Università degli Studi di Salerno

Corso di Ingegneria del Software

RistOrganizer
RAD(Requirements Analysis Document)
Versione 1.3

Data: 11/11/2024

Progetto: Nome Progetto	Versione: X.Y
Documento: Titolo Documento	Data: GG/MM/AAAA

Coordinatore del progetto:

Nome	**Matricola**

Partecipanti:

Nome	**Matricola**
Stefano Palazzo	0512117736
Francesco Casillo	0512118276
Giulia Troiano	0512118888
Vincenzo Ferrara	0512117970
Scritto da:	Vincenzo Ferrara, Giulia Troiano, Francesco Casillo, Stefano Palazzo

Revision History

Data	**Versione**	**Descrizione**	**Autore**
14/12/2024	1.0	Prima Versione	Stefano Palazzo, Francesco Casillo, Vincenzo Ferrara, Giulia Troiano

Indice

1. Introduzione

Questo documento di Object Design descrive l'architettura logica e fisica del sistema **RistOrganizer**, delineando la struttura delle principali classi, interfacce e componenti che

costituiscono l'applicazione. L'obiettivo è fornire una guida chiara per l'implementazione, garantendo un equilibrio tra modularità, scalabilità e facilità di manutenzione.

Il sistema è stato progettato seguendo un approccio orientato agli oggetti, che favorisce la separazione delle responsabilità tra i vari componenti e promuove il riutilizzo del codice. Il documento illustra le principali decisioni progettuali e i compromessi valutati, spiegando nel dettaglio le scelte tecnologiche adottate per soddisfare i requisiti funzionali e non funzionali. Vengono inoltre definiti i vincoli e le linee guida per assicurare coerenza nell'implementazione e conformità agli standard di settore.

1.1 Compromessi nella progettazione orientata agli oggetti

La progettazione degli oggetti per il sistema **RistOrganizer** richiede una valutazione approfondita dei compromessi tra priorità progettuali quali:

- **Scalabilità**
- **Affidabilità**
- **Usabilità**
- **Prestazioni**
- **Compatibilità**

Ogni decisione progettuale è stata guidata dai requisiti funzionali e non funzionali, con l'obiettivo di bilanciare le esigenze operative con i vincoli economici e tecnici. Questa sezione analizza i principali compromessi affrontati durante la progettazione, evidenziando come le scelte adottate abbiano influenzato l'architettura del sistema, garantendo un equilibrio ottimale tra flessibilità, efficienza e sostenibilità nel lungo termine.

1.2 Affidabilità vs Sostenibilità economica

L'affidabilità è una priorità essenziale per il sistema **RistOrganizer**, che deve garantire continuità operativa anche durante i picchi di attività, ad esempio con 300 ordini simultanei nelle ore di punta.

Per soddisfare questo requisito, l'architettura è stata progettata utilizzando:

- **Infrastrutture scalabili**
- **Bilanciamento del carico**
- **Strumenti di monitoraggio in tempo reale**

Tuttavia, queste scelte comportano costi operativi più elevati, sia per l'infrastruttura che per la manutenzione. Per contenere i costi, il sistema è stato strutturato per consentire configurazioni iniziali meno dispendiose, come l'uso di risorse condivise o soluzioni di hosting economiche.

Questo approccio può temporaneamente ridurre l'affidabilità, ma la struttura supporta una scalabilità graduale verso configurazioni più robuste. Questo compromesso permette di bilanciare l'affidabilità con la sostenibilità economica, adattandosi alle necessità operative del ristorante con l'evolversi della piattaforma.

1.3 Scalabilità vs Sviluppo rapido

La scalabilità è stata raggiunta attraverso:

- **Un'architettura flessibile e modulare**
- **Un sistema di gestione delle sessioni**

Questo approccio supporta l'espansione del sistema per aggiungere nuove funzionalità, come:

- Monitoraggio in tempo reale di più punti vendita
- Integrazione con applicazioni di consegna

Tuttavia, l'implementazione di soluzioni avanzate per la gestione della sicurezza e dell'autenticazione introduce una maggiore complessità, richiedendo una pianificazione accurata.

Per bilanciare scalabilità e rapidità di sviluppo, il sistema è stato realizzato in modo iterativo, affrontando gradualmente le complessità del design senza compromettere i tempi di rilascio. Questo compromesso consente al sistema di prepararsi a una crescita futura, garantendo una base solida per le espansioni.

1.4 Affidabilità vs Prestazioni

L'affidabilità è un requisito fondamentale per il sistema **RistOrganizer**, che deve assicurare operatività continua anche in condizioni di carico elevato, come durante eventi speciali o promozioni.

