UNIVERSITÀ DEGLI STUDI DI URBINO CARLO BO

Corso di Laurea in Informatica Applicata

Programmazione e Modellazione a Oggetti, a.a. 2024/2025

Simulazione di Ristorante

Relazione per il Progetto del Corso di Programmazione e Modellazione a Oggetti

Milena Balducci, matricola 321791
Marco Palazzetti, matricola 312659

Luglio 2025

Indice

1 Analisi

L'obiettivo del progetto è la realizzazione di un'applicazione che simuli il funzionamento di un ristorante. Il sistema è composto da:

- Tre reparti per la preparazione dei prodotti: cucina, pizzeria e bar
- Una cassa, responsabile della gestione degli ordini e dei pagamenti
- Una sala, suddivisa in ranghi, ciascuno dei quali gestito da un cameriere
- Un insieme di dipendenti, divisi tra preparatori, camerieri e maitre

Il flusso del sistema prevede che i gruppi di clienti, composti da 2 a 10 persone, vengano assegnati a un tavolo con un numero sufficiente di posti. I camerieri raccolgono le ordinazioni e le inviano alla cassa, che le smista automaticamente ai reparti competenti.
I reparti eseguono la preparazione dei prodotti tenendo conto del carico di lavoro del personale. Le ordinazioni vengono servite al tavolo solo quando tutti i prodotti richiesti sono pronti. Al termine del pasto, viene richiesto il conto e il tavolo viene liberato per altri clienti.
Infine, al termine del turno, la cassa calcola il guadagno del ristorante e i parziali per ogni cameriere e reparto.

1.1 Requisiti

1.1.1 Requisiti minimi

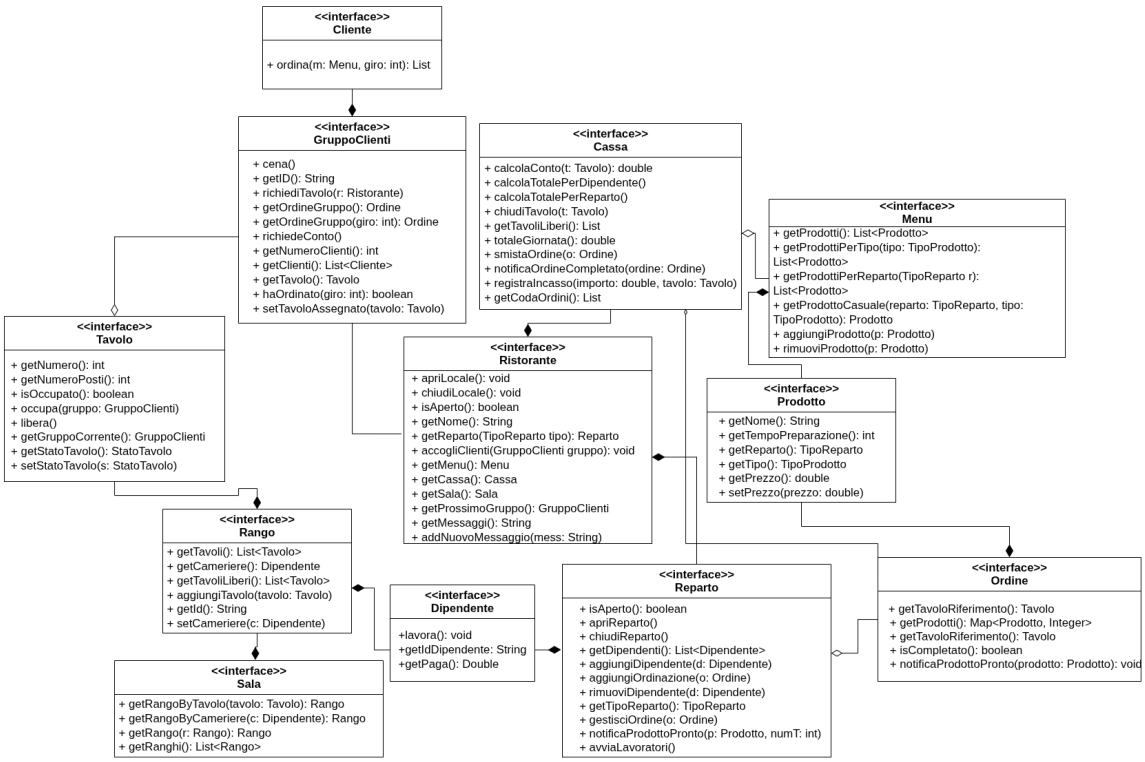
- Gestione dell'arrivo dei clienti in gruppi (2–10 persone)
- Assegnazione automatica del primo tavolo disponibile con posti sufficienti
- Notifica alla cassa dell'occupazione dei tavoli da parte dei camerieri
- Scelta dei prodotti dal menu: ogni cliente ordina una bevanda e un piatto principale
- Ogni prodotto è associato a: prezzo, tempo di preparazione, reparto di competenza
- Smistamento automatico dell'ordine dalla cassa ai reparti
- Assegnazione dei prodotti ai dipendenti con minor carico nel reparto
- Servizio simultaneo solo quando tutti i prodotti del tavolo sono pronti
- Possibilità di ordinare dessert e caffetteria dopo il pasto principale
- Emissione del conto da parte della cassa e liberazione del tavolo
- Calcolo del guadagno totale giornaliero e dei totali parziali per camerieri e reparti

