TheKnife – Manuale Tecnico

Università degli Studi dell'Insubria

Dipartimento di Scienze Teoriche e Applicate

Laboratorio Interdisciplinare A - a.a. 2024/2025

Autori e ruoli del Progetto:

- Thomas Riotto Project manager - Matricola 760981 - Sede: VA
- Marco Zaro System architect -Matricola 760194 - Sede: VA
- Alessandro Tullo Design manager -Matricola 760760 - Sede: VA
- Antonio Pesavento Document&Quality manager -Matricola 759933 - Sede: VA

INDICE

1. Introduzione

TheKnife è un'applicazione sviluppata secondo le richieste del Laboratorio Interdisciplinare A del I anno di corso in Informatica 24/25 all'Università degli studi dell'Insubria di Varese., finalizzata alla gestione di recensioni per ristoranti. Il progetto è realizzato interamente in Java 24, utilizzando IntelliJ IDEA come IDE e testato su sistemi operativi Windows 10 e 11.

L'obiettivo del progetto è simulare un ambiente gestionale completo per utenti e ristoratori, integrando la geolocalizzazione, filtraggio avanzato e autenticazione sicura.

2. Struttura del progetto

TheKnife è suddiviso in quattro packages principali:

- entita: questo package contiene le seguenti classi principali del sistema gestionale di TheKnife:
  - Cliente (gestisce preferiti e recensioni)
  - Località
  - Recensione
  - RecensioneException
  - Ristorante
  - Ristoratore
  - TipoCucina
  - Utente (classe astratta)
  - UtenteException

- io_file: questo package contiene la sola classe "GestoreFile" che permette l'inserimento e la rimozione dei dati nei file CSV;

- servizi: il terzo package contiene i servizi fondamentali che consentono il corretto funzionamento dell'app:
  - GeocodingService
  - RecensioneService
  - RistoranteService
  - UtenteService

- vista: l'ultimo package del progetto contiene tutte le classi relative all'interfaccia utente e alla gestione dei menu e delle schermate di navigazione dell'app:
  - Menu
  - MenuCliente
  - MenuIniziale
  - MenuRistoratore
  - RegistrazioneService
  - TheKnife (parte iniziale dell'applicazione)

3.Librerie esterne utilizzate

Nel progetto sono state integrate tre librerie esterne per facilitare e migliorare la gestione di alcune funzionalità chiave. Di seguito viene fornita una breve descrizione del loro utilizzo specifico:

❖ *Gson-2.13.1*

- **Scopo**: Parsing e gestione di dati in formato JSON.
- **Utilizzo nel progetto**: La libreria Gson è stata utilizzata per il recupero e l'elaborazione di dati geografici forniti in formato JSON. Ad esempio, quando si interagisce con un'API che restituisce le coordinate geografiche o informazioni su città e indirizzi, Gson consente di convertire automaticamente il contenuto JSON in oggetti Java e viceversa.
- **Vantaggi**:
  - Serializzazione e de-serializzazione semplici e veloci.
  - Compatibilità con oggetti complessi e strutture annidate.

❖ *Jbcrypt-0.4*

- **Scopo**: Sicurezza e gestione delle password.
- **Utilizzo nel progetto**: La libreria JBCrypt è stata impiegata per memorizzare le password degli utenti in modo sicuro. In particolare, le password vengono hashate tramite l'algoritmo BCrypt prima di essere salvate nel sistema, così da garantire un elevato livello di protezione contro furti o accessi non autorizzati.
- **Vantaggi**:
  - Uso di **salt** automatico per ogni password, aumentando la sicurezza.
  - Resistenza agli attacchi di tipo brute-force e rainbow table.

❖ *Opencsv-5.9*

- **Scopo**: Lettura e scrittura di file CSV.
- **Utilizzo nel progetto**: La libreria OpenCSV è stata impiegata per gestire l'I/O di dati strutturati tramite file `.csv`, ad esempio per il salvataggio di informazioni su ristoranti, recensioni, o utenti. OpenCSV semplifica sia la lettura che la scrittura, permettendo di mappare direttamente i dati in oggetti Java tramite annotazioni o metodi dedicati.
  - **Vantaggi**:
  - Supporto per parsing robusto di file CSV anche complessi.
  - Facile integrazione: Disponibile nei repository Maven, quindi facilmente includibile nel progetto.

4.Dataset Utenti

Il sistema utilizza un dataset preconfigurato per simulare utenti reali (clienti e ristoratori). Ogni utente è dotato di:

- Username
- Password (in chiaro solo per test, normalmente cifrata)
- Dati anagrafici
- Ruolo nel sistema

Consigliata: gestione dei dati sensibili in file separati o ambiente protetto.

Questi sono gli utenti di prova che abbiamo utilizzato per il progetto. Le password sono state volutamente scritte in chiaro per effettuare il login con questi utenti già esistenti.

Nome	Cognome	Username	Password	Data di nascita	Indirizzo	Ruolo
Ristoratore	Ristoratore	ristoratore01	ristoratore01!	1980-12-01	Via Roma 1, Milano	Ristoratore
Cliente	Cliente	cliente01	cliente01!	1980-01-01	Viale Milano 115, Gallarate	Cliente
Marco	Rossi	marco_rossi	Pass123!	1985-03-15	Via Roma 25, Milano	Cliente
Giulia	Bianchi	giulia_bianchi	Sicura2024!	1990-07-22	Corso Vittorio 88, Roma	Ristoratore
Roberto	Ferrari	roberto_ferrari	Strong2024!	1980-09-30	Via Indipendenza 73, Bologna	Ristoratore
Elena	Russo	elena_russo	Elena123#	1995-01-18	Corso Francia 156, Torino	Cliente
Luca	Rossetti	lrossetti95	Pass@123	1995-08-21	Roma 2, Milano	Cliente
Pippo	Calzelunghe	pippocalze	1234abcd!	1910-02-28	Via Ottorino Rossi, Varese	Ristoratore

5.Entita

UTENTE:

Rappresenta l'identità base dell'applicazione, estendibile nei ruoli di Cliente o Ristoratore.

