Applicativo Java per la gestione di un ristorante

Elaborato Ingegneria del Software

Gianni Moretti, Francesco Fantechi, Francesco Bettazzi

A.A. 2020-2021

UNIVERSITA' DEGLI STUDI DI FIRENZE
Facolta di Ingegneria
Corso di Laurea in Ingegneria Informatica

Contents

1 Motivazione e Descrizione

L'idea di questo elaborato nasce da uno di noi che lavorando in una pizzeria come cameriere si è trovato ad interagire con uno di questi applicativi. Abbiamo cosí pensato di poterne riprodurre uno personalizzato che preveda inoltre, considerato il periodo particolare che stiamo vivendo, la possibilità di gestire azioni atte alla sicurezza dei clienti come il loro monitoraggio per poterli rintracciare.
L'applicativo ha lo scopo di gestire le varie parti che compongono un ristorante, in modo da farle interagire e collaborare assieme. Per il nostro applicativo abbiamo individuato cinque figure professionali principali che possono trovarsi ad agire all'interno di un ristorante:

1. L'organizzatore della sala è colui che all'arrivo di nuovi clienti ha il compito di assegnare loro un tavolo. Ció puó essere realizzato utilizzando i tavoli singoli giá disposti in sala ad inizio serata o aggregandone alcuni se il numero di clienti é elevato. Ha quindi la possibilitá di contrassegnare i tavoli assegnati come occupati e di rimetterli disponibili una volta terminato il servizio e igenizzato il tavolo. Inoltre, in vista delle norme di distanziamento imposte dal periodo che stiamo vivendo, l'organizzatore della sala puó decidere se e quali tavoli sono o no utilizzabili.

2. Il cameriere ha il compito di gestire i vari servizi ai tavoli, ossia di prendere le comande e mandarle alla cucina per la loro realizzazione. Le comande si compongono da un insieme di piatti presenti nel menú del locale, da dei commenti opzionali sui piatti per la cucina e da un insieme di ingredienti aggiunti/rimossi dalle varie portate su preferenza e richiesta del cliente. In caso di errore di immissione della comanda o di richiesta di modifica da parte del cliente, il cameriere puó eliminare le portate sbagliate in modo da correggere correttamente il conto. Per aprire un servizio al tavolo il cameriere é tenuto a prendere un nominativo ed un recapito telfonico di uno dei presenti in modo da tenerne traccia di chi ha frequentato il locale ed essere in grado di rintracciare i clienti in caso di necessitá.

3. La cucina e quindi i cuochi ottengono le comande confermate dai camerieri ai vari tavoli e, una volta preparate, possono spuntarle come giá fatte. La cucina si puó inoltre specializzare in piú aree di competenza come ad esempio la pizzeria e il bar.

4. Il cassiere é colui che al termine del servizo a un tavolo esegue il conto, contrassegna il tavolo come da pulire ed igenizzare e si occupa di memorizzare correttamente i dati dei clienti presenti.

5. Il gestore del ristorante é colui che ha accesso ai dati memorizzati dei clienti. Puó inoltre modificare il menú del locale aggiungendo portate e variando gli ingredienti in esse presenti al fine per esempio di aggiungere una specialitá del giorno.

1.1 Possibili aggiunte

Delle possibili aggiunte non implementate potrebbero essere quelle di consentire al cliete di fare degli ordini da casa attraverso un sistema di consegna a domicilio (Es: Delivero) oppure quella di consenire all'organizzatore della sala di poter gestire le prenotazioni future dei clienti attraverso app. Login dei vari dipendenti.

