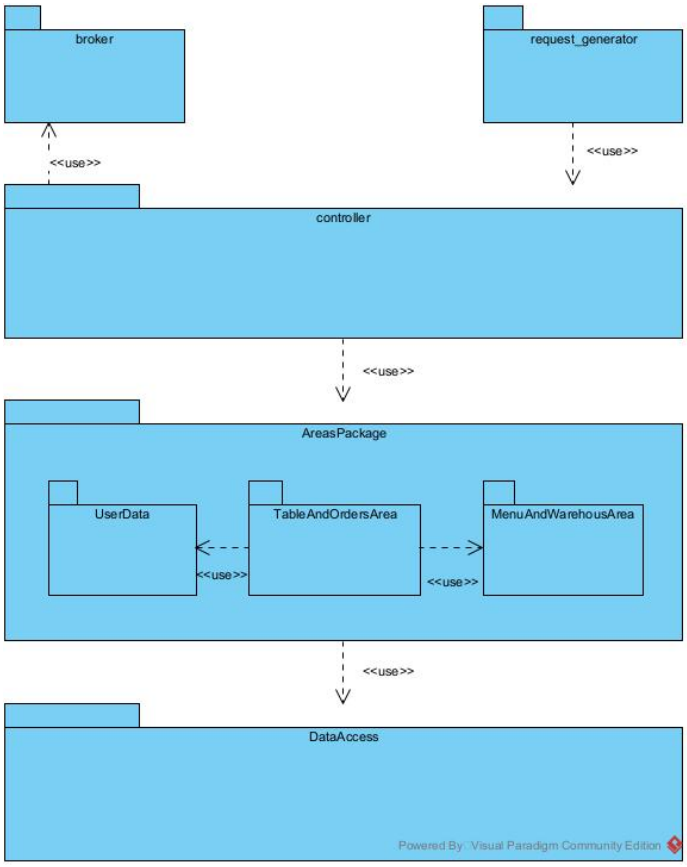
6.3.1 Main System

Il Main System è il componente più complesso perché contiene a sua volta altri componenti connessi tra loro, quindi non può essere rappresentato con un unico class diagram.
Nel livello inferiore è presente il database. Esso non può essere rappresentato come class diagram per la sua natura, inoltre non deve essere realizzato poiché ci si affida a database *relazionali* già esistenti.

Business Logic

Il class diagram è diviso in layer orizzontali, in cui ogni livello può interfacciarsi solo con il livello inferiore e può offrire un'interfaccia al livello superiore:

- **DataAccess**: livello più basso si occupa di implementare il driver per l'accesso al database.
- **Areas**: livello centrale mantiene il modello ad oggetti del sistema vero e proprio. Le aree interne comunicano tra loro solo attraverso i loro controllori, richiedendo e fornendo servizi. Un esempio è che, pur essendo la TableAndOrdersArea l'area responsabile di generare un ordine, sarà la MenuAndWareHouseArea (gestore del menù) responsabile di inizializzare il prodotto ordinato (dal prodotto del menu) e di ritornarlo all'ordine.
  Un altro esempio riguarda sempre la registrazione di un ordine presso un utente.
- **Controller**: livello più alto può accedere al modello inferiore e fornire un'interfaccia per un accesso controllato ai dati.



Areas: Il livello dei dati è a sua volta diviso in layer verticali. La costruzione di un'area segue sempre lo stesso schema:

1. Gli elementi di dominio che offrono operazioni per modificarne lo stato o per ottenere informazioni.

2. Un controller che permette l'accesso ai servizi del package ma consente anche agli elementi di dominio di richiedere servizi agli altri layer verticali.

Ogni livello si occupa di gestire specifiche macro-responsabilità:

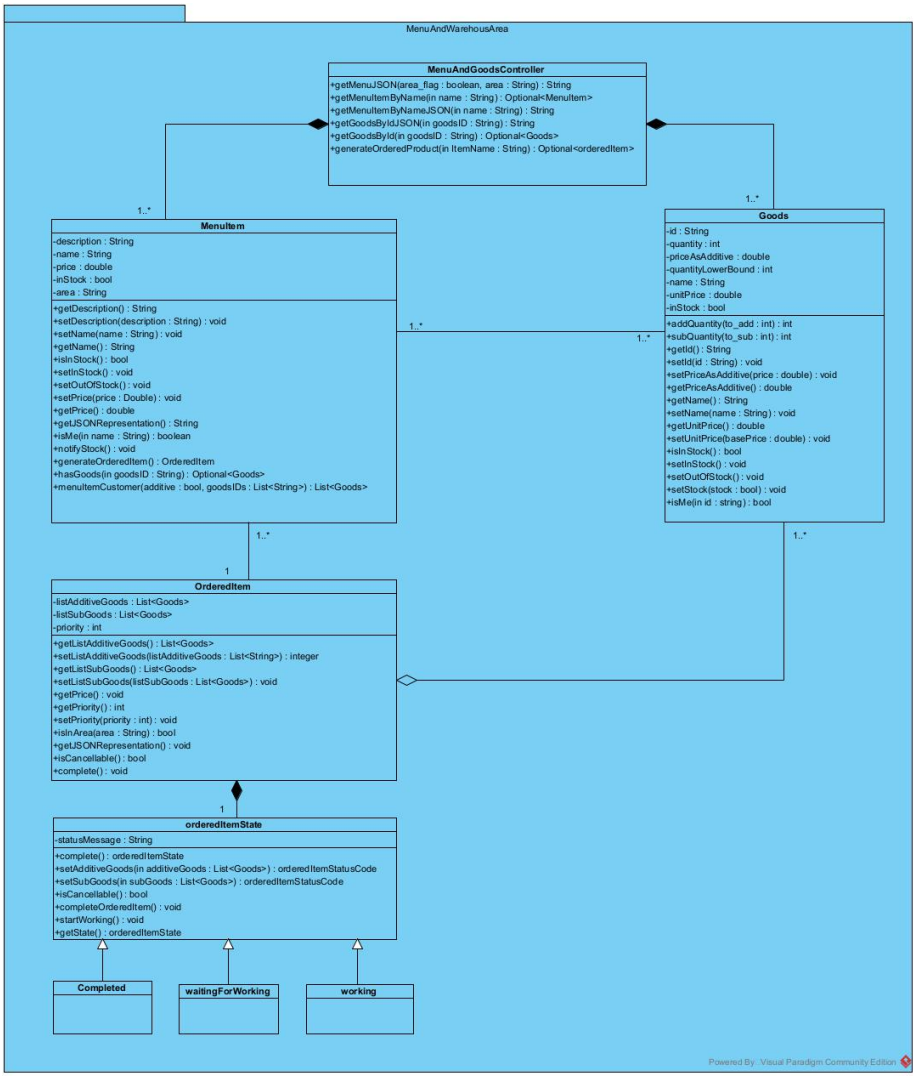
- *MenuAndWareHouseArea* si occupa di gestire le richieste di visualizzazione di prodotti sul menu, di merci, di aggiornamento della disponibilità dei prodotti in relazione alle merci ed alla creazione e personalizzazione di prodotti ordinati.

