Università degli Studi di Napoli

"Parthenope"

Progetto di Programmazione III

e

Laboratorio di Programmazione III

Studente: Crescenzo Cerqua 0124002171
Fabio Salese 0124002165

Professori: *Angelo Ciaramella, Raffaele Montella*

Anno Accademico: 2022/2023

MyRestaurant

Sommario

Descrizione del progetto

Il progetto in oggetto mira a sviluppare un sistema automatizzato per gestire gli ordini in un ristorante, sfruttando il linguaggio di programmazione Java insieme alla tecnologia JavaFX e Scene Builder per la creazione di un'interfaccia grafica intuitiva. Il sistema si concentra sulla gestione degli ordini in un contesto ristorante, dove n camerieri e m tavoli sono distinti da codici univoci. Ogni cameriere è responsabile di raccogliere le ordinazioni dai tavoli e inviarle automaticamente ai vari reparti. Abbiamo al momento creato i reparti cucina o bevande, suddividendo i prodotti in categorie.

Il sistema prevede due modalità di accesso: modalità amministratore e modalità cameriere. L'amministratore ha la capacità di:

1. Inserire o modificare nuovi piatti o bevande nel menu del ristorante.
2. Gestire i pagamenti dei clienti in base al codice del tavolo, con opzioni di pagamento tramite carta di credito, bancomat o in contanti.
3. Visualizzare periodicamente i prodotti più venduti per scopi di analisi e gestione.

D'altra parte, i camerieri possono:

1. Raccogliere le ordinazioni dei clienti in modo efficiente.

2. Annullare l'ultima ordinazione effettuata in caso di errori.
3. Fornire suggerimenti ai clienti, presentando piatti o bevande alternativi della stessa categoria.

Il sistema simula anche i tempi necessari per la preparazione di un ordine, nel nostro caso per motivi didattici impostato a pochi secondi.

Nel corso dello sviluppo del progetto, sono stati adottati almeno quattro design pattern: Command, Strategy, Prototype e Chain of Responsibility, rispettando i principi SOLID di programmazione. Inoltre, sono stati inclusi commenti dettagliati per garantire la comprensione del codice. La gestione delle eccezioni è stata implementata per garantire un comportamento robusto e il sistema ha sfruttato l'uso di database per la conservazione dei dati, nel nostro caso il database MySql.

Diagramma delle classi

Strategy

Il design pattern Strategy è un pattern comportamentale ampiamente impiegato nell'ambito dell'ingegneria del software, concepito per gestire la definizione di una famiglia di algoritmi e consentire di selezionarne uno in modo flessibile e dinamico durante l'esecuzione di un programma. Questo pattern promuove la separazione delle variazioni dell'algoritmo dalla classe principale che lo utilizza, contribuendo a migliorare la manutenibilità, la flessibilità e la riutilizzabilità del codice.

Il design pattern Strategy è utilizzato per gestire i diversi metodi di pagamento. L'interfaccia PaymentStrategy definisce un contratto per i vari metodi di pagamento, specificando il metodo makePayment(double amount) che deve essere implementato dalle sue sottoclassi.

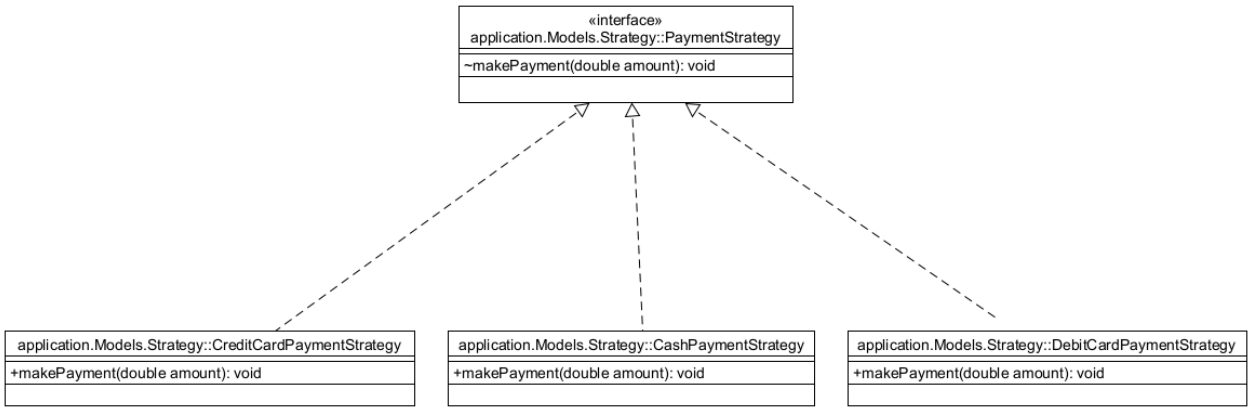
Le classi CashPaymentStrategy, CreditCardPaymentStrategy, e DebitCardPaymentStrategy implementano l'interfaccia PaymentStrategy e definiscono i metodi specifici per effettuare il pagamento con ciascun metodo. Tuttavia, al momento, queste implementazioni sono lasciate incomplete (// TODO), quindi attualmente, la logica effettiva del pagamento è da implementare all'interno di questi metodi.