1.1.2 Requisiti opzionali

- Sistema di prenotazione dei tavoli
- Sistema di mance con classifica per rendimento dei camerieri
- Gestione degli ingredienti e dei fornitori, con relativo impatto sui costi di gestione

1.2 Modello del dominio

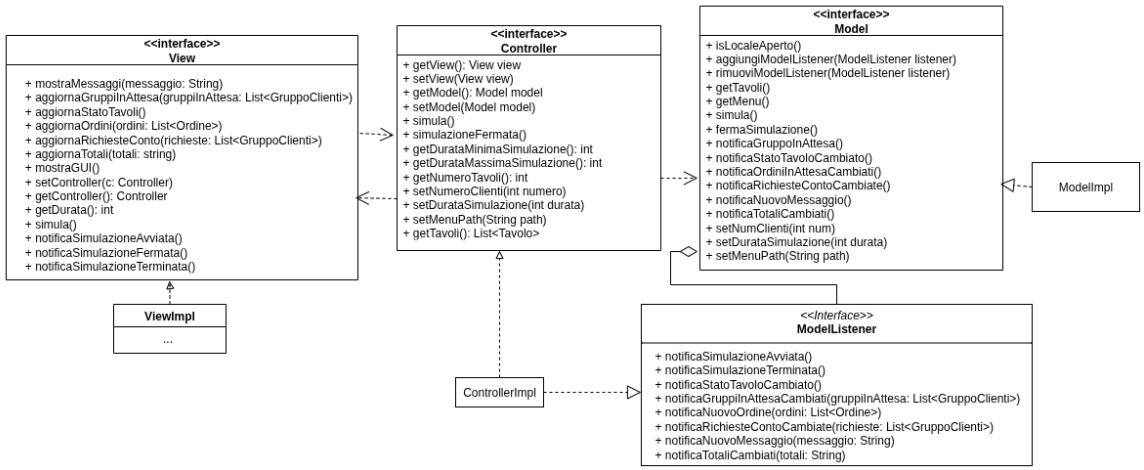
L'entità centrale della simulazione è rappresentata dall'interfaccia Ristorante, che coordina i principali componenti del sistema: Sala, Cassa e Reparti.
Ogni Reparto, specializzato in un determinato tipo di preparazione, è composto da uno o più Dipendenti incaricati di realizzare gli ordini assegnati dalla Cassa. Quest'ultima ha il compito di smistare gli ordini ai reparti competenti e di calcolare i guadagni giornalieri del ristorante.
La Sala è suddivisa in Ranghi, ognuno dei quali contiene un insieme di Tavoli e un Cameriere dedicato. Il cameriere si occupa sia della raccolta degli ordini da ciascun tavolo del proprio rango, sia della gestione delle richieste di conto, che inoltra alla Cassa.
L'interazione tra Camerieri e Reparti avviene sempre attraverso la Cassa, che funge da nodo centrale della comunicazione.
I Clienti sono organizzati in GruppiClienti, ciascuno composto da un numero variabile di persone. Ogni gruppo richiede l'accesso al Ristorante, che delega l'assegnazione di un tavolo a un dipendente. Una volta accomodato, il gruppo interagisce con il cameriere responsabile del proprio tavolo e con il tavolo stesso durante l'intera esperienza.



2 Design

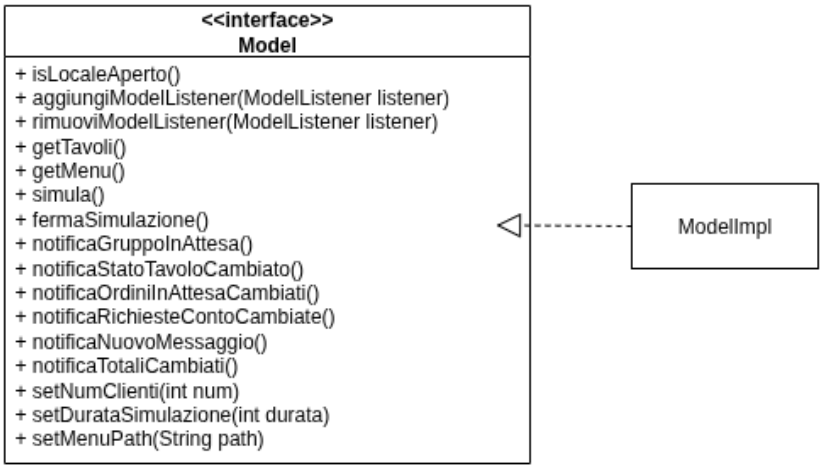
2.1 Architettura

Per lo sviluppo dell'architettura è stato adottato il pattern architetturale Model-View-Controller.
Il Model ha il compito di gestire lo stato della simulazione, offrendo metodi per accedere e modificare i dati.
La View si occupa invece della rappresentazione grafica delle informazioni contenute nel Model, presentandole all'utente in modo chiaro e intuitivo.
L'interazione dell'utente avviene tramite la View, che riceve gli input; la loro elaborazione e gestione è affidata al Controller, il quale ha anche il compito di aggiornare il Model in base a tali input.
Poiché il Model deve restare indipendente da View e Controller, si ricorre al pattern Observer (tramite l'interfaccia ModelListener) per notificare al Controller eventuali cambiamenti di stato. In questo contesto, il Model agisce da soggetto osservato, mentre il Controller ricopre il ruolo di osservatore, potendo così informare le View in seguito a ogni aggiornamento.