- Dati anagrafici: nome, cognome, username, data di nascita, luogo di domicilio.
- Cifratura sicura della password tramite algoritmo BCrypt.
- Formattazione automatica: nomi e luoghi sono capitalizzati (prima lettera maiuscola di ogni parola).
- Classe astratta da estendere con comportamenti specifici.

CLIENTE (ESTENDE UTENTE):

La classe **Cliente** rappresenta un utente del sistema che interagisce con i ristoranti in qualità di consumatore.

FUNZIONALITÀ PRINCIPALI:

Preferiti: Il cliente può costruire e mantenere una lista di ristoranti preferiti tramite i metodi `aggiungiPreferito()` e `rimuoviPreferito()`. La gestione garantisce l'assenza di duplicati e l'integrità della lista. Il metodo getPreferiti() consente l'accesso all'elenco attuale.

Recensioni: Ogni cliente ha la possibilità di recensire ristoranti visitati tramite aggiungiRecensione(), modificare le proprie valutazioni con modificaRecensione() e rimuoverle con rimuoviRecensione(). Il metodo haRecensito() consente di verificare la presenza di una recensione già registrata per un ristorante specifico, evitando inserimenti multipli.

INTEGRAZIONE CON ALTRE ENTITÀ

La classe si interfaccia con la logica interna dei ristoranti, delegando l'archiviazione e l'aggiornamento delle recensioni direttamente alla classe Ristorante. Questo assicura una gestione coerente tra cliente e locale.

RISTORATORE (ESTENDE UTENTE):

Ristoratore (estende Utente)

La classe Ristoratore rappresenta un'estensione specializzata della classe Utente, dedicata alla gestione dei locali di proprietà e all'interazione con i clienti attraverso le recensioni ricevute.

Funzionalità principali

- **Gestione dei ristoranti** Ogni istanza può possedere uno o più ristoranti, memorizzati in una lista interna.
  - Aggiunta: `aggiungiRistorante()`
  - Consultazione: `getRistoranti()`
  - Verifica proprietà: `possiede()` garantisce l'univocità e la corretta associazione del locale al ristoratore.
- **Interazione con le recensioni** Il ristoratore può rispondere alle recensioni ricevute dai clienti e modificarle successivamente.
  - Risposta: `rispondiARecensione()`
  - Modifica: `modificaRisposta()` L'identificazione della recensione avviene tramite il nome e la località del ristorante, per garantire precisione e integrità.
- **Analisi e sintesi** La panoramica delle valutazioni è consultabile tramite visualizzaRiepilogo(), che calcola la media ponderata delle recensioni ricevute per ciascun locale, fornendo una valutazione complessiva e aggiornata.
  - Il metodo privato `trovaRistorante()` consente operazioni di ricerca interna e rapida identificazione dell'entità desiderata.

Design e Sicurezza

La classe è pensata per offrire un'interfaccia coerente e controllata, garantendo che solo il ristoratore proprietario possa interagire con i dati sensibili delle proprie attività. La separazione netta dei ruoli tra cliente e gestore è rispettata in ogni metodo pubblico.

RISTORANTE:

La classe **Ristorante** rappresenta l'entità principale per la gestione dei locali all'interno del sistema. Ogni istanza identifica un ristorante registrato, associato a uno specifico proprietario (ristoratore) e caratterizzato da attributi descrittivi e funzionalità orientate alla valutazione da parte dei clienti.

Identificatori e attributi:

- **Nome univoco** del locale
- **Località**: coordinate geografiche ottenute tramite API o inserite manualmente
- **Tipologia di cucina**: categorizzazione gastronomica (es. italiana, giapponese, messicana...)
- **Servizi offerti**: possibilità di delivery e prenotazione
- **Prezzo medio** indicativo per cliente
- **Lista di recensioni** ricevute dai clienti
- **Proprietario**: istanza associata di Ristoratore

La chiave identificativa del ristorante è definita dal metodo `getChiave()`, combinando nome e località per garantire unicità.

Funzionalità principali:

- **Gestione delle recensioni**
  - Aggiunta: `aggiungiRecensione()`
  - Rimozione: `rimuoviRecensione()`
  - Modifica: `modificaRecensione()` Le recensioni sono tracciate nel tempo e gestite in modo controllato per assicurare coerenza.
- **Calcolo reputazione**
  - `getMediaStelle()` restituisce la media aggiornata delle valutazioni ricevute, utile per ordinamenti, filtri o report.
- **Gestione delle informazioni**
  - Aggiornamento dinamico dei dati descrittivi (nome, cucina, servizi, prezzo...)
  - Accesso ai dati tramite metodi getter: `getNome()`, `getLocalita()`, `getTipoCucina()`, `getDelivery()`, `getPrenotazione()`, `getPrezzoMedio()`, `getProprietario()`, `getRecensioni()`
- **Rappresentazione e comparazione**
  - `equals()` ridefinito per garantire l'unicità basata su nome e posizione
  - `toString()` ottimizzato per la visualizzazione in interfacce e documenti

Tutte le operazioni precedenti sono pensate per mantenere un alto grado di controllo e integrità, favorendo una gestione efficiente e sicura dei dati sensibili legati ai locali e alle interazioni con i clienti.

RECENSIONE:

La classe **Recensione** costituisce il nucleo funzionale per la valutazione dei ristoranti all'interno dell'applicazione, permettendo ai clienti di esprimere opinioni e punteggi in modo strutturato e gestibile.

