ODD – Object Design Document

Versione	1.1
Data	06/12/2021
Destinatario	Prof.ssa F. Ferrucci, Prof. Fabio Palomba
Presentato da	Tutti i team member
Approvato da	Salvatore Amideo Alice Vidoni

Revision History

Data	Versione	Descrizione	Autori
06/12/2021	0.1	Prima stesura	Alessandro Cavaliere, Nicola Cappello, Carmine Citro, Claudio Buono, Maria Rosaria Giudice
06/12/2021	1.0	Sviluppo introduzione	Alessandro Cavaliere, Nicola Cappello, Carmine Citro, Claudio Buono, Maria Rosaria Giudice
09/12/2021	1.0	Aggiunta Design Pattern e Class Interface	Alessandro Cavaliere, Nicola Cappello, Carmine Citro, Claudio Buono, Maria Rosaria Giudice
10/12/2021	1.1	Correzione formattazione del documento	Nicola Cappello, Alessandro Cavaliere

Sommario

1. Introduzione

1.1 Object Design trade-off

Durante la fase dell'Object Design, sono stati attentamente individuati e analizzati i seguenti trade-off:

- **Response Time vs Memory:** Nel caso in cui il software non rispetti i requisiti di tempo di risposta è necessario utilizzare più memoria in maniera tale da velocizzare il sistema.
- **Performance vs Maintenance:** La manutenibilità viene generalmente preferita alla performance in modo da facilitare gli sviluppatori nei processi di aggiornamento del software.
- **Availability vs Fault Tollerance:** La disponibilità del sistema all'utente (anche in caso di errore di una funzionalità) è un trade-off molto importante. Alcune volte la Fault Tollerance è necessaria per salvaguardare il continuo funzionamento e la continua disponibilità del sistema.
- **Developement Cost vs Security**: Per avere un sistema sicuro, ci vogliono costi maggiori; quindi, il sistema non permette di sviluppare un prodotto software che sia simultaneamente sicuro e costi poco.
- **Memory vs Extensibility:** Il sistema deve permettere l'estensibilità a discapito della memoria utilizzata, in modo tale da permettere al cliente di richiedere agli sviluppatori nuove funzionalità.

1.2 Interface Documentation Guidelines

Gli sviluppatori dovranno seguire precise linee guida per la stesura del codice. Per la scrittura del codice JavaScript ci si atterrà allo standard Google JavaScript.

1.2.1 Package

I nomi dei package sono tutti in "lowerCamelCase". Per esempio, my.gestioneOrdiniPasto.package , ma non gestioneordinipasto.package o my.gestione_ordini_pasto.package.

1.2.2 Classi e interfacce JavaScript

Interfacce, classi, record e name typedef sono scritti in "upperCamelCase". Le classi non esportate sono locali: non sono contrassegnate con @private e quindi non sono indicate con un carattere di sottolineatura finale. I nomi dei tipi sono tipicamente nomi o nomi di frasi. Ad esempio, Request, ImmutableList, o VisibilityMode. Inoltre, i nomi delle interfacce possono essere aggettivi o frasi di aggettivo (esempio, Readable).

1.2.3 Metodi

I nomi dei metodi sono scritti in "lowerCamelCase".

I nomi dei metodi sono tipicamente verbi o frasi di verbi. Per esempio, invioMessaggio. I metodi get e set, per le proprietà non sono mai richieste, ma se venissero utilizzate dovrebbero essere denominate, per esempio, getSaldo (e isSaldo per i boolean) o setSaldo(value) per i set.

1.2.4 Nomi costanti

I nomi costanti usano COSTANT_CASE: tutte le lettere maiuscole, con parole separate dalle sottolineature. Ogni costante è una proprietà statica @const o una dichiarazione const locale nel modulo. Prima di scegliere il caso costante, bisogna considerare se il campo costante sembra davvero una costante immutabile e quindi utile per il proprio scopo.

1.2.5 Nomi non costanti

I nomi di campi non constanti (static o altro) sono scritti in "lowerCamelCase" con una sottolineatura finale per i campi privati. Questi nomi sono in genere nomi o frasi. (es. computedValues).

1.2.6 Nomi dei Parametri

I nomi dei parametri sono scritti in "lowerCamelCase"

I nomi dei parametri formati da un solo carattere devono essere evitati nei metodi pubblici. Eccezione: quando richiesto da un framework di terze parti, i nomi dei parametri possono iniziare con $. Questa eccezione non si applica ad altri identificatori (ad esempio variabili locali).

1.2.7 Nomi delle variabili Locali

I nomi delle variabili locali sono scritti in "lowerCamelCase", solo quando le variabili finali, immutabili o locali non sono considerati constanti.

1.2.8 Enum names

Gli Enum names sono scritti in "UpperCamelCase", simili alle classi, composti generalmente da un singolo nome.

1.2.9 Nomi dei parametri del modello

I nomi dei parametri del modello devono essere concisi, singola parola o singola lettera, e devono essere maiuscoli, come TYPE or THIS.