- *TableAndOrdersArea* si occupa di gestire lo stato dei tavoli, la creazione degli ordini ad un tavolo e la modifica degli ordini.

- *UserData* si occupa di verificare quali siano i ruoli di un utente fornendoli al richiedente, e di registrare gli ordini associati agli utenti.

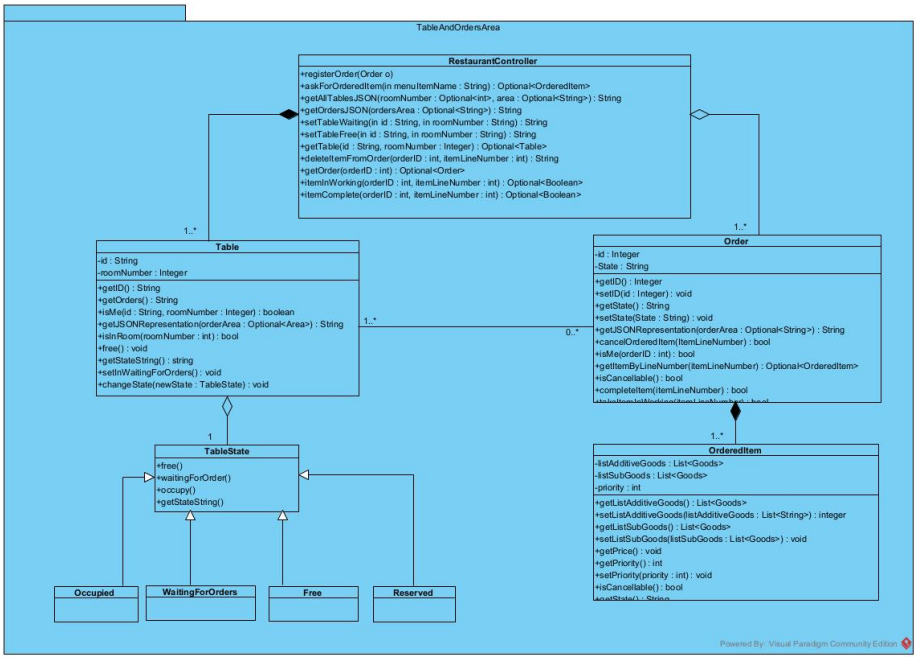
Nel dettaglio i tre package sono progettati nel seguente modo.

MenuAndWareHouseArea:



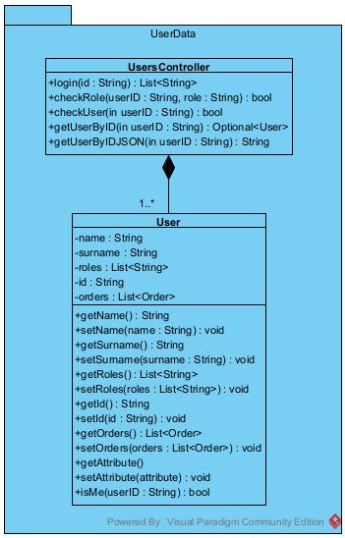
La classe *orderedItem* in linea con lo state chart descritto nei capitoli precedenti è stato progettato utilizzando il design patter State ed è associato a i prodotti del menu.

TableAndOrdersArea:

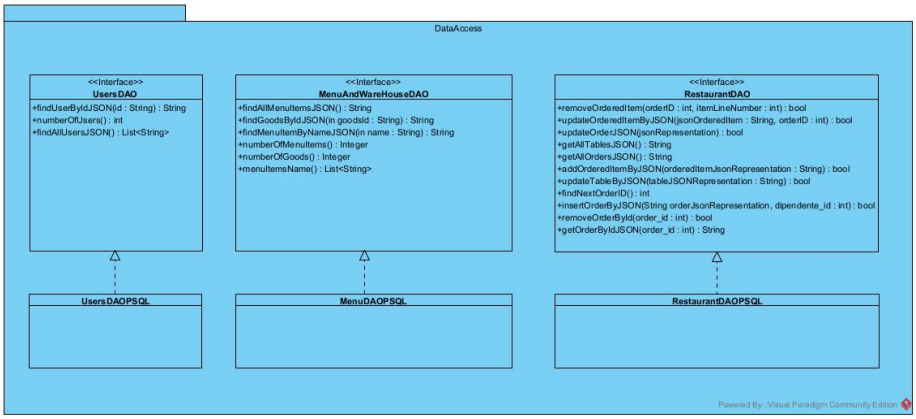


Anche in questo caso in linea con lo state chart il *Table* è stato progettato utilizzando il design patter State.

UserData:

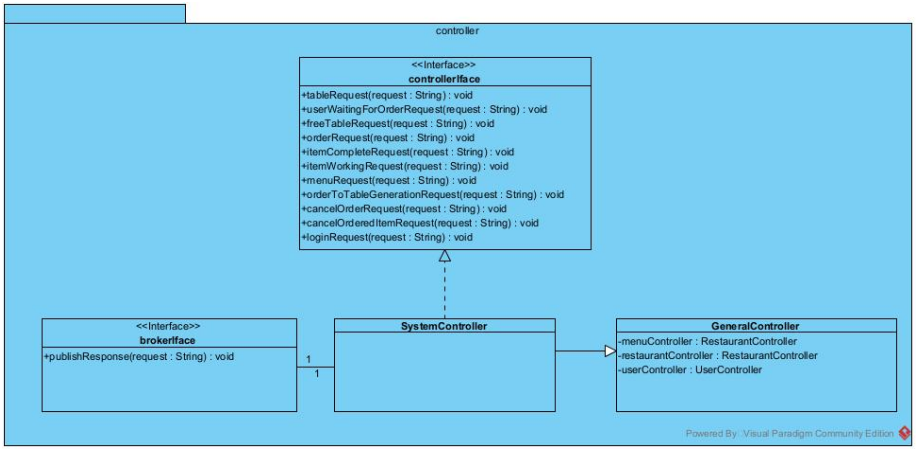


DataAccess:



Ad ogni layer verticale del livello superiore è associato una sola classe di accesso del livello inferiore. Così facendo ogni area può accedere solo alla parte di database interessata.

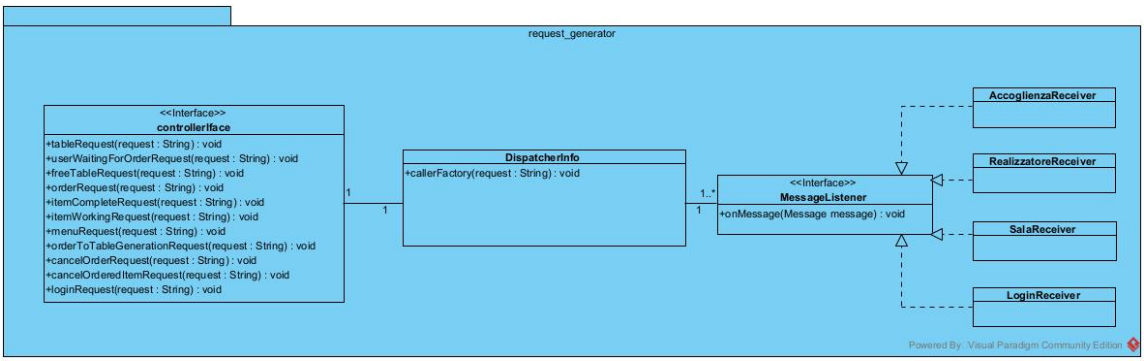
Controller:



Il *SystemController* offre tutte le funzioni di callback, per prelevare le informazioni necessarie dal modello. Inoltre definisce l'interfaccia che un Broker deve avere qualora voglia ricevere i nuovi eventi generati dalla parte centrale.