La classe PaymentMethodSelector funge da factory per selezionare il metodo di pagamento desiderato, restituendo la strategia corretta in base al nome del metodo di pagamento specificato.

Il controller menuformadminController implementa il pattern Strategy nel metodo menuPayBtn(). Quando l'utente preme il pulsante "Pay", il controller recupera il metodo di pagamento selezionato e utilizza il PaymentMethodSelector per ottenere la corretta strategia di pagamento. Successivamente, invoca il metodo makePayment(double amount) della strategia scelta, passando l'importo totale come argomento.

Ad esempio, se l'utente seleziona il pagamento in contanti, il controller acquisisce la strategia di pagamento per contanti dal PaymentMethodSelector e chiama makePayment(double amount) per effettuare il pagamento utilizzando la strategia dei contanti.

In definitiva, il pattern Strategy consente di selezionare e utilizzare in modo dinamico diversi metodi di pagamento, separando l'implementazione specifica dei vari metodi di pagamento dalla logica del controller.



Command

Il design pattern Command è un pattern comportamentale ampiamente utilizzato nell'ingegneria del software, concepito per incapsulare una richiesta come oggetto, consentendo così di parametrizzare i client con richieste, accodarle, registrare le richieste e supportare operazioni di annullamento. Questo pattern promuove l'incapsulamento, il disaccoppiamento e la flessibilità nell'esecuzione delle operazioni.

Abbiamo utilizzato il design pattern Command in due controller distinti: cardProductController e menuformController. Entrambi i controller hanno

implementazioni di Command, specificamente AddToCartCommand e RemoveFromCartCommand, che incapsulano azioni specifiche.

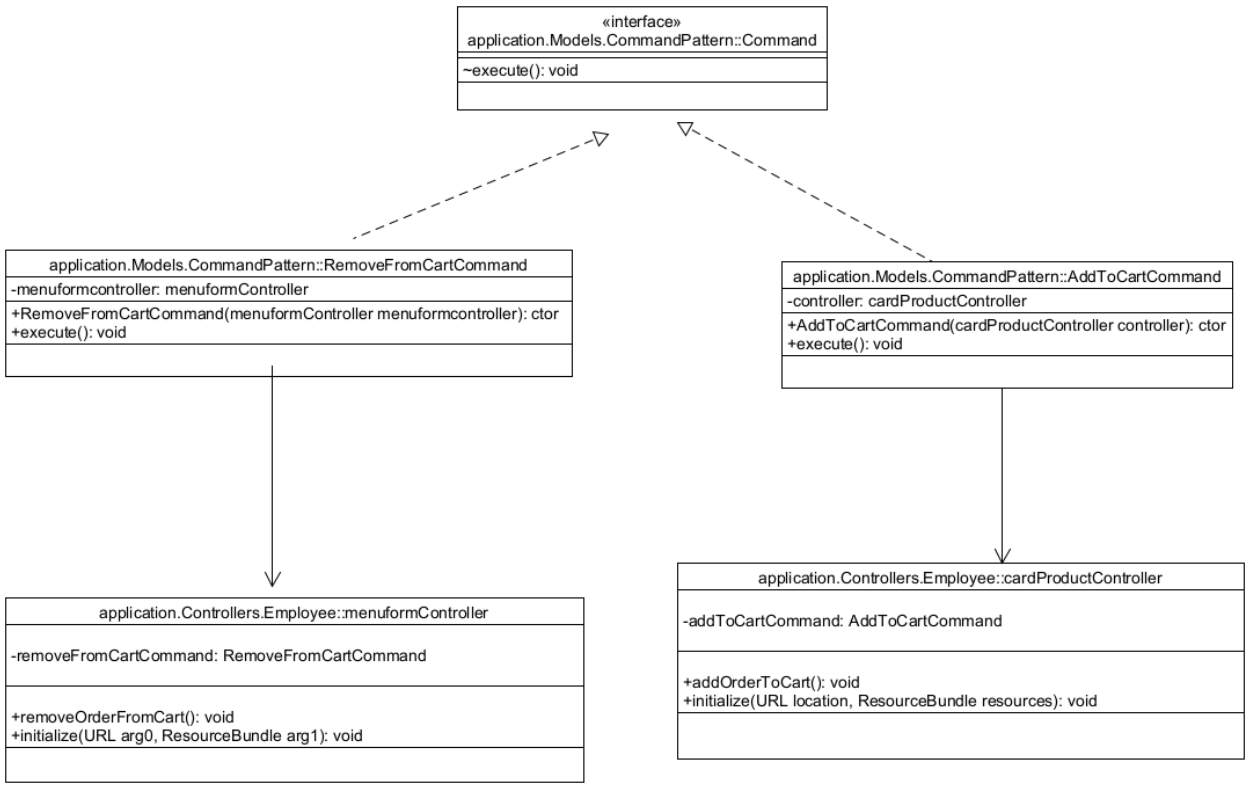
Nel cardProductController, il AddToCartCommand incapsula l'azione di aggiungere un prodotto al carrello. Quando l'utente preme il pulsante "Add", viene eseguito il comando AddToCartCommand, che invoca il metodo addOrderToCart(). Questo metodo esegue l'aggiunta del prodotto al carrello e gestisce vari controlli, come la verifica della disponibilità del prodotto, l'aggiornamento dello stock e la visualizzazione di alert in base allo stato dell'operazione. Inoltre, invoca il metodo preparaOrdine() della catena di gestione (chain) per preparare l'ordine.