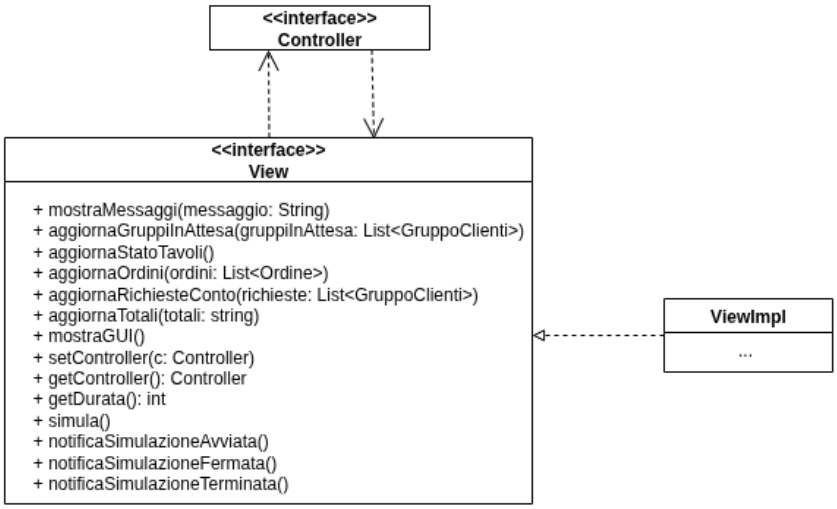
2.1.1 Model

Il Model è responsabile della gestione delle modalità di accesso e modifica dei dati relativi all'applicazione e al suo dominio. Il suo compito principale è determinare in modo coerente come le interazioni dell'utente influenzino lo stato dell'applicazione.
A tal fine, il Model fornisce al Controller un'interfaccia per accedere allo stato attuale del sistema.
Per implementare questa funzionalità è stata definita un'entità Model, incaricata di mantenere internamente lo stato dell'applicazione, inclusi i componenti principali come il ristorante e i gruppi di clienti attualmente presenti.
L'implementazione concreta di questa entità è realizzata nella classe ModelImpl.



2.1.2 View

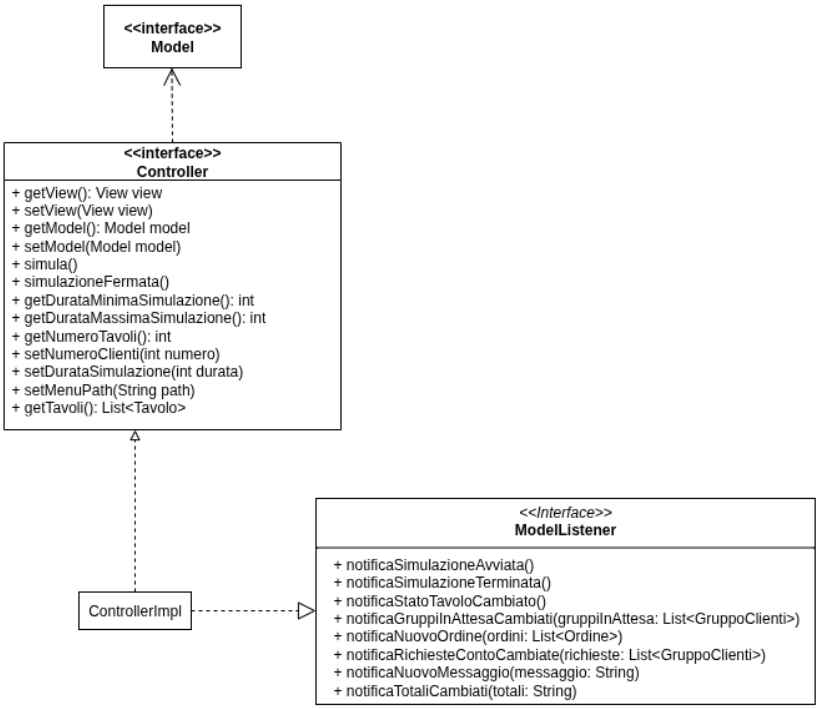
La View rappresenta l'interfaccia grafica dell'applicazione e si occupa della gestione dell'esperienza utente, visualizzando i dati e permettendo l'interazione con il sistema. È responsabile di rilevare le azioni dell'utente e notificare il Controller affinché le gestisca correttamente.
Si è scelto di progettare View e Controller in modo indipendente dal framework grafico utilizzato così da rendere semplice l'adozione futura di altre librerie grafiche. In tal modo, un eventuale cambiamento del framework richiederebbe esclusivamente la riscrittura della View, lasciando inalterati Controller e Model.



2.1.3 Controller

Il Controller ha il compito di gestire le interazioni dell'utente provenienti dalla View, traducendole in azioni da applicare al Model. Allo stesso tempo, si occupa di aggiornare la View in base ai cambiamenti che avvengono nel Model.
Come descritto nella sezione precedente, la classe concreta che realizza il Controller, ControllerImpl, implementa anche l'interfaccia ModelListener, assumendo il ruolo di osser-

vatore del Model. In questo modo, è in grado di reagire ai cambiamenti di stato che si verificano durante la simulazione e rifletterli tempestivamente sulla View.