Request Generator

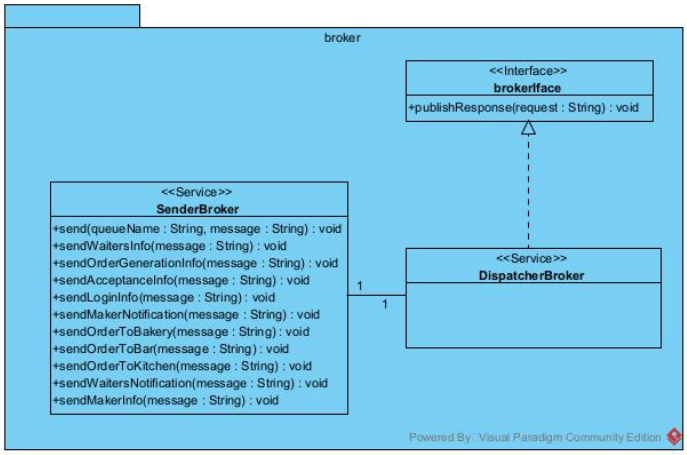
Si interfaccia con i *Proxy* per ricevere i messaggi e, attraverso un Dispacher, selezionare la funzione di callback del *Controller* giusta per la richiesta. Il Request Generator inoltre definisce l'interfaccia che un Controller deve avere, qualora voglia iscriversi a quest'ultimo come, dualmente, il Controller fa con il Broker.



La classe *DispatcherInfo* a seconda del messaggio ricevuto chiama una funzione di callback dell'interfaccia di controller.
Il *MessageListener* è l'interfaccia di un *Receiver*. Esistono Receiver diversi, poiché i messaggi in arrivo hanno come sorgente Proxy diversi; identicamente però richiamano un'unica funzione del *Dispatcher*.
Una seconda soluzione poteva essere di gestire un unico *Receiver* aumentando però la complessità di implementazione.

Broker

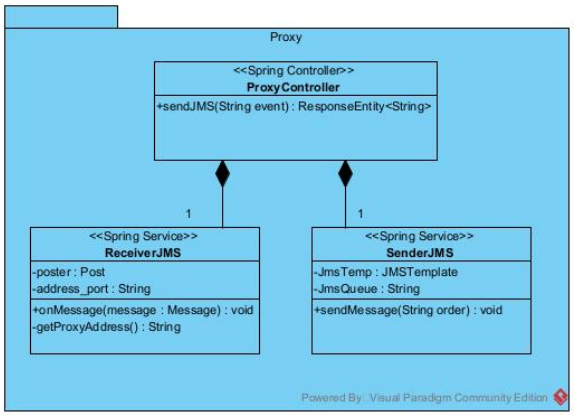
Si interfaccia con i *Proxy* per inviare i messaggi. Il broker come il duale del request generator.



Implementa l'interfaccia con i metodi di callback richiesti dal *SystemController*, a seconda della richiesta viene richiamata uno specifico metodo del *Sender*. I metodi del *Sender* possono essere di due tipi:

- Info: inviano un evento(richiesta) solo a coloro che hanno generato l'evento scatenante.
- Notification: inoltrano una notifica. Una notifica è un evento generato dalla Business Logic per informare tutti i Subcribers della verifica di un evento.

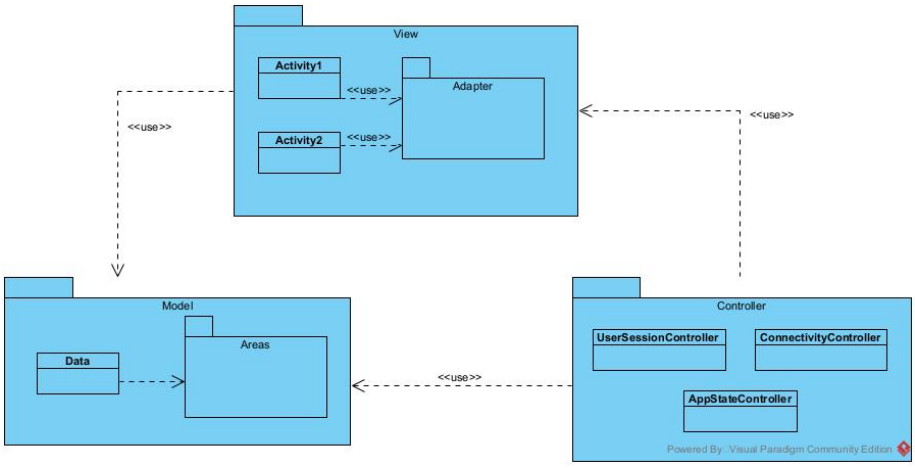
6.3.2 Proxy



Lo scopo del Proxy è quello di inviare/ricevere i messaggi a specifici gruppi di utenti. Si occupa quindi di interfacciarsi dualmente con il Main System (attraver il ReceiverJMS e SenderJMS), e di interfacciarsi con le applicazioni esterne.

6.3.3 Applicazione Mobile

In accordo con il pattern MVC quindi, ad alto livello:



Brevemente:

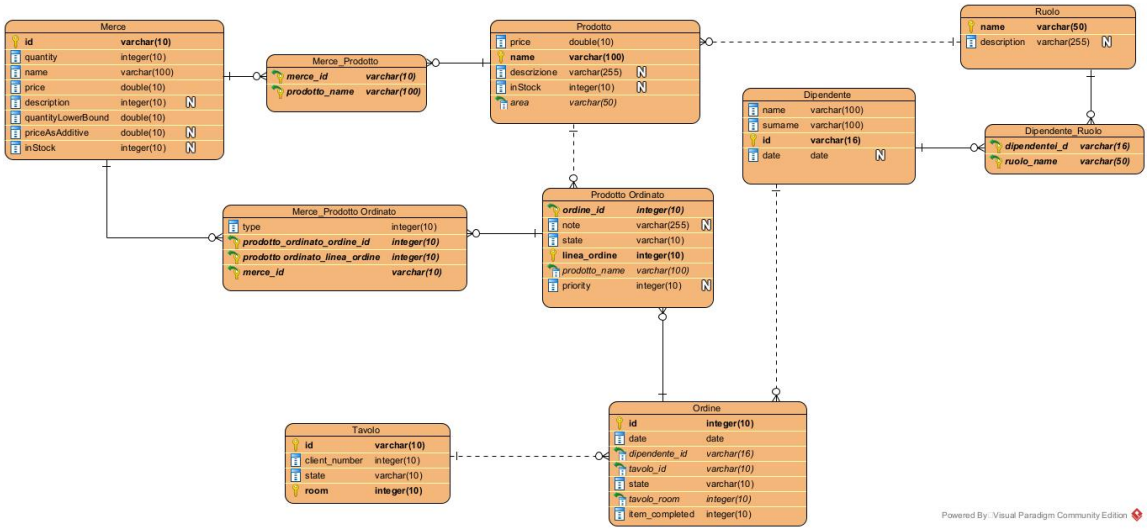
- Controller si occupa di gestire gli eventi da e per l'esterno, interagendo con l'utente che utilizza l'applicazione.
  - *AppStateController* è una classe realizzata con design pattern Singleton. Essa mantiene le informazioni riguardo allo stato dell'applicazione corrente (ad esempio l'activity in esecuzione).
  - *UserSessionController* mantiene le informazioni riguardo all'utente connesso. Il suo scopo è di identificare il ruolo dell'utente in un particolare contesto e associare il relativo indirizzo del proxy da contattare. Ad esempio se l'utente sceglie di aggiornare lo stato di un tavolo allora è sicuramente con il ruolo di *Addetto all'accoglienza*.
  - *ConnectivityController* è una classe realizzata con design pattern Singleton. Esso mantiene l'unico server in esecuzione che accoglie i messaggi esterni e interagisce con gli altri controller.

- Model si occupa di gestire tutti i dati utili all'applicazione. Il package Areas al suo interno è idealmente identico all'omonimo package nella *Business Logic*. In realtà però contiene solo classi e attributi che interessano ai fini dell'utilizzo. La classe **Data** è una classe con design pattern Singleton poiché all'interno del sistema deve esistere una sola istanza per gestire il modello.

- View si occupa di realizzare un'interfaccia grafica dei dati del modello.

6.4 Database

Il database deve contenere le informazioni di base che devono essere utilizzate, a partire dai dipendenti che vengono registrati nel sistema e finire con le ordinazioni che vengono effettuate durante l'utilizzo del sistema.



Le caratteristiche principali del database sono le seguenti:

1. Le merci hanno, oltre alle informazioni di base quali prezzo, codice identificativo, nome, etc, hanno un attributo *priceAsAdditive*. Tale attributo, se non nullo, indica il prezzo da aggiungere se quella merce è usata come merce aggiuntiva di un prodotto.

2. Il *Prodotto Ordinato* è un'associazione ternaria tra *Ordine*, *Prodotto* e *Merce*.

3. Ogni prodotto ha un attributo collegato all'entità *Ruoli*. Esso serve per indicare a che attività appartiene quel prodotto.

Il precedente database si riferisce solo ai casi d'uso che si sono scelti di implementare. Non contiene, ad esempio, la gestione delle vendite e lo storico del locale.

6.5 Sequence Diagram

I Sequence Diagrams inerenti il Main System sono stati organizzati secondo diversi livelli di dettaglio con una logica a "decomposizione".
Per una apposita lettura ovviamente i Sequence di uno specifico livello di dettaglio iniziano con un determinato prefisso.

1. I Sequence top rappresentano il funzionamento dell'intera applicazione, dall'arrivo di un messaggio al Request Generator fino all'arrivo della richiesta al Broker. //I sequence top richiamano operazioni offerte dai Controller delle tre aree verticali.
   Il loro prefisso è: "top".

2. controllersSequence sono i Sequence dei Controller verticali dell'Areas. Il loro prefisso è dato dal nome dell'area con la lettera minuscola.

3. menuAndWareHouseArea: Sequence delle operazioni del Controller dell'area menu e merci.

4. tablesAndOrdersArea: Sequence dell'area del Controller dell'area ristorante(tavoli ed ordini).

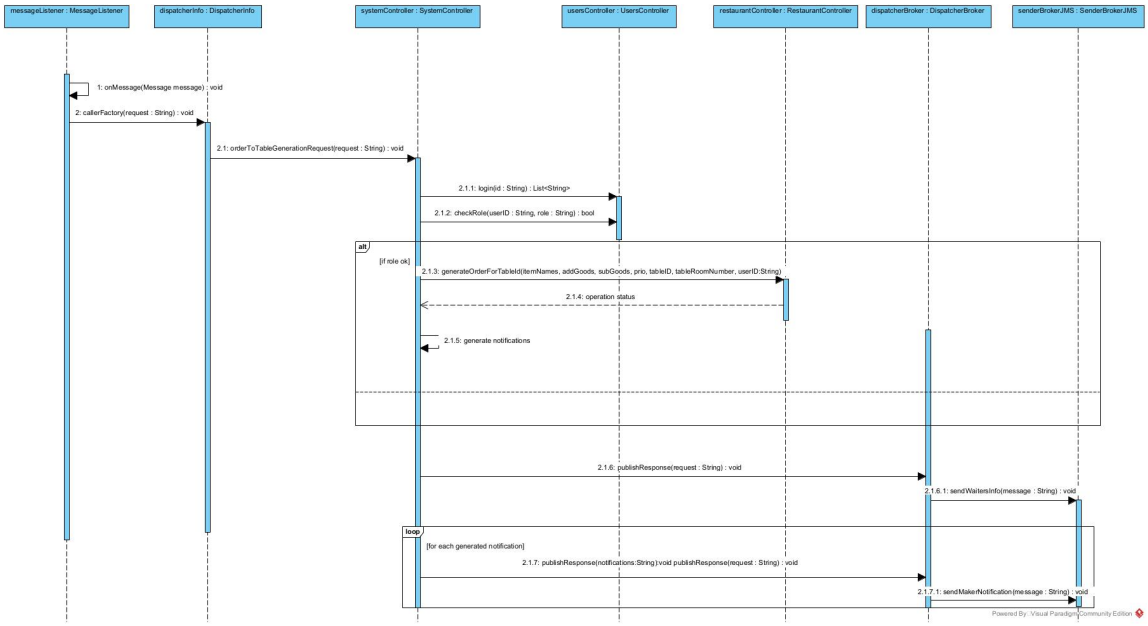
5. userData: Sequence del controller area utente.

Sono stati infine forniti Sequence inerenti il funzionament dei proxy
I sequence digram mostrati successivamente sono diagrammi di dettaglio che non mostrano la realizzazione delle chiamate a funzioni di livello più basso. Per esse si rimanda al progetto in Visual Paradigm.