Nel menuformController, il RemoveFromCartCommand incapsula l'azione di rimuovere un ordine dal carrello. Quando l'utente preme il pulsante "Remove", viene eseguito il comando RemoveFromCartCommand, che invoca il metodo removeOrderFromCart(). Questo metodo controlla se è stato selezionato un ordine, quindi lo rimuove dal carrello attraverso il DatabaseManagerMenu, aggiornando i dati visualizzati nella tabella.

Entrambi i controller hanno un'implementazione simile per l'esecuzione dei comandi: il metodo execute() viene chiamato quando si preme il pulsante corrispondente, separando l'azione specifica dal controller stesso. Questo rispetta il principio del design pattern Command, consentendo una maggiore flessibilità e disaccoppiamento delle azioni dall'interfaccia utente.

Inoltre, entrambi i controller usano l'interfaccia Command per implementare le diverse azioni, il che facilita l'estensione del sistema aggiungendo nuove azioni senza dover modificare il codice esistente.

Infine, entrambi i controller si collegano ai rispettivi oggetti Command, aggiungendo un gestore di eventi al pulsante corrispondente e eseguendo il comando quando il pulsante viene premuto.



Prototype

Il design pattern Prototype è un pattern creazionale ampiamente utilizzato nell'ingegneria del software per affrontare il problema della creazione efficiente di oggetti duplicati, noti come "cloni", di un oggetto esistente. Questo pattern è particolarmente idoneo quando l'istanziazione di oggetti è onerosa in termini di risorse computazionali o quando l'oggetto da duplicare è complesso da configurare. Il design pattern Prototype si concentra sulla creazione di nuovi oggetti basati su un prototipo esistente, consentendo così una duplicazione senza la necessità di conoscere i dettagli interni dell'oggetto in questione.