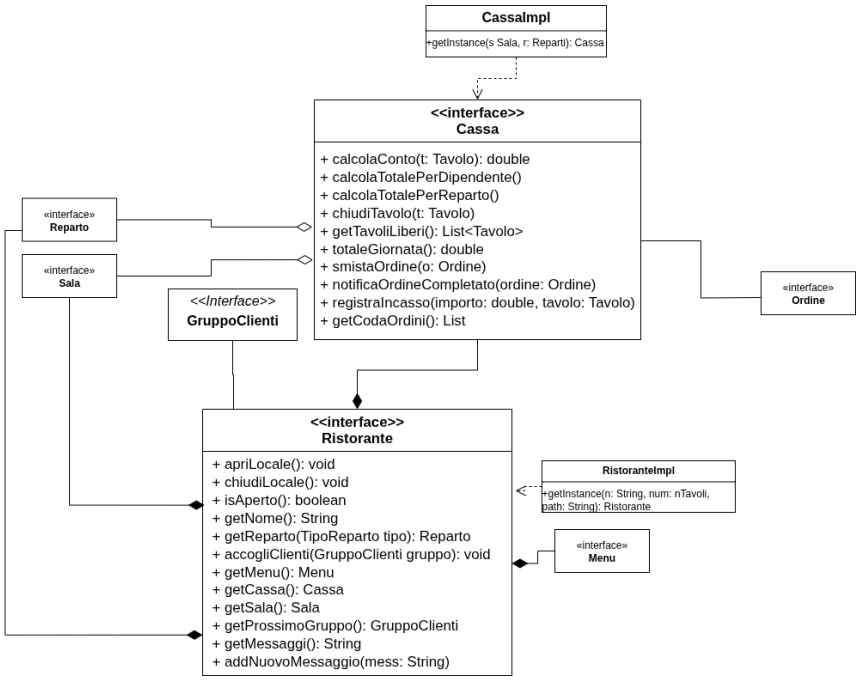


2.2 Design dettagliato

2.3 Balducci

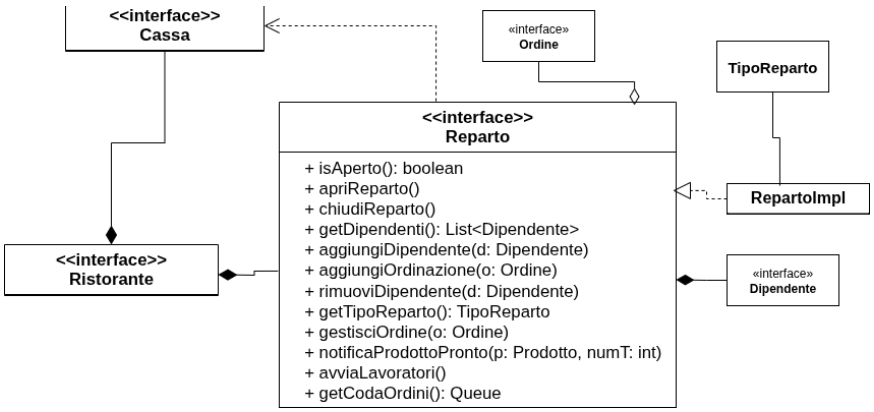
2.3.1 Ristorante, Cassa

L'interfaccia `Ristorante` è stata implementata seguendo il pattern **Singleton**, in quanto ha senso che esista una sola istanza di ristorante in esecuzione, dato che tutte le operazioni e le interazioni (tra clienti, tavoli, personale, ecc.) fanno riferimento a un unico ambiente condiviso. Lo stesso principio è stato applicato anche all'interfaccia `Cassa`, che svolge un ruolo altrettanto centrale nella gestione degli ordini e del flusso economico del locale. Implementare entrambe le entità come Singleton previene la creazione involontaria di istanze multiple che potrebbero portare a comportamenti incoerenti o difficili da gestire.



2.3.2 Reparto

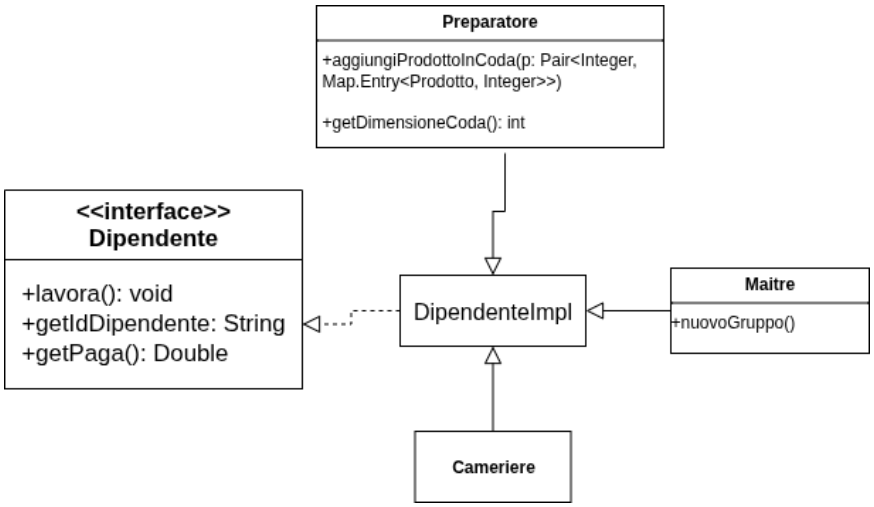
La presenza di diversi tipi di reparto potrebbe far pensare a un'implementazione basata su una classe astratta, da estendere poi con una sottoclasse per ogni specifico reparto. Tuttavia, dal momento che il comportamento e la logica interna sono sostanzialmente identici per tutte le tipologie di reparto, si è scelto un approccio più semplice e diretto. In particolare, si è deciso di rappresentare le differenze tra i reparti tramite un'entità `TipoReparto`, che si occupa di specificare per ciascun reparto il nome e il numero di dipendenti assegnati. In questo modo si evita una proliferazione di classi inutilmente complesse, mantenendo il codice più leggibile e facilmente manutenibile.