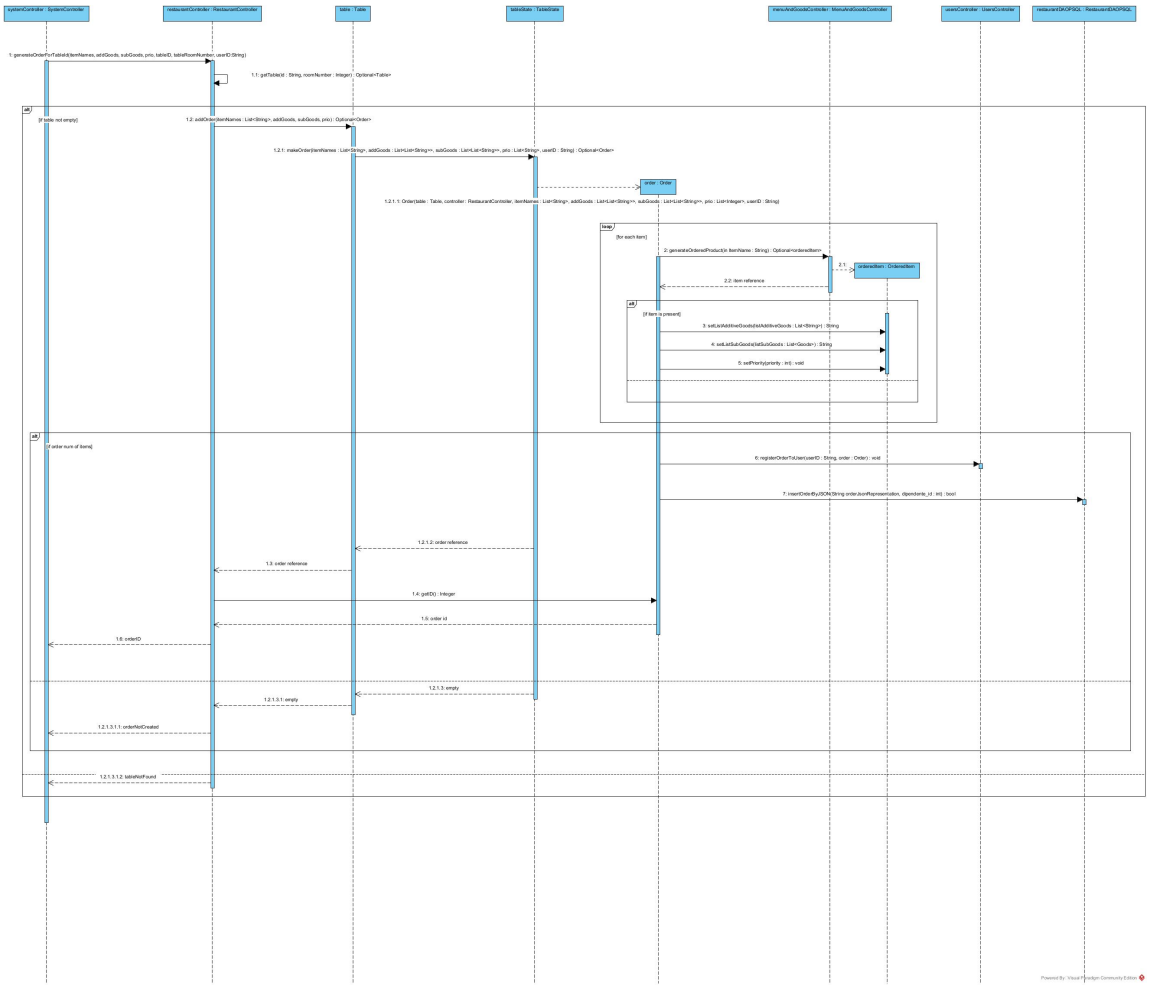
6.5.1 Creazione di un'ordine

Rappresenta il generico meccanismo di generazione di svolgimento di una operazione con generazione di notifiche. In genere ogni evento ha suoi particolari metodi per andare a generare le notifiche (metodi di utilita' generateNotifications, non riportati per non appesantire inutilmente i diagrammi).

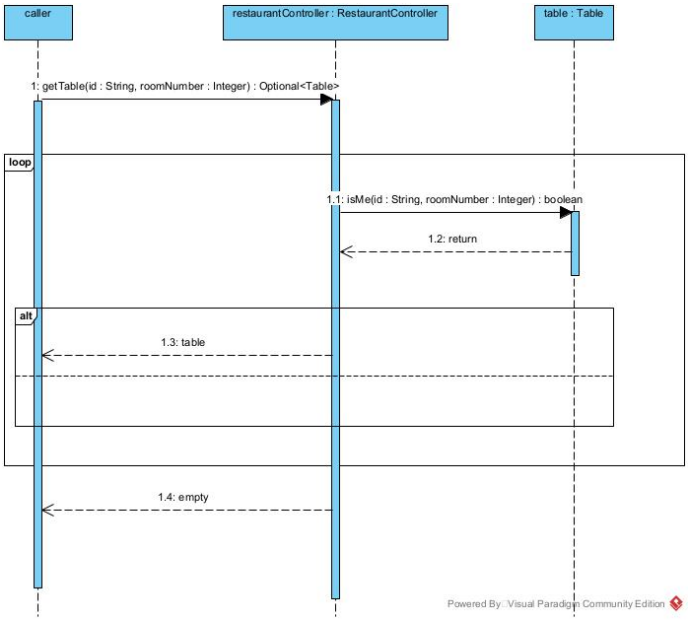
In questo caso specifico ovviamente le notifiche non sono altro che le comande. In relazione alla decomposizione descritta precedentemente, per la creazione di un'ordine il SD *top*:



Il Sequence precedente utilizza il metodo generateOrderForTableId, il cui SD è il seguente:



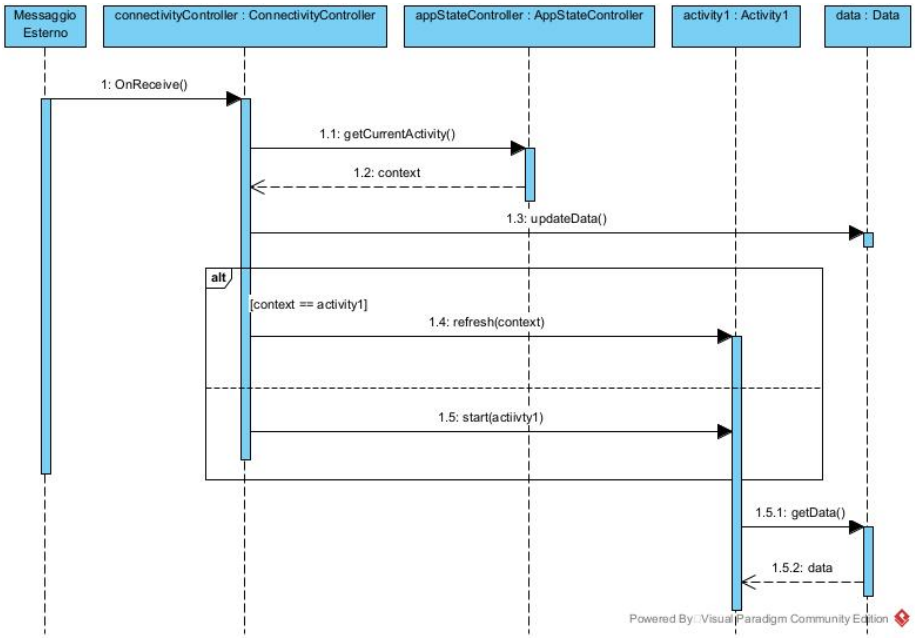
A sua volta esso richiama il metodo getTable, il cui SSD:



Analogamente tutti i SD vengono letti nello stesso modo.