Associazioni

Ogni istanza è univocamente collegata a:

- un cliente autore della recensione
- un ristorante destinatario

Attributi

- **Voto**: punteggio espresso in stelle da **1 a 5**
- **Testo**: messaggio descrittivo opzionale fornito dal cliente
- **Risposta**: replica del ristoratore, modificabile nel tempo

Funzionalità

- **Creazione e modifica**
  - `aggiungiRecensione()`
  - `modificaRecensione()`
  - `rimuoviRecensione()` (gestito dal Cliente)
- **Risposte del ristoratore**
  - `aggiungiRisposta()`
  - `modificaRisposta()` L'interazione con il campo risposta è vincolata e coerente, evitando sovrascritture accidentali.
- **Accesso ai dati**
  - `getVoto()`, `getTesto()`, `getCliente()`, `getRistorante()`, `getRisposta()`
- Unicità e rappresentazione
  - `equals()` ridefinito per garantire l'unicità della recensione basata su cliente e ristorante
  - `toString()` ottimizzato per l'integrazione con interfacce grafiche e report testuali

Ogni recensione conserva metadati temporali per garantire trasparenza nelle modifiche e favorire la revisione storica.

6. I/O su file

La classe **`GestoreFile`** fornisce un insieme di metodi statici per la gestione centralizzata delle operazioni di **I/O** su file CSV, relativi ai dati persistenti del sistema: utenti, ristoranti, recensioni e preferiti. Essa utilizza la libreria **OpenCSV** per il parsing e la scrittura dei file, convertendo ogni riga in oggetti del dominio applicativo.

Tutte le operazioni di lettura e ricerca hanno complessità lineare **`O(n)`** rispetto al numero di righe nel file, in quanto i dati vengono letti sequenzialmente in memoria senza alcuna indicizzazione. Le operazioni di inserimento, essendo realizzate tramite append, hanno complessità **`O(1)`**, mentre aggiornamenti ed eliminazioni richiedono la riscrittura completa del file e quindi presentano anch'esse complessità **`O(n)`**.

La classe è progettata per offrire una persistenza semplice e leggibile, ma non è adatta a grandi volumi di dati a causa delle inefficienze derivanti dalla gestione testuale dei CSV e dall'assenza di strutture dati ottimizzate per la ricerca o la modifica. Rappresenta un **trade-off** tra semplicità implementativa e performance, adatta per sistemi con dataset limitati o in fase di prototipazione.

7. Servizi

GeocodingService:

La classe `GeocodingService` è **un'utility statica** progettata per supportare la geolocalizzazione tramite conversione di indirizzi in coordinate geografiche (latitudine e longitudine), oppure per l'acquisizione manuale delle stesse da input utente. La classe è **non istanziabile**: tutti i metodi sono statici e il costruttore è privato.

La geocodifica automatica avviene tramite l'interrogazione di un servizio esterno basato su OpenStreetMap (Nominatim), mediante richiesta HTTP e parsing della risposta JSON. In caso di successo, il metodo restituisce un array `double[2]` contenente latitudine e longitudine. In presenza di errori di rete, formato errato o assenza di risultati, il metodo restituisce `null`.

In alternativa, è possibile acquisire manualmente le coordinate tramite console, con validazione dell'input numerico e gestione della terminazione anticipata tramite parola chiave. Anche in questo caso, il risultato è restituito come array `double[2]`.

Dal punto di vista computazionale:

- la geocodifica automatica ha complessità `O(n)` per la codifica dell'indirizzo e `O(m)` per il parsing della risposta JSON, con il **tempo di rete** come principale fattore limitante.
- l'acquisizione manuale ha complessità trascurabile dal punto di vista computazionale, essendo dominata dall'interazione con l'utente. Il ciclo di validazione ha complessità `O(t * k)`, dove `t` è il numero di tentativi e `k` la lunghezza dell'input.

La rappresentazione compatta tramite array consente una gestione semplice ed efficiente delle coordinate. La classe è pensata per garantire robustezza nell'input/output geografico, con fallback manuale in caso di indisponibilità del servizio remoto.

RECENSIONESERVICE:

La classe **RecensioneService** fornisce una serie di **metodi statici** per la gestione delle recensioni nell'ambito dell'applicazione. Opera come strato di servizio, fungendo da **intermediario** tra la logica applicativa e la persistenza su file CSV, delegata alla classe **GestoreFile**.

Le sue responsabilità comprendono l'aggiunta, modifica, cancellazione, visualizzazione e validazione delle recensioni, nonché la gestione delle risposte dei ristoratori. Alcune operazioni prevedono anche il caricamento e l'assegnazione in memoria delle recensioni ai relativi ristoranti.

Dal punto di vista computazionale, le operazioni di caricamento e ricerca di recensioni ereditano la complessità lineare **`O(n)`** dai metodi di lettura dei file CSV. Anche operazioni di aggiornamento o rimozione comportano lettura e riscrittura completa del file, mantenendo la complessità `O(n)` rispetto al numero di recensioni presenti.

La gestione in memoria avviene tramite strutture come **`HashMap<String, Ristorante>`** per una rapida associazione delle recensioni ai ristoranti tramite chiavi univoche, portando a una complessità **`O(1)`** per operazioni di accesso in mappa. Tuttavia, l'efficienza generale rimane vincolata alle limitazioni strutturali della persistenza testuale.

L'approccio seguito è adatto a dataset contenuti, ed è orientato alla **chiarezza** e **modularità** piuttosto che alle prestazioni su larga scala.

RISTORANTESERVICE:

La classe `RistoranteService` fornisce un insieme di metodi statici per la gestione delle operazioni sui ristoranti, tra cui ricerca filtrata, aggiunta di ristoranti, caricamento recensioni e interazione con l'utente tramite input da console. Funziona come strato di servizio, delegando alla classe `GestoreFile` le operazioni di persistenza su file CSV. La funzionalità principale è il metodo di **ricerca avanzata** che consente l'applicazione combinata di filtri opzionali (tipologia cucina, fascia di prezzo, delivery, prenotazione, media stelle, distanza geografica) su una lista di ristoranti caricata interamente in memoria. Ogni filtro viene applicato in sequenza, mantenendo la complessità di ciascuna iterazione a `O(1)`, con complessità complessiva `O(n)` dove n è il numero di ristoranti presenti nel file. Le operazioni di caricamento da file (ristoranti, recensioni) ereditano anch'esse complessità `O(n)`. La classe include logica di validazione sui parametri numerici e geografici, oltre a gestire l'interazione utente in modalità testuale.