Le soluzioni implementate includono:

- **Bilanciamento del carico**
- **Monitoraggio continuo**
- **Meccanismi di failover**

Questi strumenti garantiscono resilienza e tempi di ripristino rapidi in caso di problemi. Tuttavia, introducono un overhead che può influire sulle prestazioni complessive.

Per bilanciare affidabilità e prestazioni, sono stati implementati:

- **Meccanismi di caching**
- **Strategie per ottimizzare l'accesso ai dati**

Queste soluzioni migliorano i tempi di risposta, consentendo di mantenere un alto livello di affidabilità senza compromettere significativamente le prestazioni, rispondendo efficacemente alle esigenze operative del ristorante.

1.5 Riutilizzabilità vs Semplicità

La riutilizzabilità dei componenti è uno dei principi chiave del sistema **RistOrganizer**, pensato per:

- **Semplificare la manutenzione**

- **Ridurre la duplicazione del codice**

La progettazione di componenti riutilizzabili, come:

- Moduli per la gestione dei menu
- Configurazione delle sale

Accelera lo sviluppo e assicura coerenza tra le diverse funzionalità. Tuttavia, questa scelta incrementa la complessità del design, poiché i componenti devono essere sufficientemente flessibili per soddisfare esigenze operative differenti.

Per bilanciare riutilizzabilità e semplicità, è stato adottato un **approccio modulare**, che consente di:

- Riutilizzare i componenti chiave
- Mantenere leggibilità e comprensione del codice

Questo approccio facilita una crescita sostenibile del sistema, mantenendo la complessità su un livello gestibile.

2. Linee guida per il front-end

2.1 Organizzazione dei file:

Mantieni una struttura ben organizzata per i file del front-end. Ad esempio:

- CSS/SCSS: per gli stili.
- JS: per l'interattività lato client, laddove necessario.
- Template HTML/PHP: per la generazione dinamica del contenuto.

2.2 Comunicazione con il Back-End:

Per funzionalità interattive come il caricamento dinamico dei dati, utilizza richieste AJAX tramite JavaScript che invocano endpoint PHP.

2.3 Componentizzazione:

Adotta un approccio modulare, creando componenti riutilizzabili. Ad esempio, header, footer e card possono essere definiti come file separati e inclusi dove necessario (include o @include nei motori di template).

2.4 Validazione e Sicurezza:

Assicurati che i dati mostrati siano sempre validati e sanitizzati per evitare attacchi XSS. Usa funzioni come htmlspecialchars per i dati dinamici.

2.5 Progressive Enhancement:

Progetta il front-end con un approccio "server-side first", rendendo la tua applicazione pienamente funzionale anche senza JavaScript, e aggiungi funzionalità più avanzate con JavaScript solo dove necessario.

2.6 Linee guida per il back-end

Mantieni una chiara separazione tra la logica applicativa, la gestione dei dati e la presentazione. Adotta un approccio MVC (Model-View-Controller) per separare responsabilità e migliorare la manutenibilità.

2.7 Definizione delle Interfacce:

Utilizza interfacce per definire contratti chiari tra i componenti.

Implementa principi SOLID per garantire che le classi abbiano responsabilità ben definite e siano facilmente estensibili.

2.9 Gestione delle Dipendenze:

Usa un gestore di pacchetti come Composer per includere librerie di terze parti. Evita di implementare soluzioni da zero se esistono librerie affidabili e consolidate.

2.10 Gestione della Configurazione:

Centralizza la configurazione in file dedicati, come un file PHP o un formato standard come .env. Evita di inserire dati sensibili nel codice sorgente.

2.11 Sicurezza:

Sanitizza sempre i dati in input e utilizza meccanismi di protezione contro SQL Injection e XSS. Implementa meccanismi di autenticazione e autorizzazione sicuri, come sessioni o token.

2.12 Logging ed Error Handling:

Implementa un sistema di log per registrare eventi e gestire errori in modo strutturato.

Crea una gestione centralizzata degli errori per garantire che vengano gestiti correttamente senza esporre informazioni sensibili.

3. Modello concettuale

- **Oggetti principali:**

- **Prenotazione:** Rappresenta una prenotazione di un cliente. Contiene informazioni come data, orario, numero di persone e stato.
- **Menu:** Gestisce piatti e bevande, ognuno con prezzo e descrizione.
- **Tavolo:** Rappresenta i tavoli del ristorante, con capacità e stato (libero, occupato).
- **Personale:** Gestisce i turni del personale, i ruoli e i dati di contatto.

- **Relazioni principali:**
  - **Prenotazione ↔ Tavolo:** Una prenotazione è associata a un tavolo.
  - **Menu ↔ Prenotazione:** Ogni prenotazione può includere piatti del menu.
  - **Turno ↔ Personale:** Ogni turno è associato a un membro del personale.