2 Requisiti

2.1 Use Case

Dalla descrizione del nostro modello di dominio, abbiamo individuato nelle varie figure professionali del ristorante gli attori in gioco e nelle loro mansioni i corrispettivi casi d'uso. Riportiamo quindi di seguito l'Use Case Diagram risultante:

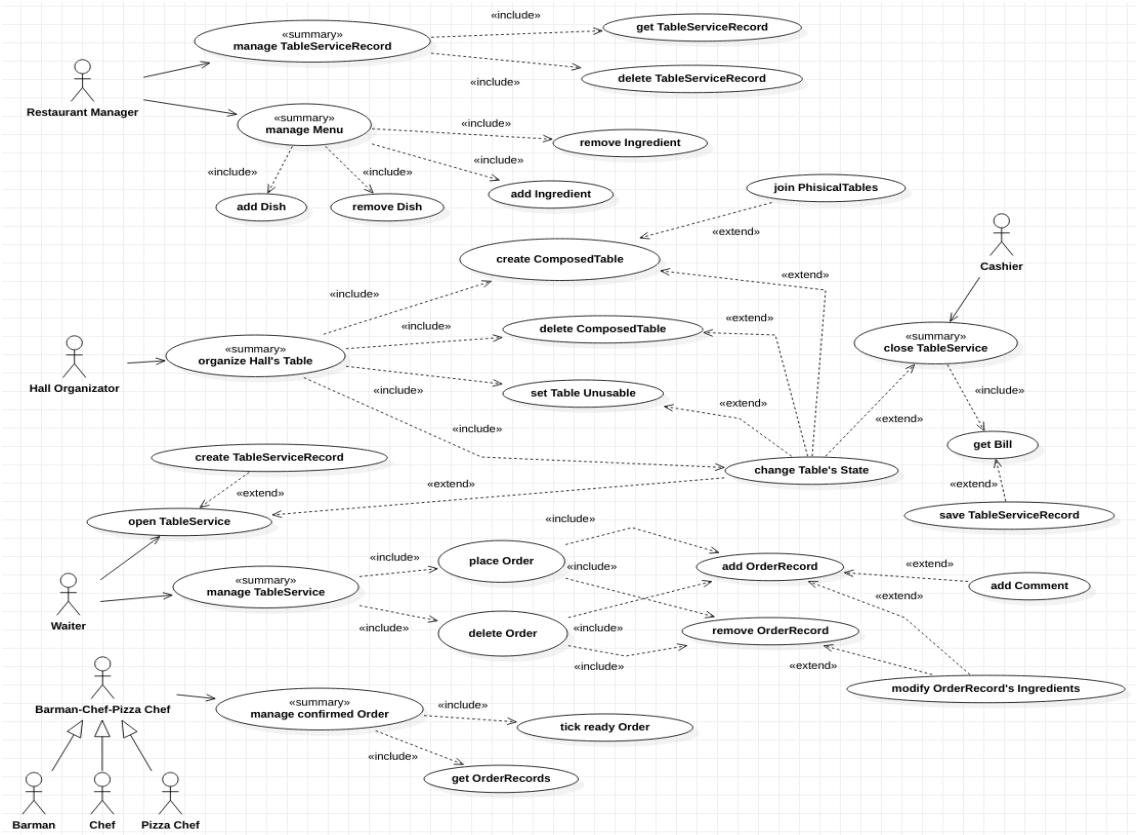

Figure 1: Use Case Diagram

2.2 Use Case Template

Riportiamo di seguito dei template relativi ad alcuni dei pricipali casi d'uso individuati nel nostro progetto:

2.3 Mockups

Riportiamo di seguito dei possibili mockups relativi alle interfaccie grafiche della nostra app per l'interazione con i nostri attori.