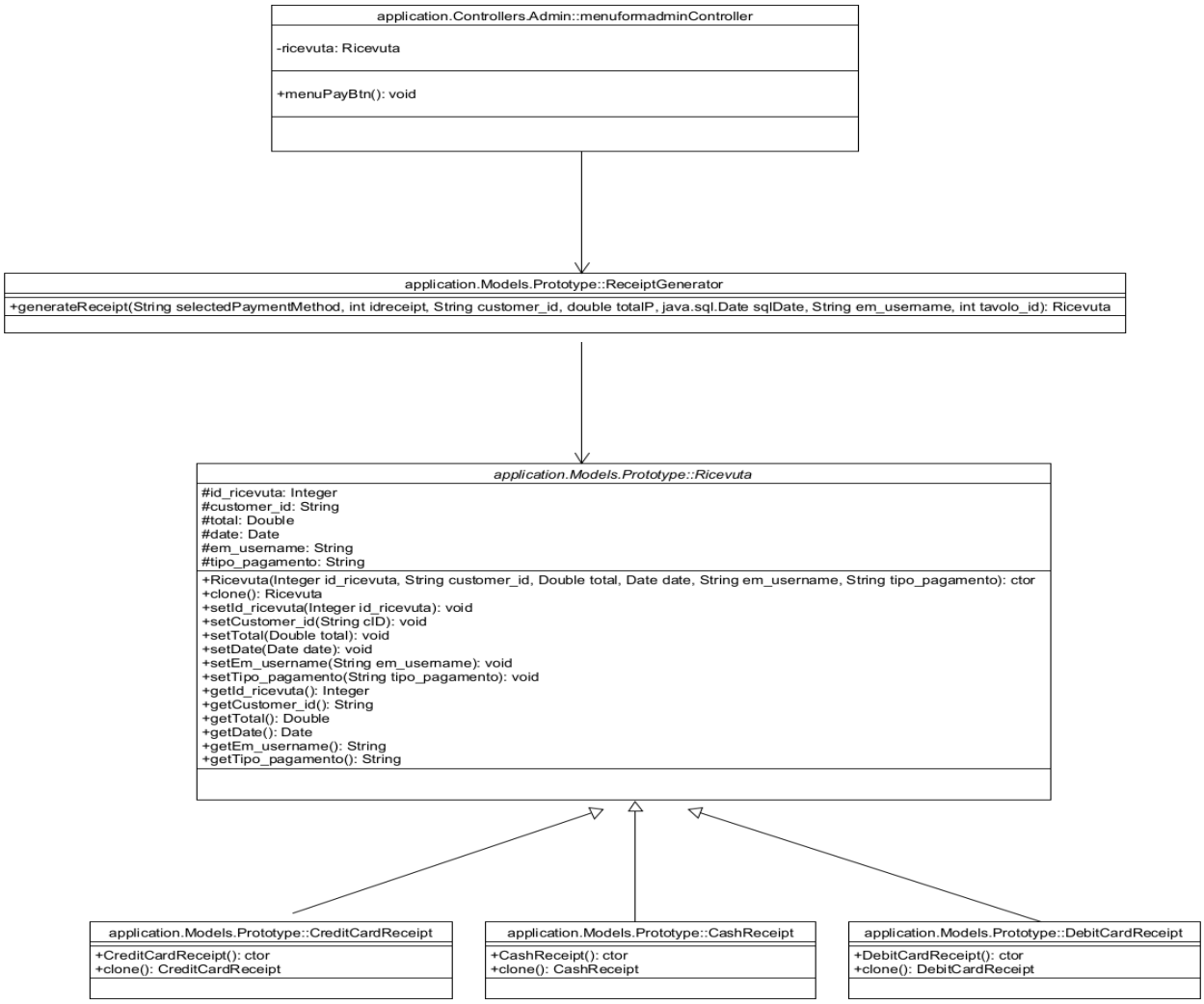
Il pattern Prototype è utilizzato per la generazione di diverse tipologie di ricevute, come Ricevute in Contanti, con Carta di Credito e con Carta di Debito.
La classe astratta Ricevuta definisce i campi principali e metodi che una ricevuta dovrebbe avere. Questa classe è il prototipo di base che definisce come dovrebbero essere le ricevute.
Le classi CashReceipt, CreditCardReceipt, e DebitCardReceipt estendono la classe

Ricevuta e rappresentano implementazioni specifiche di una ricevuta, corrispondenti ai diversi metodi di pagamento.

La classe ReceiptGenerator funge da factory per creare istanze delle varie tipologie di ricevuta in base al metodo di pagamento selezionato.

Nel controller menuformadminController, il metodo menuPayBtn() utilizza il ReceiptGenerator per creare una ricevuta basata sul metodo di pagamento scelto. Quando viene selezionato un metodo di pagamento, la classe ReceiptGenerator crea la corrispondente istanza di ricevuta tramite la chiamata a generateReceipt(). Questo metodo restituisce un'istanza della sottoclasse appropriata di Ricevuta, che viene assegnata alla variabile ricevuta.
Successivamente, il controller utilizza l'alert per mostrare una copia della ricevuta, consentendo all'utente di visualizzare i dettagli della transazione.