La progettazione privilegia chiarezza e modularità piuttosto che scalabilità, risultando adatta a dataset di piccole o medie dimensioni, in ambienti desktop o di prototipazione.

UTENTESERVICE:

La classe `UtenteService` fornisce metodi statici per la gestione delle operazioni relative agli utenti del sistema, fungendo da **interfaccia tra la logica di business e la persistenza su file CSV**. La classe è progettata come utility non istanziabile, con metodi statici e costruttore privato.

Le funzionalità principali comprendono:

- **Registrazione di un nuovo utente** (Cliente o ristoratore) mediante inserimento nei file persistenti.
- **Autenticazione** tramite verifica di username e password cifrata, con successivo caricamento dei dati utente associati (preferiti o ristoranti gestiti) e delle relative recensioni.
- **Gestione dei preferiti** per i clienti, tramite aggiunta o rimozione dei ristoranti dalla lista personale e aggiornamento dei dati persistenti.

Le operazioni di lettura e scrittura interagiscono con la classe `GestoreFile` e hanno complessità lineare **`O(n)`** rispetto alla dimensione dei file CSV coinvolti. L'uso di strutture `ArrayList` per la gestione in memoria garantisce accesso rapido ai dati utente dopo il caricamento iniziale. La ricostruzione completa dell'oggetto utente autenticato consente di garantire consistenza e isolamento tra i dati persistiti e quelli usati in sessione.

L'approccio seguito è orientato alla chiarezza e all'integrità dei dati caricati, rendendo la classe adatta a sistemi di dimensioni contenute con accesso utente centralizzato.

8. vista

Le classi `Menu`, `MenuCliente`, `MenuRistoratore` e `MenuIniziale` costituiscono l'interfaccia testuale dell'applicazione, progettata per gestire l'interazione utente in base al ruolo.

La classe **`Menu`** è astratta e definisce la struttura comune per tutti i menu, mentre le sottoclassi concretizzano la logica specifica di ciascun contesto: cliente, ristoratore o utente non autenticato. Ogni menu è implementato come **ciclo sincrono** che presenta un elenco di opzioni numerate, riceve input da console tramite `Scanner`, esegue la validazione dell'input e richiama i servizi applicativi corrispondenti (`UtenteService`, `RistoranteService`, `RecensioneService`, ecc.). L'input viene gestito in modo robusto, con messaggi di errore e la possibilità di interrompere l'operazione attraverso parole chiave predefinite (es. "stop").

Le operazioni disponibili includono la ricerca filtrata di ristoranti, la gestione di recensioni e preferiti, l'aggiunta o la rimozione di contenuti e l'autenticazione utente.

Tutte le elaborazioni sono demandate ai servizi sottostanti, mantenendo i menu focalizzati sulla logica di presentazione e controllo del flusso.

Dal punto di vista computazionale, la complessità delle operazioni dipende dai servizi richiamati:

- La navigazione nei menu ha complessità costante **`O(1)`** per ogni iterazione.
- Le ricerche e i filtraggi operano su liste lineari, con complessità **`O(n)`** rispetto alla dimensione del dataset (ristoranti, recensioni).
- Le operazioni di scrittura su file (CSV) hanno complessità **`O(k)`**, dove k è il numero di righe coinvolte.

La struttura dei menu è progettata per essere modulare, estensibile e adatta a contesti di dimensioni contenute, garantendo una buona esperienza utente anche in assenza di interfaccia grafica.

9.Algoritmi principali

`cercaRistorante()`

Il metodo `cercaRistorante` permette di cercare ristoranti applicando una serie di filtri opzionali. Tutti i parametri possono essere nulli, ad eccezione della località che è obbligatoria. I ristoranti vengono selezionati solo se soddisfano **tutti** i criteri specificati.

Funzionamento

1. **Validazione parametri**
   Prima di eseguire la ricerca, il metodo effettua controlli sui parametri:
   a. La località non può essere `null`.
   b. I valori numerici (prezzi, media stelle, raggio) devono essere coerenti e validi:
      i. `prezzoMinimo` e `prezzoMassimo` non devono essere negativi.
      ii. `prezzoMinimo` non può essere maggiore di `prezzoMassimo`.
      iii. `mediaStelle`, se presente, deve essere compresa tra 1.0 e 5.0.
      iv. `raggioKm`, se presente, deve essere positivo.
2. **Caricamento ristoranti**
   I ristoranti vengono caricati da file tramite il metodo `GestoreFile.caricaRistoranti()`. Questo può sollevare eccezioni di tipo `IOException` o `CsvException`.
3. **Applicazione dei filtri**
   I ristoranti vengono filtrati uno per uno. Per ogni ristorante, vengono applicati i seguenti filtri, in ordine:
   a. Tipo di cucina
   b. Località (con eventuale raggio geografico)
   c. Prezzo minimo e massimo
   d. Disponibilità del servizio di delivery
   e. Disponibilità della prenotazione online
   f. Media stelle
4. Se un ristorante non soddisfa un filtro, viene immediatamente escluso.

Risultato
I ristoranti che soddisfano tutti i criteri vengono aggiunti a una lista che viene restituita come risultato. Se nessun ristorante corrisponde, la lista sarà vuota.

Complessità

- **Temporale**: O(n), dove *n* è il numero totale di ristoranti caricati. Ogni ristorante viene analizzato una sola volta, e ciascun filtro ha costo costante.
- **Spaziale**: O(k), dove *k* è il numero di ristoranti che soddisfano i criteri e sono aggiunti al risultato.

Vantaggi

- Modularità e leggibilità: ogni filtro è separato e facilmente modificabile.
- Efficienza: l'uso di `continue` permette di scartare rapidamente i ristoranti non compatibili.
- Estendibilità: è possibile aggiungere nuovi filtri in modo semplice senza alterare la logica principale.

`VisualizzaDettagliRistorante()`

Il metodo `visualizzaDettagliRistorante` consente di visualizzare i dettagli di un ristorante selezionato da una lista di risultati. È pensato per un'interfaccia testuale (console) e fornisce all'utente un modo semplice per accedere alle informazioni complete di un singolo ristorante.