2.3.3 Dipendente

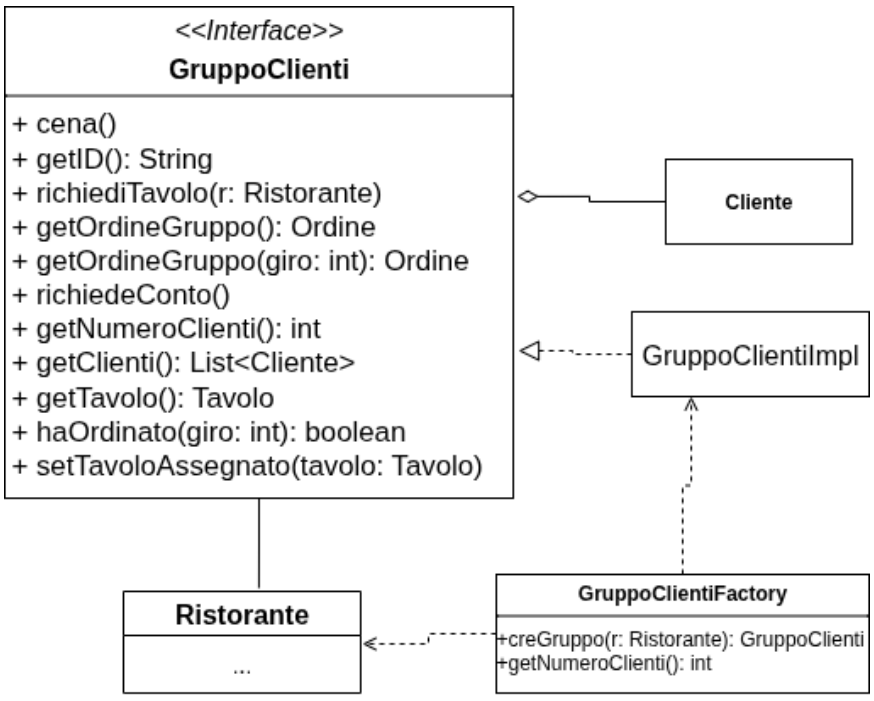
Si è deciso di rendere `DipendenteImpl` una classe astratta, visto che ci sono tre diversi tipi di dipendente: `Preparatore, Cameriere` e `Maitre`.
`Maitre` ha il metodo aggiuntivo `nuovoGruppo`, che aggiorna il contatore interno all'oggetto rappresentante il numero di gruppi in attesa ancora da gestire.
`Preparatore` ha i metodi aggiuntivi `getDimensioneCoda` e `aggiungiProdottoInCoda`,

utilizzati per gestire la coda interna all'oggetto contenente i prodotti da preparare.
La paga di ciascun dipendente è definita nell'entità `StipendiDipendenti`.



2.3.4 Cliente, GruppoClienti

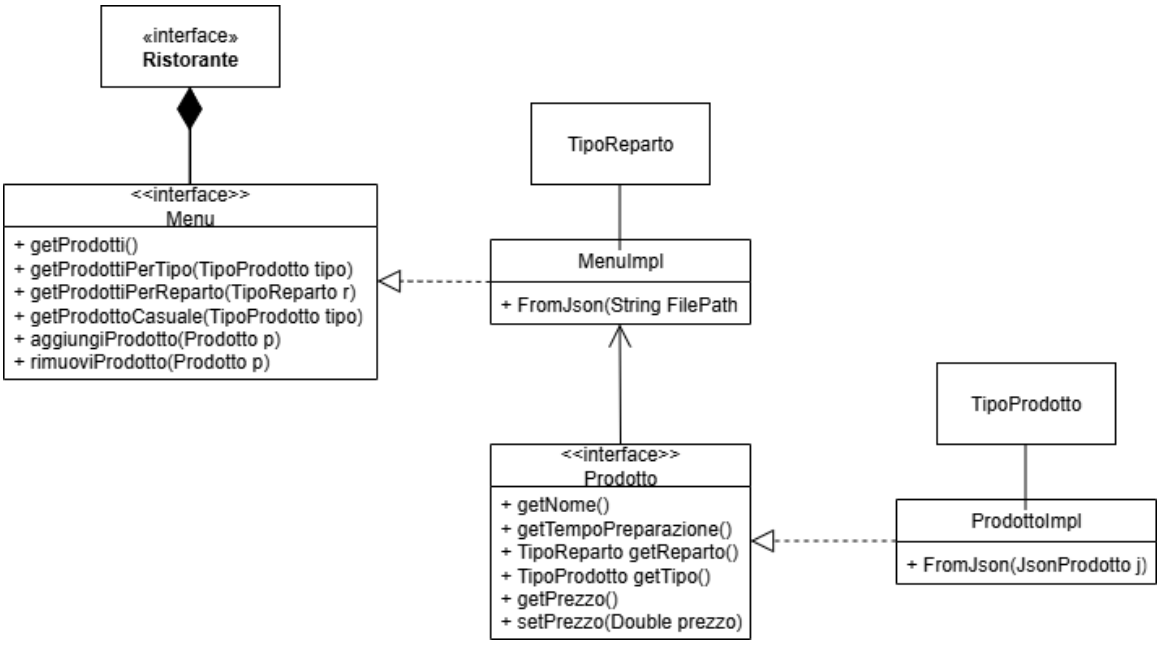
Per gestire l'arrivo dei clienti al ristorante, è stato adottato il pattern **Factory**, incaricato della creazione dei gruppi di clienti uno alla volta, con dimensioni determinate in modo casuale. Questo approccio consente di mantenere il codice più leggibile e modulare, rendendo semplice intervenire sulla logica di generazione dei gruppi. L'intera responsabilità di tale logica è infatti incapsulata in un componente dedicato, favorendo la manutenibilità e l'estensibilità del sistema.
Ciascun `Cliente` è creato dal proprio `GruppoClienti` in base alla dimensione di quest'ultimo e gestito da esso.