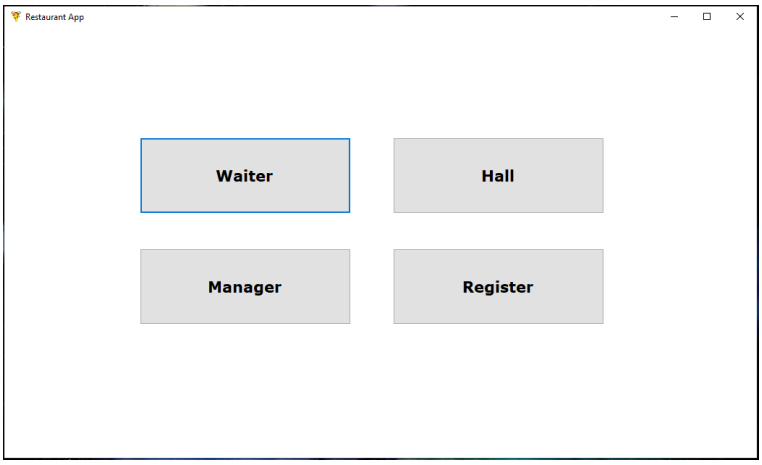

Figure 2: Mockup raffigurante un prototipo della home page dell'app

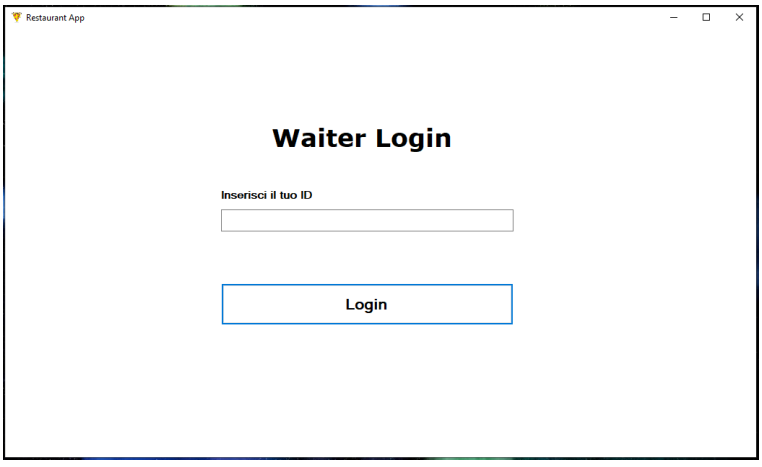

Figure 3: Mockup raffigurante un prototipo della pagina di login del cameriere

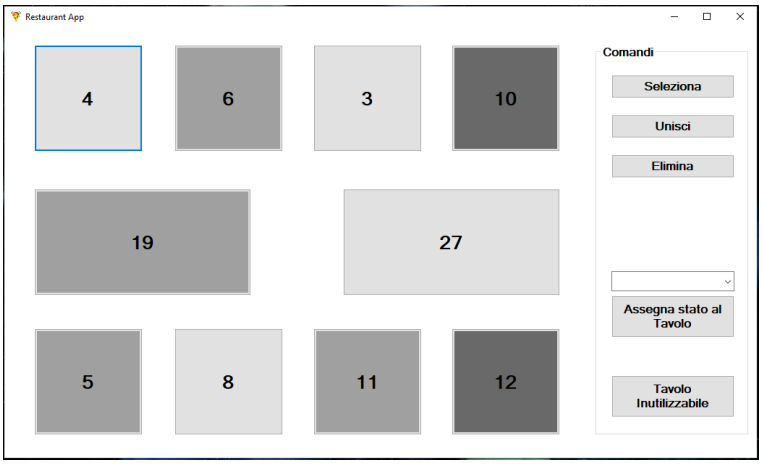

Figure 4: Mockup raffigurante un prototipo dell'interfaccia dell'organizzatore della sala

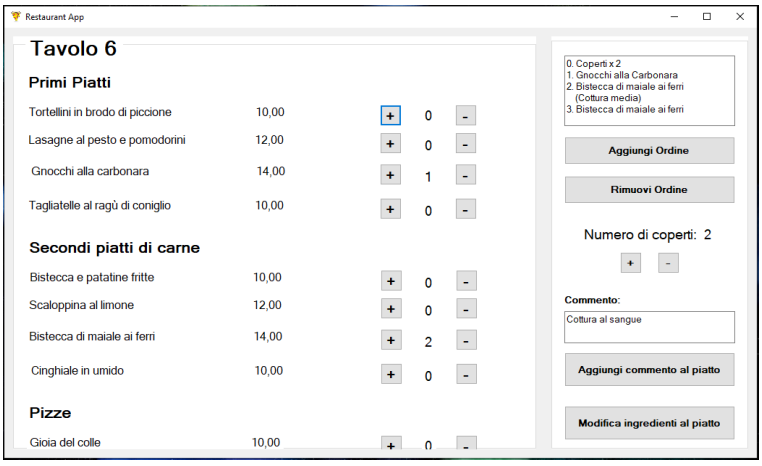

Figure 5: Mockup raffigurante un prototipo dell'interfaccia del cameriere al momento della creazione di un ordine