Funzionamento

1. **Selezione del ristorante**
   Il metodo richiama `selezionaRistorante(risultati)`, che consente all'utente di scegliere un ristorante dalla lista. La funzione restituisce un indice intero corrispondente alla posizione dell'elemento selezionato.
2. **Verifica dell'indice**
   Si verifica che l'indice restituito sia valido, cioè compreso tra `0` (incluso) e `risultati.size()` (escluso). Se non è valido, la funzione termina senza eseguire ulteriori azioni.
3. **Visualizzazione dei dettagli**
   Se l'indice è valido, viene stampato su console il dettaglio del ristorante corrispondente. L'output sfrutta il metodo `toString()` della classe `Ristorante`, che deve essere sovrascritto per fornire una descrizione informativa del ristorante (nome, indirizzo, tipo cucina, valutazioni, ecc.).

Complessità

- **Temporale**: O(1) – la selezione e la visualizzazione avvengono in tempo costante.
- **Spaziale**: O(1) – non vengono allocate strutture dati aggiuntive.

Considerazioni

- Il metodo presuppone che `selezionaRistorante(...)` sia definito e funzionante.
- Il metodo è progettato per ambienti console.
- È utile in combinazione con una lista di risultati filtrati o ricercati in precedenza.

`VisualizzaRecensioni()`

Il metodo `visualizzaRecensioni` consente a un ristoratore di visualizzare le recensioni disponibili per i propri ristoranti. L'utente può selezionare uno dei ristoranti per visualizzarne nel dettaglio i giudizi ricevuti.

Funzionamento

1. **Messaggio iniziale e recupero ristoranti**
    a. Il metodo stampa un'intestazione di sezione.
    b. Recupera la lista dei ristoranti associati al ristoratore tramite `ristoratore.getRistoranti()`.
2. **Verifica della presenza di ristoranti**
    a. Se la lista è nulla o vuota, informa l'utente che non ci sono ristoranti registrati e termina.
3. **Filtro dei ristoranti con recensioni**
    a. Scorre la lista dei ristoranti per individuare solo quelli che contengono almeno una recensione (`haRecensioni()`).
    b. Crea una lista filtrata contenente solo questi ristoranti.
4. **Verifica della presenza di recensioni**
    a. Se nessun ristorante ha recensioni, il metodo lo comunica all'utente e termina.
5. **Presentazione dei ristoranti recensiti**
    a. Mostra un elenco numerato dei ristoranti con recensioni, includendo:
        i. Nome del ristorante
        ii. Numero di recensioni disponibili
        iii. Media delle stelle (valutazione media)
6. **Selezione dell'utente**
    a. L'utente può scegliere un ristorante da visualizzare oppure digitare `0` per tornare al menu principale.
    b. Se la selezione è fuori dai limiti, viene mostrato un messaggio di errore.
7. **Visualizzazione delle recensioni**

Se la selezione è valida, il metodo chiama `visualizzaRecensioniRistorante(...)` per mostrare tutte le recensioni del ristorante selezionato.

Complessità

- **Temporale**:
  - O(n) per filtrare i ristoranti con recensioni, dove *n* è il numero totale di ristoranti del ristoratore.
  - O(m) per stampare i ristoranti filtrati, dove *m* è il numero di ristoranti con almeno una recensione.
- **Spaziale**: O(m), dove *m* è la dimensione della lista dei ristoranti con recensioni.

Considerazioni

- Il metodo è pensato per l'interazione da console.
- Offre una buona esperienza utente grazie a messaggi chiari e gestione degli input non validi.

`Registrazione()`

Il metodo `registrazione` permette di avviare il processo di registrazione di un nuovo utente nel sistema, interagendo direttamente con l'utente via console. Il metodo guida l'utente nella compilazione dei dati richiesti, salva i dati nel file CSV e indirizza l'utente al menu appropriato (cliente o ristoratore) al termine della procedura.

Funzionamento

Inizio registrazione:

Il metodo richiama `registrazioneService.registraUtente()` per avviare il processo di registrazione tramite console. Questo metodo gestisce l'interazione completa con l'utente per l'inserimento e la validazione dei dati (nome, cognome, username, password, data di nascita, domicilio e ruolo).
Se l'utente inserisce "STOP" in qualsiasi punto, la registrazione viene annullata e viene mostrato un messaggio.

Salvataggio dell'utente:

Se l'oggetto `Utente` creato non è `null`, si tenta di registrarlo persistendo i dati su file CSV tramite `UtenteService.registraUtente(nuovoUtente)`.

In caso di errore di I/O o di scrittura CSV, viene stampato un messaggio di errore specifico e il metodo termina.
Se il salvataggio fallisce (ad esempio per duplicazione username o errori logici nel servizio), il metodo lo comunica chiaramente e termina.

Conferma e indirizzamento al menu:

Se la registrazione va a buon fine, viene mostrato un messaggio di conferma. Successivamente, il metodo verifica il tipo dell'oggetto `nuovoUtente` utilizzando il pattern matching `instanceof` e avvia il rispettivo menu interattivo:

- Se è un `Cliente`, viene istanziato e mostrato un oggetto `MenuCliente`.
- Se è un `Ristoratore`, viene istanziato e mostrato un oggetto `MenuRistoratore`.

Complessità

- **Temporale**:
  - O(1) per la maggior parte delle operazioni principali.
  - O(n) per la verifica dell'unicità dello username durante la chiamata a `esisteUtente`, dove n è il numero di utenti registrati.
- **Spaziale**:
  - O(1), salvo per l'oggetto `Utente` creato temporaneamente.

Considerazioni

- Il metodo è pensato per l'interazione da console, con una forte attenzione alla **robustezza dell'input**.
- L'uso del pattern matching con `instanceof` migliora la leggibilità del codice e semplifica il flusso di selezione del menu.
- È presente una gestione completa dei possibili errori:
  - Input annullato (STOP)
  - Fallimento della registrazione
  - Errori di I/O e CSV
- Il codice rispetta la **responsabilità singola**: delega la logica di raccolta dati a `RegistrazioneService` e quella di salvataggio a `UtenteService`.