2.4 Palazzetti

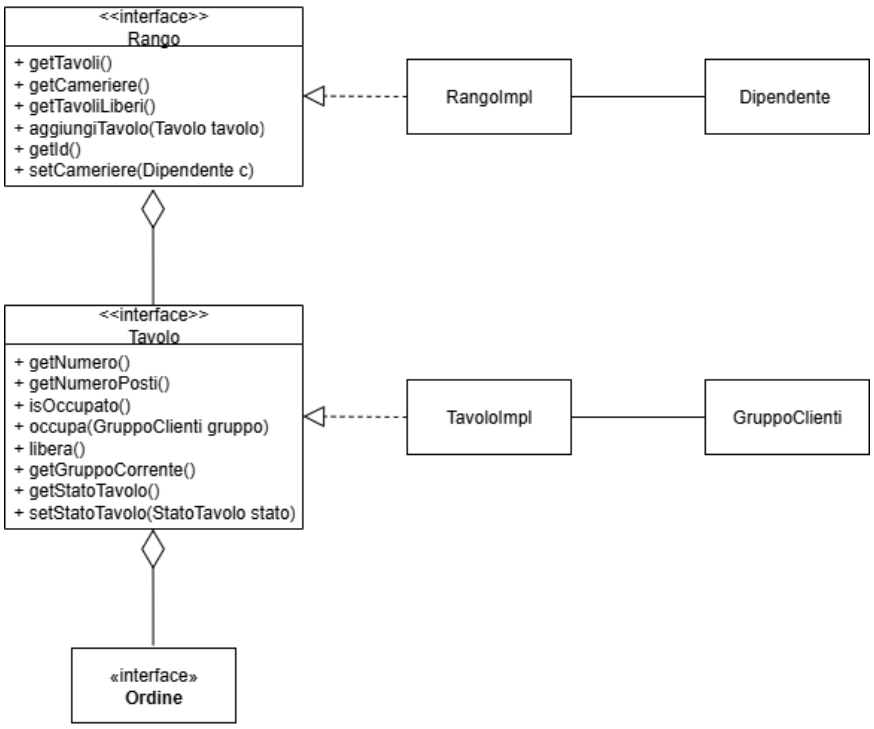
2.4.1 Menu, Prodotto

La classe `Menu` rappresenta l'insieme dei prodotti disponibili nel ristorante.
È stata implementata utilizzando il pattern Factory, in quanto si occupa della creazione e gestione dei vari `Prodotto`, favorendo modularità e facilità di estensione del sistema.
La classe `Prodotto` modella ogni singolo elemento offerto ai clienti. Anche questa entità è parte integrante del pattern Factory, essendo istanziata tramite il `Menu`. Pur mantenendo una struttura semplice, `Prodotto` è facilmente estendibile e fornisce attributi essenziali come il tipo e il reparto di competenza, utili per il filtraggio e lo smistamento degli ordini.