3 Progettazione e Implementazione

3.1 Scelte implementative e considerazioni

L'applicativo si presta per essere relalizzato ed opearare come un sistema distribuito, prevedendo cioé un server centrale con al suo interno i dati comuni accessibili dai vari terminali attraverso un protocollo di comunicazione. Per le competenze richieste dall'elaborato, lo schema da noi riportato e implementato ha il solo scopo di illustrare le interazioni principali fra le varie parti, non é

quindi atto ad una realizzazione concreta.

3.2 Class Diagram

Qui di seguito riportiamo la realizzazione del Class Diagram che descrive la nostra logica di dominio in prospettiva di implementazione:

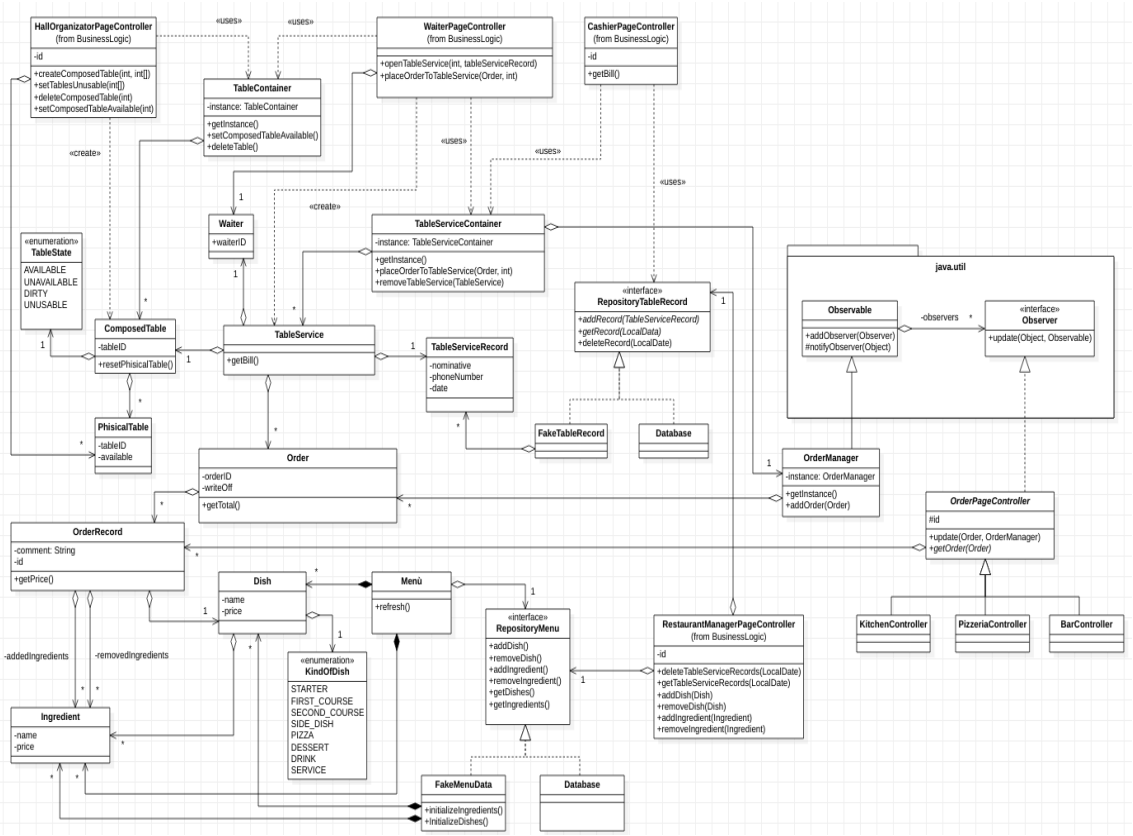

Figure 6: Class Diagram

3.3 Classi ed Interfacce

Per l'implementazione del nostro applicativo abbiamo sia definito nuove classi ed interfacce specifiche per il nostro progetto, sia utilizzato alcune di quelle giá presenti nelle librerie standard di Java. Le principali classi contenute nel package ...