Punti di forza

- Codice ben strutturato e leggibile.
- Elevata modularità: ogni operazione è demandata al servizio corretto.
- Ottima gestione dei flussi alternativi (annullamento, errore, successo).

Possibili miglioramenti

- Potrebbe essere utile loggare su file gli errori di registrazione, specialmente quelli di I/O.
- Separare la logica di selezione del menu in un metodo dedicato per ridurre la complessità della funzione.
- Introdurre un sistema di tentativi massimi per evitare loop infiniti in caso di input non valido.

Conclusione

Il metodo `registrazione` è un esempio ben progettato di gestione di input utente in un'applicazione console-based. È strutturato in modo modulare, robusto e chiaro, offrendo un'esperienza fluida e interattiva all'utente finale.

`aggiungiPreferito`

Il metodo `aggiungiPreferito` consente di aggiungere un ristorante alla lista dei preferiti di un cliente, sia a livello logico (in memoria) sia a livello persistente (su file CSV). Ritorna `true` solo se l'operazione viene completata con successo in entrambi i livelli.

Il metodo verifica inizialmente che i parametri `cliente` e `preferito` non siano null. In tal caso, restituisce `false` per evitare errori durante l'esecuzione.

Successivamente, invoca `GestoreFile.aggiungiPreferito(cliente, preferito)`, metodo responsabile della persistenza dell'associazione tra cliente e ristorante preferito. Se la scrittura su file fallisce, il metodo restituisce immediatamente `false`.

Se invece la scrittura ha successo, viene aggiornata anche la struttura dati in memoria del cliente, tramite il metodo `cliente.aggiungiPreferito(preferito)`. Il risultato di questa ultima operazione viene infine restituito come esito finale.

Il metodo è robusto e segue una logica transazionale: la modifica viene applicata in memoria solo dopo il successo della scrittura persistente, riducendo il rischio di inconsistenze. Rispetta inoltre il principio di responsabilità singola, delegando la gestione dei file a una classe dedicata.

Complessità

- **Temporale**:
  - O(1) per il controllo dei parametri null.
  - O(1) per `GestoreFile.aggiungiPreferito(...)`,
  - O(1) per `cliente.aggiungiPreferito(...)`, dove *p* è la complessità dell'inserimento nella struttura dati (tipicamente O(1) se è una lista o un set).
- **Spaziale**:
  - O(1), poiché non vengono allocate strutture dati aggiuntive significative.

`rimuoviPreferito`

Il metodo `rimuoviPreferito` consente di eliminare un ristorante dalla lista dei preferiti di un cliente, sia a livello persistente (su file CSV) sia a livello logico (in memoria). Restituisce `true` solo se entrambe le operazioni hanno successo.

Il metodo inizia controllando se i parametri `cliente` e `preferito` sono `null`. In tal caso, ritorna `false` per evitare comportamenti imprevisti.

Successivamente, viene chiamato il metodo `GestoreFile.rimuoviPreferito(cliente, preferito)`, incaricato della rimozione dell'associazione dal file CSV. Se tale operazione fallisce, il metodo restituisce immediatamente `false`.

Solo se la rimozione dal file va a buon fine, il metodo procede a rimuovere il ristorante dalla lista dei preferiti del cliente in memoria tramite `cliente.rimuoviPreferito(preferito)`. Il valore restituito da quest'ultima operazione viene infine ritornato.

La struttura del metodo è progettata per mantenere coerenza tra il livello persistente e quello in memoria, seguendo una logica transazionale: prima si modifica il file, poi si aggiorna la struttura dati in memoria. Inoltre, rispetta il principio di separazione delle responsabilità, delegando la gestione dei file a una classe apposita.

Complessità

- **Temporale**:
  - O(1) per il controllo dei parametri null.
  - O(f) per `GestoreFile.rimuoviPreferito(...)`, dove *f* è la complessità della rimozione da file (tipicamente O(n), dove *n* è il numero di righe nel CSV).
  - O(p) per `cliente.rimuoviPreferito(...)`, dove *p* è la complessità della rimozione dalla struttura dati interna (tipicamente O(n) per liste, O(1) per set o mappe).
  - Totale: **O(f + p)**.
- Spaziale:
  - O(1), in quanto non vengono allocate strutture dati ausiliarie significative.

`visualizzaPreferiti`

Il metodo `visualizzaPreferiti` ha lo scopo di mostrare a video l'elenco dei ristoranti preferiti associati a un determinato cliente. È un metodo privato e presumibilmente viene utilizzato all'interno di un'interfaccia utente testuale.

Il metodo comincia stampando un'intestazione fissa. Recupera poi la lista dei ristoranti preferiti del cliente tramite `cliente.getPreferiti()` e ne verifica il contenuto. Se la lista è vuota, viene stampato un messaggio informativo all'utente e il metodo restituisce `false`.

Se invece la lista contiene almeno un ristorante, il metodo itera su di essa e stampa ciascun elemento, numerandolo a partire da 1. Al termine, restituisce `true` per indicare che ci sono stati preferiti da visualizzare.

La logica è semplice e orientata all'utente, con un output chiaro e leggibile. Inoltre, il valore booleano restituito può essere utile ad altri metodi per sapere se ci sono preferiti da mostrare.

Complessità

- **Temporale**:
  - O(1) per la stampa iniziale e il controllo se la lista è vuota.
  - O(n) per l'iterazione e la stampa di ciascun ristorante, dove *n* è il numero di ristoranti preferiti del cliente.
  - Totale: **O(n)**.
- **Spaziale**:
  - O(1), poiché non vengono create strutture dati ausiliarie; l'unico spazio occupato è quello già usato per la lista dei preferiti.

`aggiungiRecensione`

Il metodo `aggiungiRecensione` consente a un cliente di aggiungere una recensione a un ristorante. La funzione prevede diversi controlli di validità prima di effettuare l'operazione.