È importante sottolineare che la scelta di questo design pattern è stata presa in considerazione di future implementazioni quali avere la capacità di clonare le ricevute emesse nel programma ci potrebbe essere utile per scopi di gestione dei dati, stampe multiple, risoluzione di errori e per mantenere un registro storico delle transazioni passate. Questo approccio ci consente di conservare copie separare delle ricevute per vari scopi, contribuendo alla flessibilità e alla gestione efficiente delle operazioni nel sistema di gestione degli ordini e delle ricevute.



Chain of Responsibility

Il design pattern Chain of Responsibility è un pattern comportamentale utilizzato nell'ingegneria del software per creare una catena di oggetti responsabili del trattamento di richieste. Ogni oggetto all'interno della catena è in grado di processare la richiesta, trasmetterla all'oggetto successivo nella catena o decidere di interromperla. Questo pattern promuove la riduzione del codice accoppiato, la separazione delle responsabilità e la flessibilità nell'assegnazione delle operazioni.

Il Chain of Responsibility viene implementato per la preparazione degli ordini all'interno di un ristorante. Ci sono due gestori: CucinaHandler e BevandeHandler.

RepartoHandler è la classe astratta che definisce l'interfaccia per gestire l'ordine. Contiene un riferimento al prossimo gestore nella catena e un metodo per preparare

l'ordine, che controlla se il gestore attuale deve gestire il tipo di ordine e, in caso affermativo, calcola il tempo di preparazione e aggiorna il database.

CucinaHandler e BevandeHandler estendono RepartoHandler e implementano i metodi per gestire tipi specifici di ordini, nel primo caso "Meals" e nel secondo "Drinks".

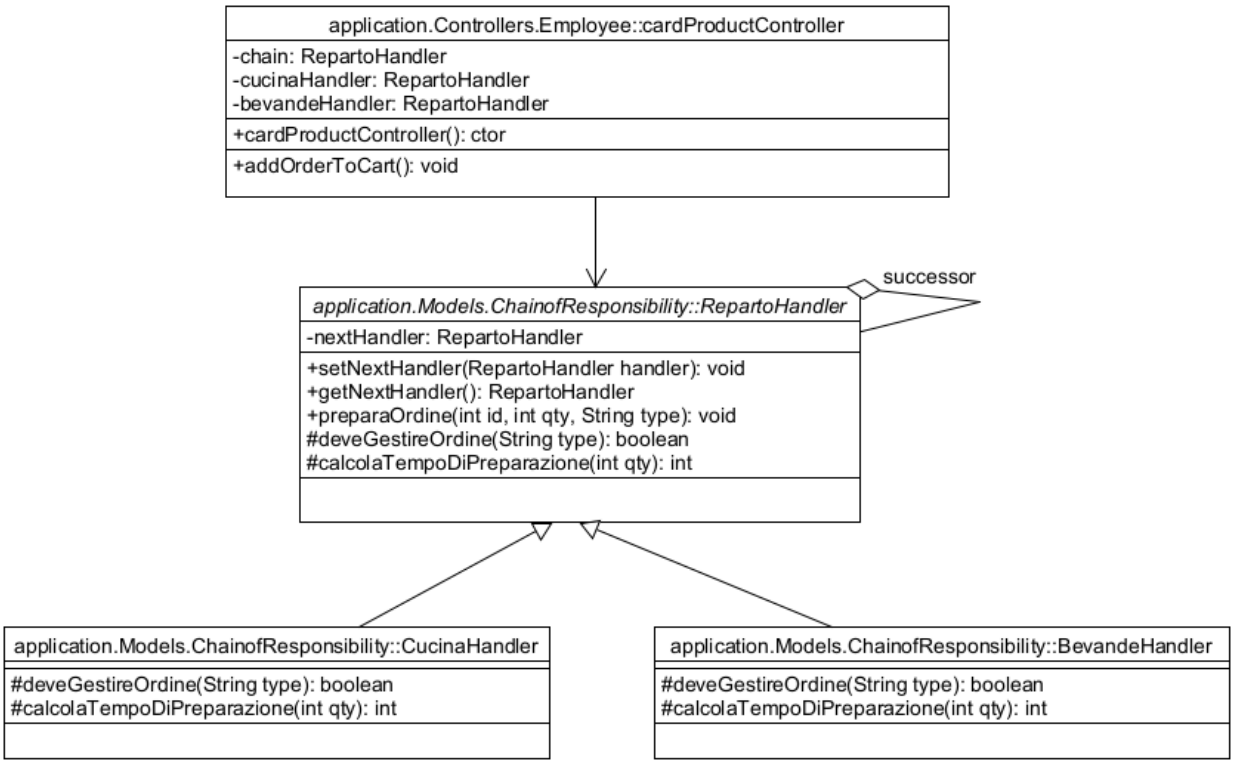
Nel controller cardProductController, viene inizializzata la catena di gestori nel costruttore. Quando viene aggiunto un ordine al carrello tramite il metodo addOrderToCart(), la richiesta viene passata alla catena di gestori utilizzando chain.preparaOrdine(customerID, qty, type);.