3.4 Design Patterns

Nella realizzazione del nostro progetto ci siamo imbattuti in delle situazioni dove é emersa la necessitá di introdurre dei design patterns noti al fine di gestirle in modo agile ed elegante. I patterns utilizzati nel nostro applicativo sono:

1. Observer
2. Singleton
3. MVC

3.4.1 Observer

Il pattern Comportamentale Observer é utilizzato per notificare ad uno o piú oggetti che svolgono il ruolo di osservatori quando un altro elemento osservato varia il suo stato di interesse.
Nella nostra logica di dominio abbiamo avuto la necessitá di introdurre questo pattern per notificare alla cucina l'invio da parte del cameriere di una nuova comanda presa a un tavolo. Nonostante siano state deprecate, per implementare il pattern Observer ci siamo serviti delle classi "Observable" e "Observer" e dei loro rispettivi metodi "notify" e "update" forniti dalla libreria java.util. Abbiamo inoltre deciso di implementare il pattern Observer in modalitá push, la quale prevede che l'oggetto osservato notifichi gli osservatori inviando direttamente il cambiamento al momento della sua variazione.
Come é possibile infatti apprezzare dal Class Diagram di Figura 6, la classe Order Manager svolge il ruolo di oggetto osservato (Observable) che notifica l'arrivo di una nuova comanda inviandola direttamente alla cucina, alla pizzeria e al bar che svolgono il ruolo di osservatori (Observer). Questo avviene attraverso il metodo "addOrder" che preso come parametro il nuovo ordine piazzato dal cameriere lo invia agli Observers attraverso il metodo "notify", questi ultimi aggiorneranno poi il loro stato attraverso il metodo "update" (Vedi Sequence Diagram in Figura 9).

3.4.2 Singleton

Il pattern Creazionale Singleton é utilizzato per avere un'unica istanza di una determinata classe.
Nel nostro progetto abbiamo infatti introdotto questo pattern per garantire l'unicitá delle istanze delle classi "TableContainer", "TableServiceContainer" e "OrderManager", le quali interagiscono con piú attori e classi e si ha quindi la necessitá che siano sempre le stesse. Come é possibile osservare nel Sequence Diagram di Figura 9, gli attori non possiedono i riferimenti a l'unica instanza di queste classi, ma li ottengono al momento che necessitano di utilizzarle attraverso il rispettivo metodo "getInstance".

3.4.3 MVC

Il pattern Architetturale Model-View-Controller é utilizzato quando si ha la necessitá di accedere e modificare dei dati attraverso interazioni differenti con i client. Per far questo si possono dunque individuare le tre componenti principali:

- Model: É la parte che definisce il modello dei dati e le operazioni che possono essere eseguite su queste presentandole alla View e al Controller. Il Model puó inoltre notificare ai vari componenti della View eventuali aggiornamenti in seguito a richieste del Controller al fine di presentare ai Client dati sempre aggiornati.

- View: Corrisponde alle varie interfacce dell'applicazione con cui gli utenti si possono trovare ad interagire per eseguire delle azioni sul sistema.

- Controller: Svolge il ruolo da mediatore fra il View e il Model, trasformando le interazioni dell'utente sul primo in azioni sul secondo. Per farlo sfrutta inoltre un meccanismo che prende il nome di business logic. Questo consiste nel mettere in vita determinati oggetti del Model solo nel momento del loro effettivo bisogno ossia quando devono interagire in risposta alle richieste del client.