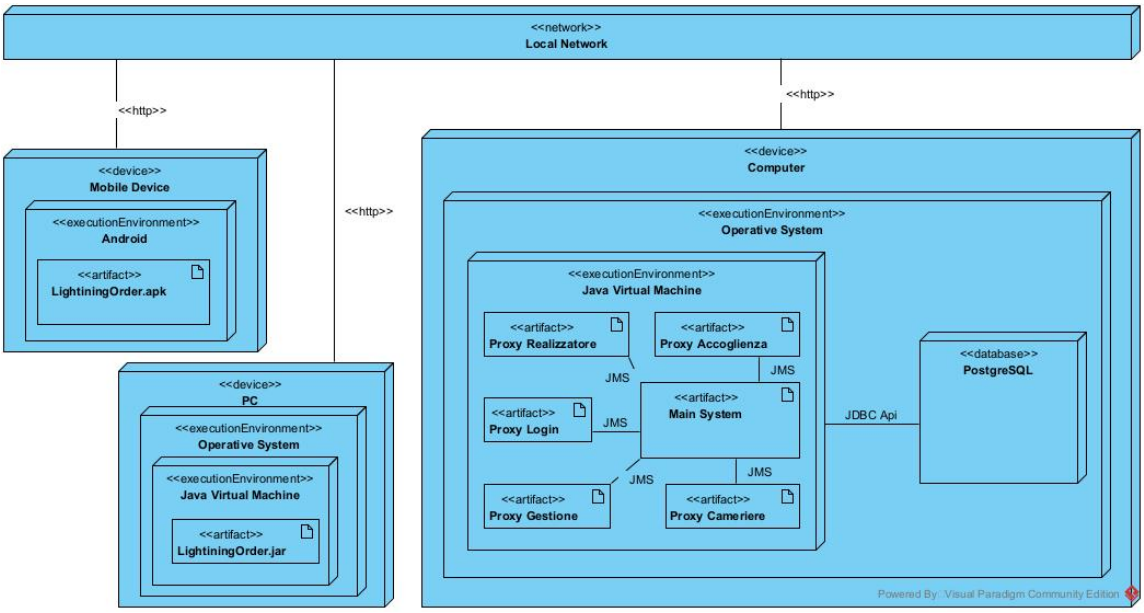
6.5.2 Aggiornamento View

Lato Applicazione Mobile il seguente SD mostra l'interazione nell'MVC per aggiornare la View alla ricezione di un messaggio che contiene un update del modello.



In linea generale il funzionamento è il precedente, ma non è rigidamente rispettato per tutti i possibili casi. Ad esempio in alcuni casi non è necessario richiamare una view, ma solo far visualizzare a video una notifica di aggiornamento.

6.6 Deployment Diagram



I Proxy sono progettati per comunicare tramite JMS quindi non richiedono che siano avviati nella stessa macchina. Essi possono essere collegati anche su altri *device* e poi connessi in rete. Per motivi di mezzi a disposizione si è scelto di avviare i proxy sulla stessa macchina del Main System.
Analogo discorso vale per il database.

6.7 Messaggi

Nel sistema vengono scambiati due tipi di eventi:

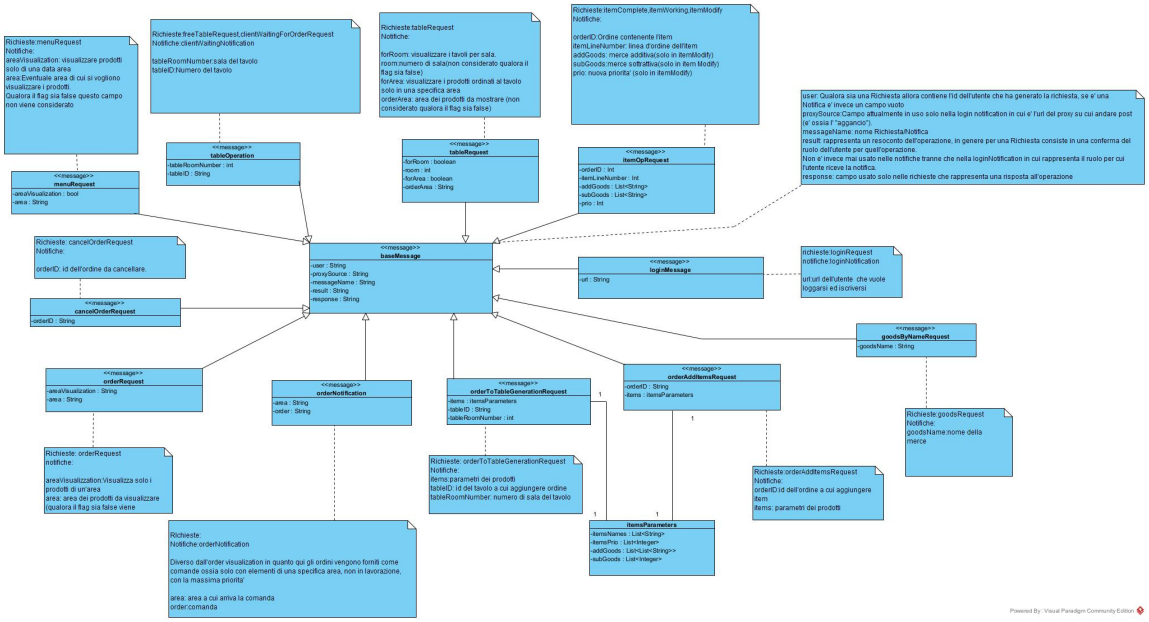
- Richieste.
- Notifiche.

Le richieste sono eventi che spingono l'applicazione centrale a cambiare stato.
Quando un utente si siede ad un tavolo, ad esempio, l'addetto all'accoglienza genera una richiesta. L'applicazione centrale, notificata dal Proxy, va a cambiare lo stato del tavolo e produce un "messaggio di risposta".
Un messaggio di risposta ad una richiesta arriva sempre mediante chiamata indiretta

e riassume il cambiamento dello stato dell'applciazione centrale a seguito dell'evento richiesta. Nell'esempio precedente all'addetto all'accoglienza, dopo aver generato un evento, sempre in maniera indiretta gli arriva un report che riporta il nuovo stato del tavolo (sempre in maniera indiretta tramite *Webhook*). Per comodità dunque si indicano come richieste sia gli eventi generati dall'applicazione esterna sia i "messaggi di risposta" alle richieste . Nonostante i nomi, come già accennato, questi funzionano mediante un meccanismo di invocazione indiretta, sono di fatto delle vere e proprie **notifiche** (Al sistema centrale viene notificato che un cliente si e' seduto ed all'addetto viene notificata l'avvenuto cambio di stato).
Si riconsideri ora l'esempio: Una volta che il cliente si è seduto e che lo stato è cambiato, è necessario notificare ai camerieri che devono andare a prendersi l'ordinazione. Chi deve pero' generare questa notifica? Secondo lo stile adottato l'applicazione ha originalmente generato un evento gestito solo dall'applicazione centrale che va dunque anche l'onere di notificare i camerieri. E' dunque compito dell'applicazione centrale andare a generare degli eventi che chiamiamo notifiche.
Nell'esempio l'applicazione centrale oltre alla risposta alla richiesta (inoltrato al proxy accoglienza) genera una notifica che, inoltrata al proxy camerieri, viene inviata in broadcast, notificando di fatto i camerieri.
Si puo' quindi riassuntivamente dire che:

- Richieste: Sono gli eventi generati dagli utenti tramite l'applicazione.
- Notifiche: Sono gli eventi generati dall'applicazione centrale in seguito ad una richiesta.