In primo luogo, viene verificato che nessuno dei parametri (`cliente`, `ristorante`, `recensione`) sia nullo. Se uno di questi è nullo, il metodo restituisce `false`.

Successivamente, viene controllato che il cliente che sta tentando di aggiungere la recensione sia effettivamente colui che l'ha scritta (`recensione.getCliente().equals(cliente)`) e che il ristorante della recensione corrisponda al ristorante passato come parametro. Se questi vincoli non sono rispettati, l'operazione viene interrotta.

Una volta superati i controlli, la recensione viene salvata persistentemente attraverso il metodo `GestoreFile.aggiungiRecensione`. Se questa operazione fallisce, viene restituito `false`.

Infine, la recensione viene aggiunta anche alla struttura interna del cliente, tramite `cliente.aggiungiRecensione`, che dovrebbe mantenere una lista delle recensioni effettuate.

Complessità

- **Temporale**:
  - I controlli di nullità e uguaglianza sono O(1), dato che si presume siano semplici confronti o accessi a riferimenti.
  - L'operazione `GestoreFile.aggiungiRecensione` dipende dall'implementazione interna della scrittura su file, ma si può approssimare come **O(1).**
  - `cliente.aggiungiRecensione` si presume essere O(1) se aggiunge in una lista, o **O(n)** se verifica l'assenza di duplicati.
  - Totale: **O(n)** nel caso peggiore.
- **Spaziale**:
  - O(1), dato che non vengono allocate strutture dati aggiuntive rilevanti in memoria.

`modificaRecensione`

Il metodo `modificaRecensione` permette a un cliente di modificare una recensione precedentemente scritta per un determinato ristorante.

L'operazione inizia con una serie di controlli:

- Verifica che nessuno dei parametri (`cliente`, `ristorante`, `nuovaRecensione`) sia nullo.
- Controlla che il cliente della nuova recensione corrisponda al cliente passato come parametro, e che il ristorante della nuova recensione sia coerente con quello passato.

Superati questi controlli, viene caricata in memoria la lista delle recensioni del ristorante tramite `caricaRecensioniRistorante(ristorante)` (probabilmente da file).

Successivamente, si cerca la vecchia recensione del cliente su quel ristorante attraverso `ristorante.trovaRecensioneCliente(cliente)`. Se non esiste alcuna recensione precedente, l'operazione fallisce.

Se invece viene trovata, si prova ad aggiornare il contenuto nel file CSV tramite `GestoreFile.aggiornaRecensione`. Se la modifica a livello di file ha successo, si procede ad aggiornare anche la struttura interna del cliente con `cliente.modificaRecensione`.

Complessità

- **Temporale**:
  - I controlli iniziali sono O(1).
  - `caricaRecensioniRistorante` potrebbe avere una complessità **O(n)**, dove *n* è il numero di recensioni nel file.
  - `ristorante.trovaRecensioneCliente` potrebbe essere **O(m)**, dove *m* è il numero di recensioni associate al ristorante.
  - `GestoreFile.aggiornaRecensione` potrebbe anch'esso essere **O(n)**, a seconda dell'implementazione (ad esempio, riscrittura parziale/totale del file).
  - `cliente.modificaRecensione` si assume **O(1)** o **O(k)**, dove *k* è il numero di recensioni del cliente.
  - Totale: **O(n + m)** nel caso peggiore.
- **Spaziale**:
  - O(1) se le recensioni vengono caricate una alla volta o tramite iterazione. Potrebbe diventare **O(n)** se tutte le recensioni vengono caricate in memoria contemporaneamente.

`eliminaRecensione`

Il metodo `eliminaRecensione` permette a un cliente di eliminare una recensione precedentemente scritta su un determinato ristorante.

L'algoritmo segue questi passi:

1. **Validazione input**:
   a. Controlla che `cliente` e `ristorante` non siano nulli.
2. **Caricamento recensioni**:
   a. Chiama `caricaRecensioniRistorante(ristorante)` per assicurarsi che le recensioni del ristorante siano caricate in memoria.
3. **Ricerca recensione del cliente**:
   a. Recupera la recensione scritta dal cliente per quel ristorante con `ristorante.trovaRecensioneCliente(cliente)`.
   b. Se non esiste una recensione, ritorna `false`.
4. **Eliminazione fisica del file**:
   a. Chiama `GestoreFile.eliminaRecensione(recensione)` per rimuovere la recensione dal file CSV.
   b. Se l'eliminazione fallisce, ritorna `false`.
5. **Rimozione dalla struttura dati del cliente**:
   a. Rimuove la recensione dalla lista interna del cliente con `cliente.rimuoviRecensione(ristorante)`.
   b. Ritorna `true` se questa operazione ha successo.

Complessità

- **Temporale**:
  - Controlli iniziali sono **O(1)**.
  - `caricaRecensioniRistorante` potrebbe essere **O(m)**, con *m* numero di recensioni del ristorante.
  - `ristorante.trovaRecensioneCliente(cliente)` è tipicamente **O(k)**, dove *k* è il numero di recensioni del ristorante.
  - `GestoreFile.eliminaRecensione` è **O(n)**, con *n* numero totale di recensioni nel file, poiché potrebbe dover riscrivere il file CSV.
  - `cliente.rimuoviRecensione` presumibilmente è **O(1)** o **O(k')**, dove *k'* è il numero di recensioni del cliente, dipendente dall'implementazione.

Totale: Dominato da **O(n)** a causa della manipolazione del file.

- **Spaziale**:
  - O(m) per il caricamento delle recensioni del ristorante, se necessario.
  - O(1) aggiuntivo per le operazioni rimanenti.

`aggiungiRistorante`

Il metodo `aggiungiRistorante` consente a un ristoratore di aggiungere un nuovo ristorante alla piattaforma.