La catena di gestori, rappresentata da cucinaHandler e bevandeHandler, è organizzata in modo che se un tipo di ordine viene riconosciuto, il gestore appropriato elaborerà l'ordine e aggiornerà il database.

La catena di gestori utilizza il tempo di preparazione calcolato all'interno dei gestori specifici, come 5000 millisecondi per le bevande e i pasti.

Questo design pattern ci consente al controller di inoltrare la richiesta dell'ordine attraverso i vari gestori senza la necessità di conoscere direttamente il dettaglio di quale gestore gestirà quale tipo di ordine. Ogni gestore controlla se è in grado di gestire l'ordine in base al tipo e lo prepara, o altrimenti lo inoltra al gestore successivo.

Il Chain of Responsibility facilita la gestione modulare degli ordini senza creare dipendenze dirette tra il controller e i vari tipi di ordine o tra i gestori specifici. Ogni gestore è responsabile di decidere se può gestire l'ordine e, in caso contrario, passare l'ordine al successivo nella catena.



Struttura del Database del Ristorante

Il database del ristorante contiene diverse tabelle, ognuna con uno scopo specifico per gestire i dati relativi ai clienti, ai dipendenti, ai prodotti e agli ordini effettuati.

Ecco una panoramica delle tabelle e dei loro campi:

Tabella customer

Descrizione: Contiene informazioni sugli ordini effettuati dai clienti.

Campi:

- id: Identificativo univoco dell'ordine (Chiave primaria).
- customer_id: Identificativo del cliente, ossia il numero del tavolo.
- prod_id: Identificativo del prodotto ordinato.
- prod_name: Nome del prodotto.
- type: Tipo di prodotto (es. "Drinks" o "Meals").
- quantity: Quantità del prodotto ordinato.
- price: Prezzo del prodotto.
- date: Data dell'ordine.
- image: Percorso dell'immagine del prodotto.
- em_username: Username del dipendente che ha registrato l'ordine.

- stato: Stato dell'ordine (es. "Pagato").

Tabella employee

Descrizione: Contiene informazioni sui dipendenti del ristorante.

Campi:

- id: Identificativo univoco del dipendente (Chiave primaria).
- username: Username del dipendente.
- password: Password del dipendente.
- question: Domanda di sicurezza per il recupero della password.
- answer: Risposta alla domanda di sicurezza.
- date: Data di registrazione del dipendente.
- role: Ruolo del dipendente (es. "employee" o "administrator").

Tabella product

Descrizione: Contiene informazioni sui prodotti disponibili nel ristorante.

Campi:

- id: Identificativo univoco del prodotto (Chiave primaria).
- prod_id: Identificativo del prodotto.
- prod_name: Nome del prodotto.
- type: Tipo di prodotto (es. "Drinks" o "Meals").
- stock: Quantità disponibile in magazzino.
- price: Prezzo del prodotto.
- status: Stato del prodotto (es. "Available").
- image: Percorso dell'immagine del prodotto.
- date: Data di inserimento del prodotto.

Tabella receipt

- Descrizione: Contiene informazioni sulle ricevute e i pagamenti effettuati.
- Campi:
- id: Identificativo univoco della ricevuta (Chiave primaria).
- customer_id: Identificativo del cliente.
- total: Totale della ricevuta.
- date: Data della transazione.
- em_username: Username del dipendente che ha registrato la transazione.

- tipo_pagamento: Metodo di pagamento.

Le tabelle sono progettate per consentire la gestione dei dati relativi agli ordini effettuati dai clienti, alla gestione dei prodotti disponibili e alle transazioni finanziarie, con un'associazione ai dipendenti che hanno registrato tali dati.
Questa struttura permette una gestione integrata delle informazioni, consentendo ai gestori di monitorare gli ordini, la disponibilità di prodotti e le transazioni finanziarie.