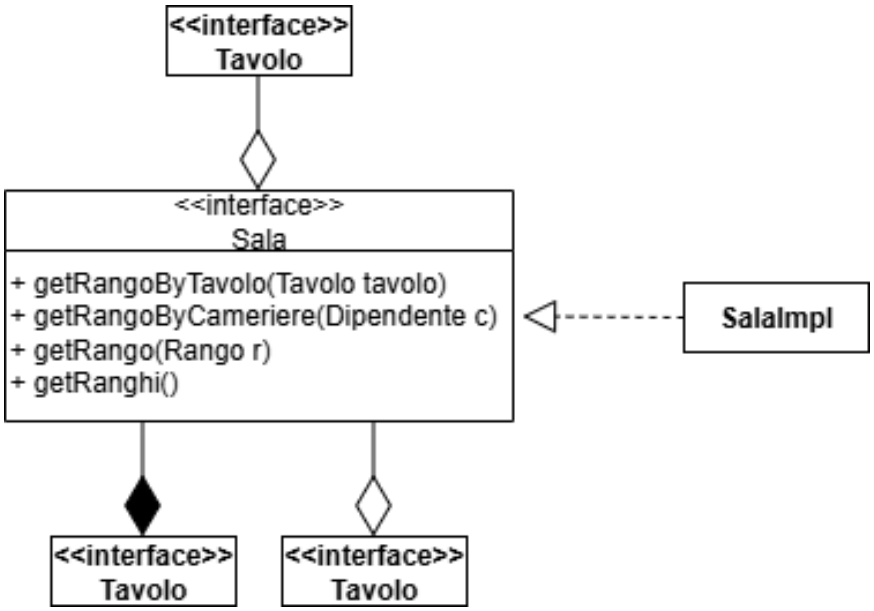
2.4.2 Tavolo, Rango

la classe `Tavolo` rappresenta i tavoli fisici della sala. Gestisce lo stato (occupato/libero) e mantiene un riferimento al `GruppoClienti` assegnato.
Questa entità utilizza un'implementazione diretta (Strategy normale), focalizzandosi sul tracciamento e sulla disponibilità dei posti.
La classe `Rango` definisce una porzione della sala affidata a un singolo cameriere.
Ogni rango contiene una lista di tavoli ed è associato a un dipendente (Cameriere). Anche in questo caso, l'approccio utilizzato è una Strategy diretta, che permette al sistema di gestire la suddivisione logica della sala in maniera semplice e coerente.



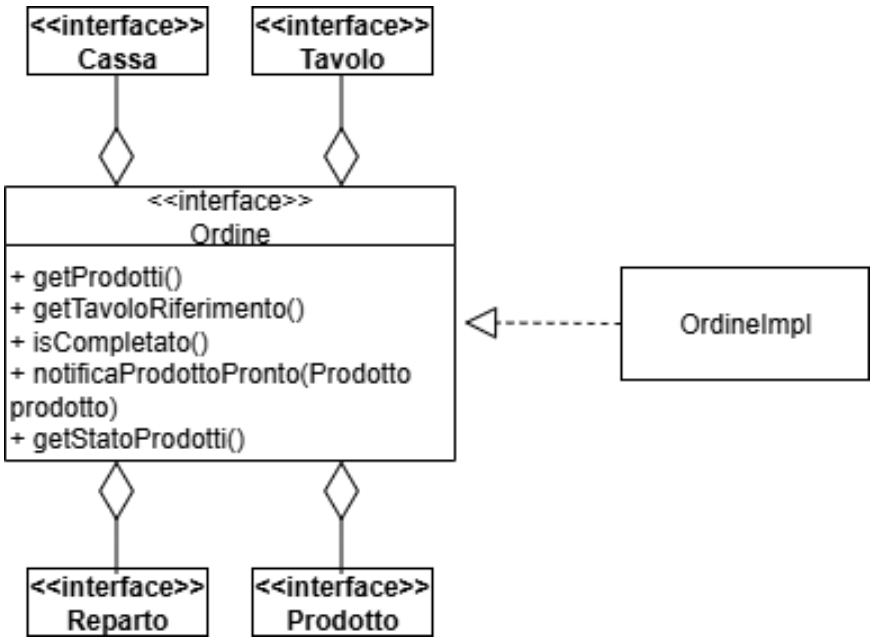
2.4.3 Sala

La classe `Sala` è il componente centrale per il coordinamento tra Tavolo, Rango e il personale di sala.
È stata implementata seguendo il pattern Singleton, in quanto ha senso che esista una sola istanza di sala che rappresenti l'intero spazio fisico del ristorante.
La Sala gestisce la lista dei ranghi e funge da punto di accesso per operazioni di servizio, come l'assegnazione di posti e la gestione della disponibilità da parte di Maitre e Cameriere.



2.4.4 Ordine

La Classe `Ordine` rappresenta l'insieme degli elementi richiesti da un gruppo di clienti.
È stata implementata utilizzando una struttura semplice basata su una Strategy normale, in quanto si occupa principalmente di raccogliere i prodotti scelti dal cliente e indirizzarli verso i reparti appropriati.
Il suo ruolo è fondamentale per l'interazione tra clienti, Menu e i reparti di preparazione, pur mantenendo una logica interna lineare e diretta



3 Sviluppo

3.1 Testing automatizzato

Questa sezione descrive in dettaglio i casi di test implementati per verificare il corretto funzionamento delle principali componenti del sistema di simulazione del ristorante. Per ogni test sono specificati:

- **Obiettivo del test**: descrive lo scopo e il comportamento atteso.
- **Scenario di test**: illustra i passaggi chiave eseguiti durante il test.
- **Implementazione tecnica**: dettaglia le classi e i metodi coinvolti.
- **Risultati attesi**: elenca le asserzioni e i criteri di successo.

3.1.1 AssegnaTavoliTest

Obiettivo del test: Verificare che un gruppo di clienti venga assegnato a un tavolo libero con capacità adeguata e che la disponibilità della sala venga aggiornata.

Scenario di test:

1. Inizializzazione di un ristorante con 20 tavoli liberi tramite `RistoranteImpl.apriLocale()`.
2. Creazione di un gruppo di 5 clienti con `GruppoClientiImpl`.
3. Invocazione di `gruppo.richiediTavolo(ristorante)`.
4. Attesa sincrona usando `synchronized` e `wait()` sul gruppo.
5. Verifica che il tavolo assegnato abbia `getNumeroPosti() >= getNumeroClienti()` e che la lista di tavoli liberi si riduca a 19.