L'algoritmo segue questi passi:

1. **Validazione input**:
   a. Verifica che `ristoratore` e `ristorante` non siano nulli.
2. **Verifica autorizzazione**:
   a. Controlla che il proprietario del ristorante (`ristorante.getProprietario()`) corrisponda al ristoratore che sta effettuando l'operazione.
3. **Aggiunta al file**:
   a. Chiama `GestoreFile.aggiungiRistorante(ristorante)` per scrivere il ristorante nel file CSV.
   b. Se l'aggiunta fallisce, ritorna `false`.
4. **Aggiornamento struttura dati**:
   a. Aggiunge il ristorante alla lista interna del ristoratore con `ristoratore.aggiungiRistorante(ristorante)`.
   b. Ritorna `true` se l'operazione ha successo.

Complessità

- **Temporale**:
  - I controlli nulli e di uguaglianza sono O(1).
  - L'aggiunta al file tramite `GestoreFile.aggiungiRistorante` è O(1)
  - L'aggiunta alla lista interna del ristoratore è tipicamente O(1).

Totale: Generalmente dominato da O(n) dovuto alla scrittura nel file.

- **Spaziale**:
  - O(1) aggiuntivo, salvo spazio per memorizzare il nuovo ristorante.

`visualizzaRiepilogo`

Il metodo `visualizzaRiepilogo` mostra un riepilogo aggregato delle recensioni ricevute da tutti i ristoranti gestiti da un ristoratore.

Funzionamento:

1. **Inizializzazione**:
   a. `totaleRecensioni`: conta il numero complessivo di recensioni ricevute da tutti i ristoranti.
   b. `sommaStelle`: accumula la somma ponderata delle stelle, calcolata come media delle stelle di ciascun ristorante moltiplicata per il numero di recensioni di quel ristorante.
2. **Ciclo sui ristoranti**:
   a. Per ogni ristorante nella lista `ristoranti`:
      i. Ottiene il numero di recensioni con `ristorante.getNumeroRecensioni()`.
      ii. Incrementa `totaleRecensioni` di questo valore.
      iii. Calcola il contributo alla somma delle stelle moltiplicando la media delle stelle del ristorante per il numero delle sue recensioni, sommando il risultato a `sommaStelle`.
3. **Stampa risultato**:
   a. Se non ci sono recensioni totali (`totaleRecensioni == 0`), stampa un messaggio indicante l'assenza di recensioni.

Altrimenti, calcola la media globale delle stelle come `sommaStelle / totaleRecensioni` e la stampa con due cifre decimali.

Complessità

- **Temporale**:
  - Il ciclo scorre tutti i ristoranti una volta: O(m), con *m* numero di ristoranti.
  - Le chiamate a `getNumeroRecensioni()` e `getMediaStelle()` sono considerate O(1).
- **Spaziale**:
  - O(1), utilizza solo variabili scalari per il conteggio e la somma.

`visualizzaRecensioni`

Il metodo `visualizzaRecensioni` permette al ristoratore di visualizzare i suoi ristoranti che hanno recensioni, e di selezionarne uno per approfondire le recensioni specifiche.

Funzionamento:

1. **Stampa titolo** per indicare la sezione recensioni.
2. **Recupera lista ristoranti** del ristoratore tramite `ristoratore.getRistoranti()`.
3. **Controllo lista ristoranti**:
   a. Se la lista è nulla o vuota, stampa messaggio e termina.
4. **Filtra ristoranti con recensioni**:
   a. Crea una lista `ristorantiConRecensioni`.
   b. Per ogni ristorante, controlla con `haRecensioni()`.
   c. Se true, aggiunge il ristorante a `ristorantiConRecensioni`.
5. **Controlla lista filtrata**:
   a. Se vuota, stampa messaggio e termina.
6. **Mostra elenco ristoranti con recensioni**:
   a. Elenca ogni ristorante con indice, nome, numero recensioni e media stelle.
7. **Mostra opzione "Torna al menu principale"** (opzione 0).
8. **Legge input dell'utente** tramite `leggiIntero()`.
9. **Valida scelta**:
   a. Se 0, esce.
   b. Se scelta non valida (fuori range), stampa messaggio e termina.
10. **Visualizza recensioni del ristorante selezionato** tramite `visualizzaRecensioniRistorante()`.

Complessità

- **Temporale**:
  - O(m) per filtrare i ristoranti, con *m* numero di ristoranti del ristoratore.
  - L'accesso e stampa è lineare nella lista filtrata (al massimo O(m)).
- **Spaziale**:
  - O(m) per la lista temporanea `ristorantiConRecensioni`.

`rispostaRecensioni`

Il metodo `rispostaRecensioni` permette al ristoratore di selezionare una recensione senza risposta di un ristorante e aggiungere una risposta testuale.

Funzionamento:

1. **Recupera recensioni senza risposta** del ristorante tramite `ristorante.getRecensioniSenzaRisposta()`.
2. **Controlla se la lista è vuota**:
   a. Se sì, stampa messaggio e termina.
3. **Stampa elenco recensioni senza risposta** con indice numerico e una versione formattata (`formatRecensione`).
4. **Ciclo di input dell'utente** per scegliere quale recensione rispondere:
   a. Chiede all'utente di inserire un numero o 0 per annullare.
   b. Se 0, termina.
   c. Se scelta non valida, stampa messaggio e ripete richiesta.
5. **Seleziona recensione scelta** (indice `scelta - 1`).
6. **Mostra recensione selezionata** in modo dettagliato.
7. **Richiede inserimento testo risposta**:
   a. Se l'utente inserisce "STOP" (case-insensitive), stampa messaggio di annullamento e termina.
8. **Prova a inviare la risposta** usando `RecensioneService.rispondiARecensione` con ristoratore, ristorante, recensione e testo risposta.
9. **Verifica successo**:
   a. Se true, stampa messaggio di successo.
   b. Altrimenti, stampa messaggio di errore.
   c. In caso di eccezione, stampa messaggio di errore.

Complessità

- **Temporale**:
  - O(n) per stampare le recensioni senza risposta, con *n* numero di recensioni senza risposta.
  - L'interazione è dominata dalla lettura input, e chiamate ai servizi esterni (dipende da loro).
- **Spaziale**:
  - O(n) per la lista `recensioniSenzaRisposta`.