4. Dettagli di implementazione

4.1 Attributi e metodi delle classi

Esempio per la classe **Prenotazione**:

Classe	**Attributi**	**Metodi**
Prenotazione	- cognome: string - identificativo: int -1	+ getCognome(): string + getIdentificativo()(): int + getnTavolo(): int + getNumeroPosti(): int + setCognome(String): void + setNumeroPosti(): void
Tavolo	- Tavolo: int - numeroPosti: int	+ getnTavolo(): int + getNumeroPosti(): int + setNumeroPosti(): void
servitaDa	- cfPersonale: string - giorno: date - idTavolo: int	+ getCfPersonale(): string + getGiorno(): date + getIdTavolo(): int + setCfPersonale(string): void + setGiorno(date): void + setIdTavolo(int): void
Personale	- codiceFiscale: string	+ getCodiceFiscale(): string

	- cognome: string - nome: string - oreLavoro: int - tipoPersonale: string	+ getCognome(): string + getNome(): string + getOreLavoro(): int + getTipoPersonale(): string + setCodiceFiscale(string): void + setCognome(string): void + setNome(string): void + setOreLavoro(int): void + setTipoPersonale(string): void
Ordinazione	- dataPrenotazione: time - identificativo: int - idTavolo: int - prezzo: float	+ getDataPrenotazione(): date + getIdentificativo(): int + getIdTavolo(): int + setDataPrenotazione(date): void + setIdentificativo(int): void + setIdTavolo(int): void + setIdTavolo(int): void
preparataDa	- cfPersonale: string - idOrdinazione: int - idTavolo: int	+ getCfPersonale(): string + getIdOrdinazione(): int + getIdTavolo(): int + setCfPersonale(string): void + setIdOrdinazione(int): void + setIdTavolo(int): void

4.2 Design Pattern

MVC (Model-View-Controller)

Il pattern MVC è stato utilizzato per separare le responsabilità principali del sistema:

- **Model:** Gestisce i dati e la logica di business, includendo classi come Prenotazione, Menu, Tavolo e Personale.
- **View:** Responsabile della presentazione dei dati all'utente. Comprende le interfacce utente come pagine JSP e moduli HTML.
- **Controller:** Coordina le richieste dell'utente e invoca le operazioni appropriate nel Model. Esempi includono le servlet PrenotazioneController e MenuController.

Questa separazione migliora la manutenibilità e facilita l'integrazione di nuove funzionalità.

Singleton

Il pattern Singleton è stato adottato per la classe DatabaseConnection, che gestisce l'accesso al database.

- **Obiettivo:** Garantire che esista una sola istanza della connessione al database durante l'esecuzione dell'applicazione.
- **Implementazione:** La classe utilizza un costruttore privato e un metodo statico getInstance() per fornire l'accesso all'istanza unica.

5. Elenco Classi e Interfacce di tipo Service e Controller

5.1 Classi

Nome	Descrizione
AdminClass	AdminClass è una **classe concreta** che eredita da UtenteClass e implementa funzionalità avanzate di gestione. L'amministratore può modificare e gestire diverse entità del sistema tramite una serie di metodi dedicati, sfruttando oggetti di interfaccia (ISpTavolo, ISpPrenotazione, ISpPortata, ecc.). Questo approccio suggerisce l'utilizzo di **pattern di progettazione** come il **Facade** per delegare le operazioni a classi di supporto.
ComandaClass	La classe ComandaClass permette di **creare, gestire e validare** una comanda per un ristorante. Utilizza controlli robusti per garantire l'integrità dei dati, lancia eccezioni in caso di input non validi e fornisce metodi per aggiungere/rimuovere portate alla comanda.
CucinaClass	CucinaClass rappresenta un utente con privilegi dedicati alla gestione delle comande per la cucina. La sua funzione principale è **modificare lo stato delle portate** nelle comande, tramite un'operazione delegata alla classe Facade (getUtenteFacade()).

DBMS_SP	**DBMS_SP** implementa diverse interfacce per la gestione di un sistema di ristorazione, come utenti, tavoli, comande, portate e prenotazioni.
PortataClass	Questa classe fornisce un modo strutturato per definire e validare oggetti con attributi descrittivi e di prezzo, garantendo che i valori rispettino le condizioni definite (es. il prezzo non può essere nullo o negativo).
prenotazioneClass	La classe **prenotazioneClass** implementa l'interfaccia **PrenotazioneInterface** e rappresenta una prenotazione con tre attributi principali: 1. **idUtente** (identificativo dell'utente), 2. **data** (data della prenotazione), 3. **idTavolo** (identificativo del tavolo prenotato).
SalaClass	**SalaClass** fornisce un insieme di funzionalità specializzate per la gestione delle comande in un contesto di ristorazione (sala). Agisce come un gestore della sala che, attraverso metodi ben definiti, può creare, modificare, interrogare e gestire comande e portate.
tavoloClass	La classe **tavoloClass** gestisce i dati relativi ai tavoli, come il numero del tavolo e il numero di posti disponibili. Garantisce l'integrità dei dati attraverso l'uso di **eccezioni** per impedire valori non validi, come numeri negativi o nulli.
UtenteClass	La classe **UtenteClass** centralizza la gestione degli utenti, garantendo l'integrità dei dati grazie a controlli rigorosi (es. validazione email, gestione tipologie). Inoltre, delega funzionalità avanzate come la **creazione di prenotazioni** alla classe **UtenteFacade**, seguendo il **pattern Facade** per separare le logiche complesse dal modello utente.