Riportiamo di seguito l'architettura del nostro applicativo suddivisa nelle tre componenti sopra descritte:

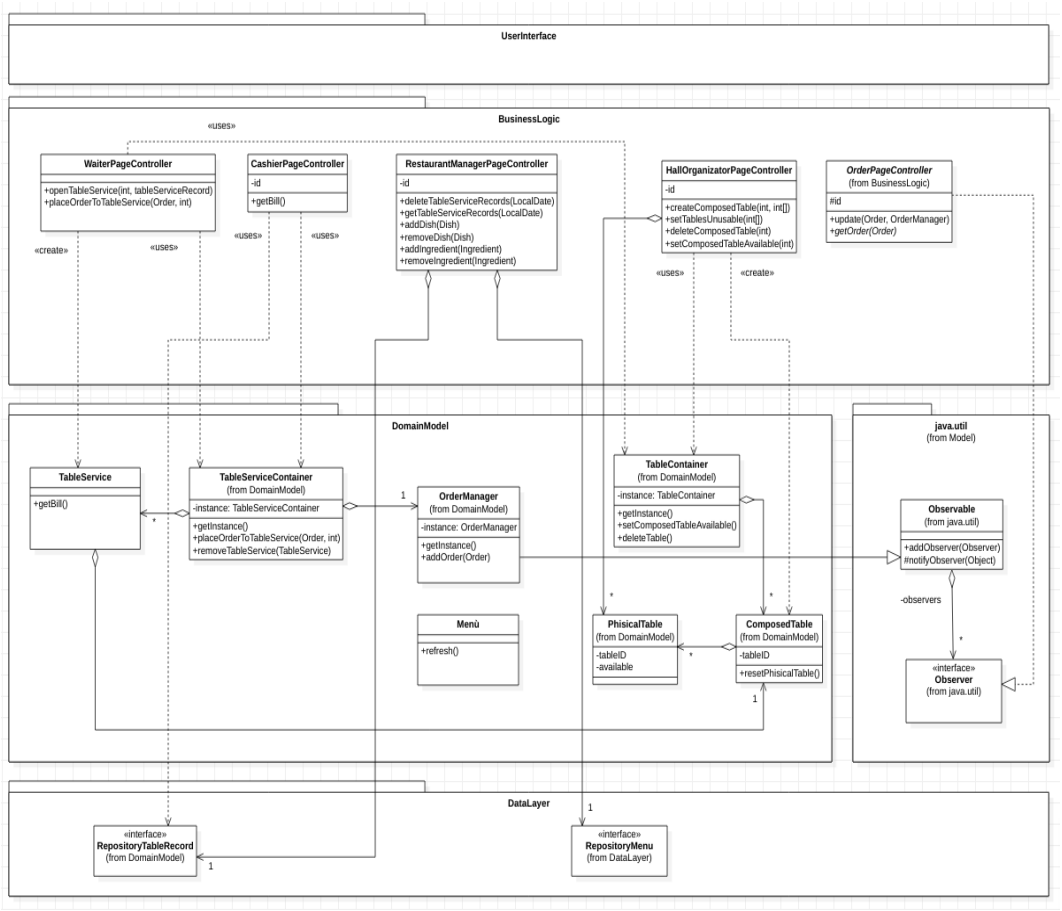

Figure 7: Raffigurazione dell'architettura dell'applicativo secondo lo schema Model-View-Controller

3.5 Disposizione delle classi nei package

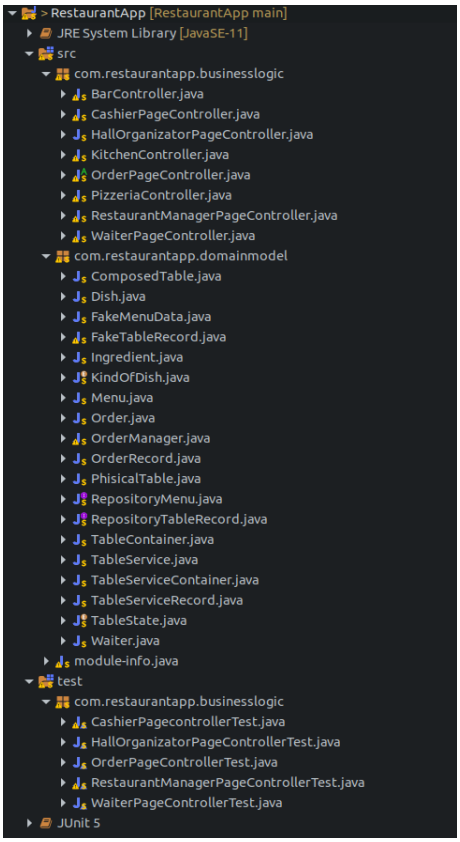

Figure 8: Raffigurazione della disposizione delle classi del progetto nei package

4 UnitTest e flusso di controllo

4.1 Unit Test

Per testare la corretta interazione e collabborazione fra le parti abbiamo deciso di eseguire i seguenti test sui principali casi d'uso dei nostri attori...