Richieste e notifiche vengono inviate tramite appositi messaggi. Un messaggio funge da "stampino" per una richiesta/notifica, ossia gli offre un insieme di campi base. Il significato di quei campi dipende pero' dalla specifica richiesta/notifica.



6.7.1 Liste di Richieste

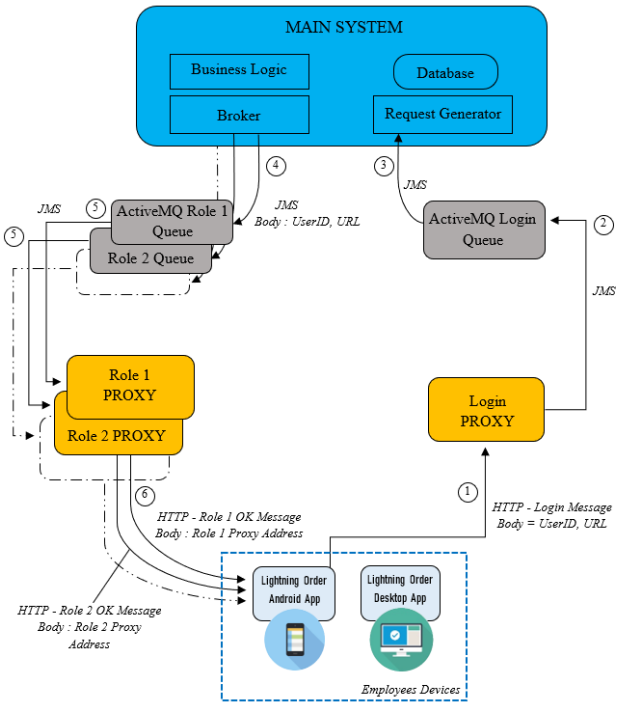
menuRequest: Richiedi una lista di menu
freeTableRequest: Un tavolo si e' liberato
clientWaitingForOrderRequest: dei clienti sono stati fatti accomodare
tableRequest: visualizzare tavoli ed ordini connessi(usata sia da camerieri che da accoglienza)
itemComplete: un realizzatore ha terminato un prodotto ordinato(il pizzaiolo ha finito la pizza)
itemWorking: un realizzatore ha iniziato a lavorare ad un prodotto
itemModify: un cameriere su richiesta del cliente modifica il prodotto
cancelOrderRequest: cancellare un ordine
orderRequest: un realizzatore vuole visualizzare la lista di ordini presenti nel locale
orderToTableGenerationRequest: Un cameriere genera un ordine ad un tavolo
orderAddItemsRequest:aggiungi dei prodotti ad un ordine
goodsRequest: visualizza della merce per uno specifico nome
loginRequest: un utente desidera loggarsi ricevendo la sua lista di ruoli

6.7.2 Lista di Notifiche

clientWaitingNotification: un cliente e seduto al tavolo attenenendo di compiere la prima ordinazione
orderNotification: notifica generata in seguito all'aggiunta di un ordine, al completamento di tutti gli ordini con una data priorita' o all'aggiunta di nuovi ordini. Coincide di fatto con una comanda.
loginNotification: notifica che un proxy invia all'utente per notificarlo della sua avveuta sottoscrizione (contiene l'indirizzo del proxy su cui andare po ad effettuare post).

6.7.3 Scenario di comunicazione

Nell'esempio sotto riportato viene rappresentata la comunicazione tra applicazione mobile e Main System attraverso il messaggio *loginRequest*.



Il contenuto informativo è rappresentato da stringe di tipo JSON.

Capitolo 7

Implementazione

La parte implementativa riguarda il processo di implementazione, le tecniche e i framework utilizzati durante lo sviluppo sia del Main System e del database backend, sia dell'infrastruttura di comunicazione dei Proxy e sia dell'interfaccia client dell'applicazione Android.

7.1 Database

Per le nostre esigenze, vi era la necessità di una base di dati sviluppabile in breve tempo e con un DBMS documentato e facile col quale comunicare. Sulla base di scelte implementative fatte anche nel Main System, la scelta è ricaduta su **PostgreSQL**. PostgreSQL è un database relazionale, la cui progettazione è stata semplificata grazie alle conoscenze preliminari acquisite in altri corsi. Implementativamente è stato definito uno *schema* che racchiude tutte le tabelle necessarie per la memorizzazione dei dati, chiamato *Restaurant*. Inoltre il modello relazionale non è del tutto identico al modello degli oggetti del Main System; questo però non provoca problemi di implementazione data la natura ben chiara delle informazioni.

All'interno del package *DataAccess* della Business Logic del Main System sono state definite tutte le query per prelevare le informazioni dalla base di dati. In tal modo l'accesso al database è del tutto mascherato e controllato. Ogni elemento del package *Areas* può accedere solo a determinate aree del database, mascherando tutti i dati memorizzati. Ad esempio l'area utente può accedere solo ad informazioni relative agli utenti, e non quelle relative al menù o alle merci a disposizione.

7.2 Main System

Il Main System è il cuore dell'applicazione, in quanto ne è il cervello pensante. L'implementazione di questa parte conta diverse centinaia di linee di codice, poiché gestisce tutte le meccaniche di creazione, eliminazione e aggiornamento di un evento relativo a tutti i possibili ruoli. Lo sviluppo, in concomitanza con i test, ha richiesto un'analisi attenta e precisa della modularità del codice.
In particolare, la scelta di un framework da utilizzare si è rivelata fondamentale per gestire opportunamento lo sviluppo. La scelta è ricaduta sul framework **Spring Boot** versione 2.5.5, versione di Java 11 e Maven per la gestione delle dipendenze. Tale decisione è dovuta alle meccaniche di injection fornite da Spring, che hanno reso estremamente semplice e scalabile la comunicazione con PostgreSQL.
Spring fornisce infatti diverse annotazioni estremamente veloci ed efficaci quali @Autowire per effettuare la dependency injection di oggetti che possono essere servizi, controller o semplici bean di configurazione. Ciò ha reso non solo di più semplice lettura il codice, ma ci ha anche notevolmente risparmiato il tempo di dover implementare le meccaniche di comunicazione fra classi sia interne sia esterne ai package implementati.

Il Main System utilizza queste facilitazioni definendosi di base la logica di controllo e poi un intreccio di controller organizzati gerarchicamente: un usersController, un restaurantController ed un menuAndWareHouseController, tutti organizzati sotto un unico GeneralController, esteso dal SystemController che ha il compito di interfacciare il sistema con il **Request Generator** e con il **Broker**. Ogni controller di base si occupa poi della comunicazione col database per le informazioni di proprio interesse, scambiandole con l'esterno attraverso il System Controller.