5.2 INTERFACCE

Nome	**Descrizione**

ComandaInterface	Questa interfaccia è progettata per gestire le operazioni di una comanda in un sistema gestionale (ad esempio, un ristorante). Fornisce metodi per: • Impostare e recuperare informazioni di base come prezzo totale, stato, cameriere e tavolo. • Gestire dinamicamente la lista delle portate (aggiunta/rimozione).
UtenteInterface	L'interfaccia UtenteInterface fornisce un modello standard per rappresentare un utente in un'applicazione, con un focus sulle proprietà fondamentali (nome, email, password, ecc.) e su eventuali estensioni (es. gestione tramite facade). Questo approccio migliora la modularità e la manutenzione del codice.
ComandaPersonaleSala	Il codice rappresenta un'interfaccia PHP denominata ComandaPersonaleSala, utilizzata per definire un insieme di operazioni che un'implementazione concreta deve supportare per gestire una comanda (ordine) in un sistema di gestione per un ristorante o una sala. L'interfaccia specifica metodi per manipolare e interrogare le comande.
ComandaResponsabileCucina	Questa interfaccia è progettata per essere implementata da classi che devono fornire una logica per aggiornare lo stato delle portate in un ordine (comanda).
ISpComanda	L'interfaccia fornisce un insieme standardizzato di metodi per gestire tutte le operazioni relative alle comande, inclusi l'aggiunta, la modifica, l'eliminazione e le interrogazioni sulle informazioni. È progettata per essere implementata da una classe che gestisce la logica operativa di un sistema di gestione delle comande in un ristorante.
ISpPortata	L'interfaccia stabilisce un contratto per la gestione di oggetti "portata", tipico in un'applicazione per la gestione di menu o ordini di un ristorante. Le implementazioni concrete dell'interfaccia devono fornire la logica effettiva per ciascun metodo.
ISpPrenotazione	Il codice definisce un'interfaccia ISpPrenotazione in PHP, che stabilisce un contratto per le classi che la implementeranno. Le

	funzioni dichiarate nell'interfaccia sono metodi statici e descrivono operazioni relative alla gestione delle prenotazioni, come aggiungere, eliminare, modificare o ottenere informazioni sulle prenotazioni.
ISpTavolo	Questa interfaccia stabilisce una serie di operazioni che una classe deve implementare per gestire i tavoli in un sistema.
ISpUtente	Questo codice definisce un'interfaccia ISpUtente in PHP, che stabilisce un insieme di metodi statici che devono essere implementati da qualsiasi classe che implementa questa interfaccia. I metodi gestiscono le operazioni relative agli utenti in un sistema di gestione utenti.
IUtentePrenotazione	Questa interfaccia impone che qualsiasi classe la implementi debba fornire una propria implementazione del metodo creazionePrenotazione, che gestisce la creazione di una prenotazione per un tavolo, utilizzando i dati forniti.
PortataAmministratore	Questa interfaccia stabilisce i metodi necessari per la gestione delle "portate" come la creazione, modifica e rimozione delle stesse.
PortataInterface	Questo codice definisce un'interfaccia in PHP chiamata PortataInterface che stabilisce un insieme di metodi, che una classe che implementa questa interfaccia deve definire.
PortataResponsabileCucina	Questo codice PHP definisce un'interfaccia chiamata PortataResponsabileCucina, che stabilisce un contratto per le classi che la implementeranno.
PrenotazioneAmministratore	Questa interfaccia definisce le operazioni di modifica e rimozione delle prenotazioni che un amministratore può eseguire su un sistema di prenotazione.
PrenotazioneInterface	Ogni classe che implementa questa interfaccia dovrà definire metodi per garantire che tutte le prenotazioni abbiano una data, un ID utente e un ID tavolo associati.

TavoloAmministratore	Questa interfaccia è destinata a gestire operazioni di aggiunta, eliminazione e modifica delle caratteristiche di tavoli in un sistema. Le classi che implementano questa interfaccia dovranno definire concretamente queste operazioni.
TavoloInterface	Questa interfaccia definisce i metodi che una classe che rappresenta un "tavolo" dovrebbe implementare per gestire il numero del tavolo e il numero di posti.