4.2 Sequence Diagram

Di seguito riportiamo un possibile scenario di interazione dei nostri attori:

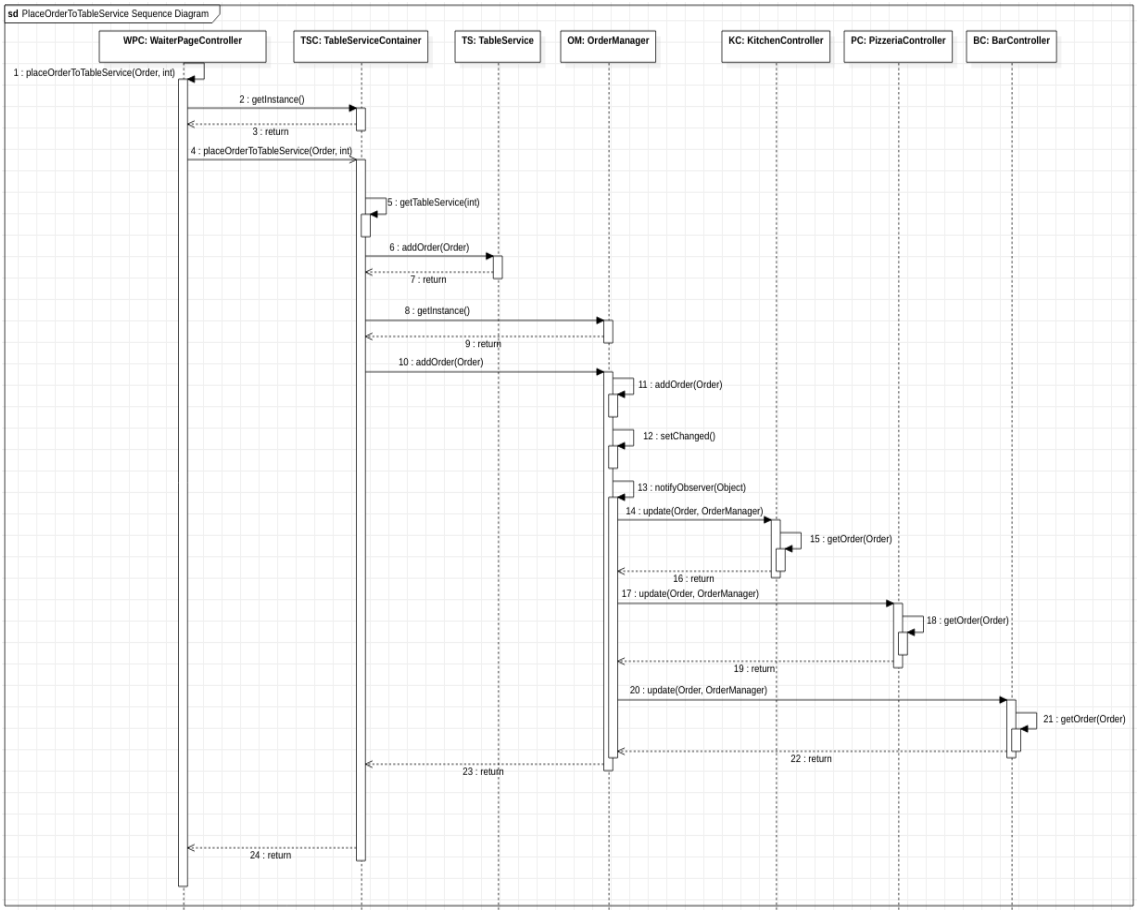

Figure 9: Sequence Diagram che documenta il flusso del controllo nel momento in cui il cameriere piazza un nuovo ordine a un tavolo