7.2.1 Request Generator

Il Request Generator è il componente che si interfaccia coi proxy adibiti alla ricezione di richieste da parte dell'applicazione.
L'interfaccia sviluppata di basa su due componenti principali:

- Il **JMS**, versione 2.0;
- **ActiveMQ Artemis**, un MOM configurabile in maniera particolarmente rapida: basta usare un comando per dichiararsi un broker ed un altro per avviarlo e subito si può iniziare a sperimentare.

I messaggi scambiati sono tutti in formato JSON. La progettazione rende di fatto già scalabile la comunicazione, di conseguenza il Request Generator ha il solo compito di fornire l'interfaccia verso Artemis attraverso un controller che prenda in carico, attraverso il JMS, i messaggi che sulle diverse code vengono prodotti.

Anche per il Request Generator è stato usato Spring Boot. In particolare, di fondamentale importanza è stata la dependency *Spring for Apache ActiveMQ 5*. Essa mette a disposizione l'infrastruttura necessaria alla comunicazione coi server Artemis utilizzabile attraverso l'interfaccia, da estendere, MessageListener. E' per questo motivo che Request Generator implementa diversi controller a seconda della coda di messaggi: ogni controller modifica il metodo onMessage() per gestire opportunamente il messaggio JSON, la cui struttura può cambiare a seconda della coda sul quale si trova, che riceve. Il dispatcher, tramite il metodo *caller factory*, va a richiamare una specifica operazione di callback della *controlInterface* definita dallo stesso Request, che è un controller deve implementare affinchè possa essere usato insieme a quest'ultimo.

7.2.2 Broker

Il Broker ha il compito esattamente opposto al Request Generator: smistare i messaggi sulle code in maniera tale da informare l'applicazione utente degli eventi ai quale essa è interessata. La struttura, poiché ricettiva, riceve i messaggi da pubblicare attraverso il System Controller che implementa un'interfaccia, la BrokerInterface, che poi il Dispatcher estende.

Anche il Broker utilizza il JMS e ActiveMQ Artemis per comunicare con l'applicazione attraverso i proxy. La differenza principale col Request Generator è che il Broker ha bisogno di aprire il messaggio JSON inviato dall'applicazione per smistarlo correttamente sulla base del codice nominativo dell'area, di fatto codice univoco per la coda su cui smistare.
Il broker dualmente al Request Generator possiede un dispatcher che implementa l'interfaccia *BrokerIface* definita dalla Business Logic. Il dispatcher a seconda dell'evento pubblicato va a notificare uno specifico Proxy.

7.3 Proxy

I proxy sono l'impalcatura su cui si fonda la rete di messaggi tra applicazione e sistema.

Il broker dualmente al request generator possiede un dispatcher che implementa l'interfaccia BrokerIface definita dalla Business Logic. Il dispatcher a seconda dell'evento pubblicato va a notificare uno specifico proxy. Ciò ha concesso un ulteriore livello di disaccoppiamento tra applicazione e Main System.

Anche l'implementazione dei proxy è basata su Spring Boot. Le dependency utilizzate sono **Spring Web** e **Spring for Apache ActiveMQ Artemis**.
I proxy hanno infatti sia il compito di fungere da client per il Request Generator sia quello di fungere da server per il Broker. Di conseguenza, la prima dependency di Spring mette a disposizione l'infrastruttura generale per aprire un server Tomcat in ascolto in localhost di default sul porto 8080, configurazione che può essere modificata attraverso il file di proprietà.
Vi sono diversi proxy, ma tutti in generale hanno la stessa struttura ed implementazione che differisce solo sulla base dei messaggi che devono leggere e scambiare.

Per la gestione dei messaggi HTTP, i proxy utilizzano dei controller specifici che si preoccupano di istanziare un metodo wrapper per le POST effettuate al filepath indicato. La sintassi è la seguente:

- @PostMapping(valude= filepath)

Attraverso quest'annotazione, è possibile gestire in ricezione le richieste POST che arrivano dall'applicazione al filepath indicato. In questo modo, ogni proxy deve solo preoccuparsi, in ricezione, di girare poi il messaggio sulla coda corretta, attraverso una convertAndSend, cioè un metodo della classe JmsTemplate fornita da Spring nella libreria *org.springframework.jms.core.JmsTemplate*.

In fase di ricezione, come per il Request Generator, ogni proxy deve invece mettersi in ascolto sui messaggi delle codce che il Broker riempie. Il lavoro che in questo caso il proxy deve effettuare è più delicato:

1. Deve spacchettare il messaggio JSON per comprendere di che tipo di evento si tratti;
2. Se si tratta di un messaggio di registrazione, deve registrare l'uri dell'applicazione al proprio Webhook e rispedire il messaggio al destinatario corretto specificando il proprio uri di destinazione;
3. Se si tratta di un messaggio da spedire ad un'applicazione, suppone che quest'ultima già conosca il proprio uri di destinazione e si preoccupa solo di verificare che essa sia registrata al proprio Webhook;

- In tal caso inoltra al singolo utente una Richiesta o una Notifica in broadcast.

4. Impacchetta il messaggio nuovamente in un JSON ed invia.

L'invio è gestito sempre attraverso richieste POST all'indirizzo contenuto nella lista Webhook attraverso funzioni di utilità che fanno utilizzo della libreria Spring *org.springframework.web.client.RestTemplate*.

Capitolo 8

Test

Vista la natura disaccoppiata dei layer verticali del sistema (che offrono e forniscono servizi) è stato molto facile andare ad effettuare, subito dopo lo sviluppo di ogni layer, un test per verificarne il corretto funzionamento. Si è dunque visto un susseguirsi di sviluppo di layer ed integrazione con test.
I test, eseguiti con **Junit5**, sono tuttora presenti e mirano a verificare il corretto cambiamento dello stato dell'applicazione a seguito di un cambiamento di un metodo o dell'aggiunta di un'operazione.
Qualora i test necessitino di dati persistenti è stata fornita la versione del db usata in un file *.sql* nella cartella test.
I test non hanno avuto il solo scopo di verificare il codice fino ad un determinato punto dello sviluppo ma servono anche per verificare il nuovo codice scritto. Alla modifica di ogni funzione di un Controller è necessario infatti rieseguire il test associato ad esso per verificarne il corretto funzionamento.
All'aggiunta di un'operazione ad un controller si forniscono dunque appositi test associati che vanno ad ampliare la suit di test che il ontroller dovrà sempre superare.

8.1 Test di Integrazione

In seguito all'integrazione con il Request Generator e il Broker, è stato possibile andare a testare l'applicazioni simulando delle richieste inviate ai Proxy.
In particolare è stato usato **Postman** per andare ad inviare gli specifici messaggi *Json* (simulando un applicazione utente) e si sono controllate risposte e notifiche arrivate ai Proxy.
L'api di test è stata fornita come esempio di funzionamento nell'applicazione nel codice del "backend".