Implementazione tecnica:

- Classi: `RistoranteImpl`, `GruppoClientiImpl`, `Cassa`, `TavoloImpl`.
- Metodi principali: `apriLocale()`, `richiediTavolo()`, `getTavoliLiberi()`.
- Sincronizzazione sul thread del gruppo per modellare l'assegnazione asincrona.

Risultati attesi:

- Il gruppo ottiene un tavolo con posti sufficienti.
- La cassa indica un tavolo occupato in meno.

3.1.2 CompletaOrdineTest

Obiettivo del test: Assicurare che `OrdineImpl` cambi stato in "completato" solo dopo che tutti i prodotti ordinati sono pronti.

Scenario di test:

1. Creazione di un tavolo con `TavoloImpl`.
2. Definizione di due prodotti e inserimento in una `Map<Prodotto,Integer>`.
3. Istanziazione di `OrdineImpl(tavolo, mappa)`.
4. Verifica iniziale: `isCompletato()` restituisce `false`.
5. Chiamata a `notificaProdottoPronto()` su ciascun prodotto e verifica intermedia.
6. Al completamento di tutte le notifiche, `isCompletato()` restituisce `true`.

Implementazione tecnica:

- Classe: `OrdineImpl` con mappe per stati e quantità.
- Metodo: `notificaProdottoPronto(Prodotto)` e controllo tramite `allMatch()`.

Risultati attesi:

- Stato iniziale non completato.
- Stato completato solo dopo l'ultima notifica.

3.1.3 FiltraProdottiPerTipoTest

Obiettivo del test: Verificare che `MenuImpl.getProdottiPerTipo()` restituisca esclusivamente i prodotti del tipo richiesto.

Scenario di test:

1. Creazione di `MenuImpl`.
2. Aggiunta di un dolce e di una portata.
3. Invocazione di `getProdottiPerTipo(TipoProdotto.DESSERT)`.
4. Verifica della dimensione e del contenuto della lista.

Implementazione tecnica:

- Classi: `MenuImpl`, `ProdottoImpl`.
- Filtraggio con stream e predicate sul tipo.

Risultati attesi:

- Una sola voce nella lista.
- Il prodotto di tipo `DESSERT` presente.

3.1.4 SmistaOrdinePerRepartoTest

Obiettivo del test: Validare che `CassaImpl.smistaOrdine()` invii ogni prodotto al reparto corretto.

Scenario di test:

1. Apertura del ristorante e creazione di un ordine misto (BAR e PIZZERIA).
2. Invocazione di `smistaOrdine(ordine)`.
3. Estrazione delle code di ordini da ogni reparto e verifica della correttezza.

Implementazione tecnica:

- Classi: `RistoranteImpl`, `CassaImpl`, `Reparto`.
- Uso di `Queue` e `poll()` per ottenere i primi ordini.

Risultati attesi:

- Il BAR riceve soltanto bevande.
- La PIZZERIA riceve soltanto portate.

3.1.5 TavoliLiberiPerRangoTest

Obiettivo del test: Assicurare che `RangoImpl.getTavoliLiberi()` restituisca solo i tavoli non occupati.

Scenario di test:

1. Creazione di due tavoli e associazione a un rango.
2. Occupazione di uno dei tavoli.
3. Invocazione di `getTavoliLiberi()` e verifica dell'elenco.

Implementazione tecnica:

- Classi: `TavoloImpl`, `RangoImpl`.
- Filtro su lista basato su `isOccupato()`.

Risultati attesi:

- Un solo tavolo libero presente.
- Ordine dell'elenco coerente con l'inserimento.

3.2 Metodologia di lavoro

Le seguenti parti del programma sono state realizzate in collaborazione:

- Package main.control (interfacce Control e ModelListener, classe ControllerImpl)
- Package main.view (interfaccia View e la sua implementazione ViewImpl, classe RistoranteFrame
- Package main.model (interfaccia Model e la sua implementazione ModelImpl)

3.2.1 Balducci

Package main.balducci.interfaces e main.balducci.classes. Quest'ultimo contiene l'implementazione di tutte le interfacce illustrare nella sezione 2.3, oltre all'enum TipoReparto, e alle classi GruppoClientiFactory, Pair, e a Maitre, Preparatore e Cameriere, che estendono la classe DipendenteImpl. Stesura del file JSON contenente l'elenco di prodotti da inserire nel menu.

3.2.2 Palazzetti

Package main.palazzetti.interfaces e main.palazzetti.classes. Quest'ultimo contiene l'implementazione di tutte le interfacce illustrare nella sezione 2.4, oltre all'enum TipoProdotto e alla classe JsonProdotto, necessaria per creare istanze di ProdottoImpl (tramite il metodo factory statico fromJson() contenuto in quest'ultima) a partire da un file JSON.

3.3 Note di sviluppo

3.3.1 Balducci

- Stream in buona parte delle classi contenute in main.balducci.classes
- Thread per avviare ciascuna implementazione di Dipendente (Cameriere, Maitre, Preparatore) e ciascun GruppoClienti su un thread differente e sincronizzazione per gestire l'accesso a risorse condivide (metodi synchronized e campi con modificatore volatile).

3.3.2 Palazzetti

- Utilizzo di ObjectMapper e TypeReference per la creazione del menu a partire da un file JSON
- Stream nella classe OrdineImpl

3.4 Sorgenti

Link GitHub: https://github.com/ameelleea/progetto_pmo_202425.git