1.2.10 Nomi Locali del modulo

I nomi locali del modulo non vengono esportati, sono privati. Non sono contrassegnati con @private e non terminano con un trattino basso. Questo vale per classi, funzioni, variabili, costanti, enum e altri identificatori del modulo.

1.2.11 script JavaScript

Gli Script in JavaScript devono rispettare le seguenti convenzioni:

1. Gli script che svolgono funzioni distinte dal mero rendering della pagina dovrebbero essere collocati in file dedicati;
2. Il codice JavaScript deve seguire le stesse convenzioni per il layout;
3. I documenti JavaScript devono iniziare con un commento;
4. Le funzioni JavaScript devono essere documentate;
5. Gli oggetti JavaScript devono essere preceduti da un commento.

1.2.12 Pagine HTML

Le pagine HTML, sia in forma statica che dinamica, devono essere conformi allo standard HTML. Inoltre, il codice HTML statico deve utilizzare l'indentazione, per facilitare la lettura, secondo le seguenti regole:

1. Un'indentazione consiste in una tabulazione;
2. Ogni tag deve avere un'indentazione maggiore del tag che lo contiene;
3. Ogni tag di chiusura deve avere lo stesso livello di indentazione del corrispondente tag di apertura;
4. I tag di commento devono seguire le stesse regole che si applicano ai tag normali

1.2.13 Fogli di stile CSS

I fogli di stile (CSS) devono seguire le seguenti convenzioni:
Tutti gli stili non in-line devono essere collocati in fogli di stile separati.
Ogni foglio di stile deve essere iniziato da un commento analogo a quello presente nei file Java.
Ogni regola CSS deve essere formattata come segue:

1. I selettori della regola si trovano a livello 0 di indentazione, uno per riga;
2. L'ultimo selettore della regola è seguito da parentesi graffa aperta "{";
3. Le proprietà che costituiscono la regola sono listate una per riga e sono indentate rispetto ai selettori;

4. La regola è terminata da una parentesi graffa "}", collocata da sola su una riga;

1.2.14 Database MongoDB

I nomi delle collection devono seguire le seguenti regole:

1. Devono essere costituiti da sole lettere maiuscole;
2. Il nome deve essere un sostantivo singolare tratto dal dominio del problema ed esplicativo del contenuto.

I nomi dei campi devono seguire le seguenti regole:

1. Devono essere costituiti da sole lettere maiuscole;
2. Se il nome è costituito da più parole, è previsto l'uso di underscore (_);
3. Il nome deve essere un sostantivo singolare tratto dal dominio del problema ed esplicativo del contenuto.

1.3 Definizioni, Acronimi e Abbreviazioni

Acronimi**:**

- RAD: Requirements Analysis Document
- SDD: System Design Document
- ODD: Object Design Document

Definizioni:

- **upperCamelCase:** è una tecnica di Naming delle variabili, consiste nell'iniziare la parola con una lettera maiuscola e le rimanenti lettere minuscole.
- **lowerCamelCase:** è la pratica di scrivere parole composte o frasi unendo tutte le parole tra loro, consiste nello scrivere più parole insieme delimitando la fine e l'inizio di una nuova parola con una lettera maiuscola.
- **HTML:** Linguaggio di programmazione utilizzato per lo sviluppo di pagine Web
- **CSS:** acronimo di Cascading Style Sheets è un linguaggio usato per definire la formattazione delle pagine Web.
- **JavaScript:** JavaScript è un linguaggio di scripting orientato agli oggetti e agli eventi, comunemente utilizzato nella programmazione Web lato client per la creazione di effetti dinamici interattivi.
- **MongoDB:** è un DBMS non relazionale, orientato ai documenti.
- **MVC:** acronimo di Model-View-Controller, è un pattern architetturale molto diffuso nello sviluppo di sistemi software.

- **Bootstrap:** è una raccolta di strumenti liberi per la creazione di siti e applicazioni per il web.
- **React:** è un'applicazione per la creazione di UI interattive.

1.4 Riferimenti

- Documento UE_RAD_V_2.0.pdf
- Documento UE_SDD_V_1.2.pdf
- Documento UE_CP_V_0.1.pdf
- Libro: Object-Oriented Software Engineering Using UML, Patterns, and Java Third Edition Autor: Bernd Brugge & Allen H. Dutoit

2. Packages

2.1 View

Il package View è formato dalle componenti React (e relativi CSS) che forniscono l'interfaccia utente agli attori. Di seguito le varie views suddivise per funzioinalità:

- Footer.js - il footer dell'applicazione web
- Homepage.js - rappresenta la pagina iniziale dell'applicazione web
- NavbarApp.js - rappresenta la navbar per gli utenti non loggati
- NavbarAttore.js - navbar dinamica che cambia a seconda dell'utente loggato
- successPopUp.js - popUp dinamico che conferma il successo di un'operazione

gestioneChat

- chat.js - rappresenta la chat con cui Cliente e Personale Adisu si scambiano messaggi

gestionFAQ

- inserimentoFAQ.js - pagina che permette di inserire una FAQ
- modificaFAQ.js - pagina che permette di modificare una FAQ
- visualizzazioneFAQ.js - pagina che permette di visualizzare le FAQs

gestioneLogin

- login.js - pagina che permette di effettuare il login

- logout.js - pagina che contiene la logica per eliminare le variabili di sessione

gestioneMenu

- VisualizzazioneMenu.js - pagina che permette di visualizzare il menù

gestioneOrdinePasto

- dettagliOrdine.js - pagina che permette di visualizzare i dettagli di un singolo ordine
- listaOrdine.js - pagina che permette di visualizzare la lista di ordini di un utente
- pagamentoPasto.js - pagina che permette di effettuare il pagamento di un pasto
- sceltaPasti.js - pagina che permette di effettuare la scelta dei pasti per un'ordinazione

gestionePersonale

- inserimentoPersonale.js - pagina che permette di inserire un personale Adisu o un operatore mensa
- RimozionePersonale.js - pagina che permette di visualizzare le info di un personale e rimuoverlo
- VisualizzazioneListaPersonale.js - pagina che permette di visualizzare la lista del personale adisu o degli operatori mensa

gestioneProfilo

- modificaPassword.js - pagina che permette la modifica della password
- profilo.js - pagina che permette all'utente di visualizzare il proprio profilo e le sue informazioni

gestioneStatistiche

- statisticheSettimanali.js - pagina che permette agli operatori mensa di visualizzare le statistiche settimanali

gestioneTesserino

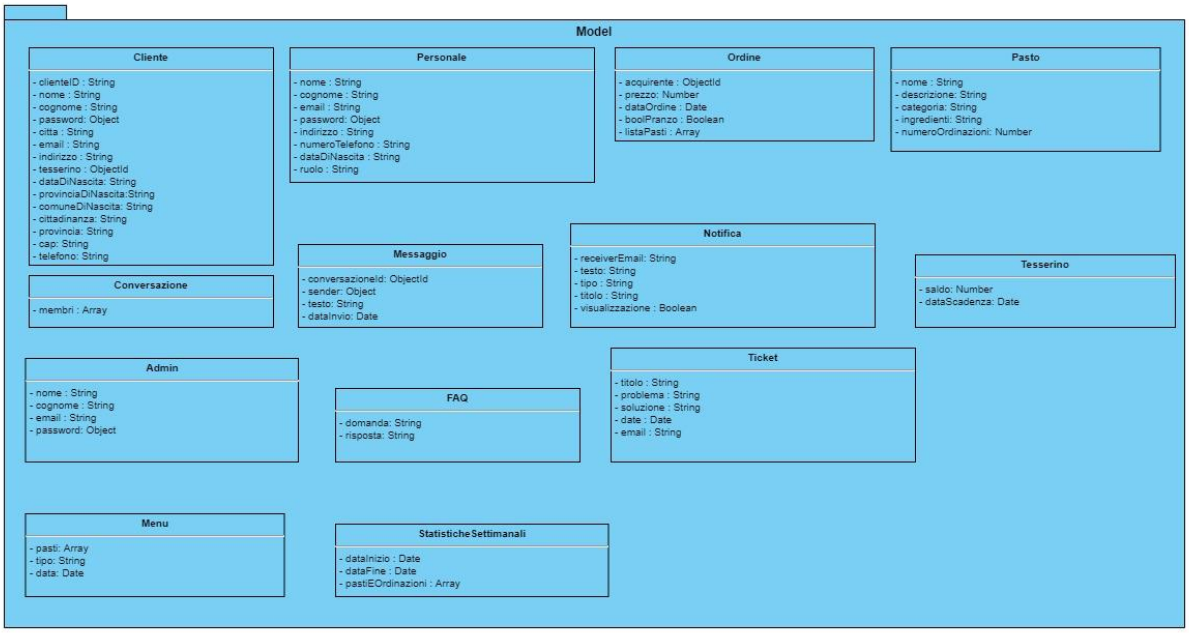
- ricaricaTesserino.js - pagina che permette al cliente di ricaricare il proprio tesserino
- richiestaTesserino.js - pagina che permette al cliente di richiedere un tesserino
- rinnovoTesserino.js - pagina che permette al cliente di rinnovare il tesserino
- visualizzaSaldo.js - pagina che permette al cliente di visualizzare il saldo del tesserino

gestioneTicket

- compilazioneTicket.js - pagina che permette la compilazione di un ticket
- risoluzioneTicket.js - pagina che permette all'Admin di risolvere un ticket
- visualizzazioneTicket.js - pagina che permette all'Admin di visualizzare la lista dei ticket non risolti

2.2 Model

Il package Model si occupa di fare da tramite tra l'applicazione e il database sottostante.
Ogni classe contenuta all'interno del pacchetto fornisce i metodi per accedere ai dati utili all'applicazione.
I moduli contenuti all'interno del package sono: Cliente, Admin, Personale, Conversazione, Messaggio, Notifica, Tesserino, Ordine, Menu, Pasto, FAQ, Ticket e Statistiche Settimanali.

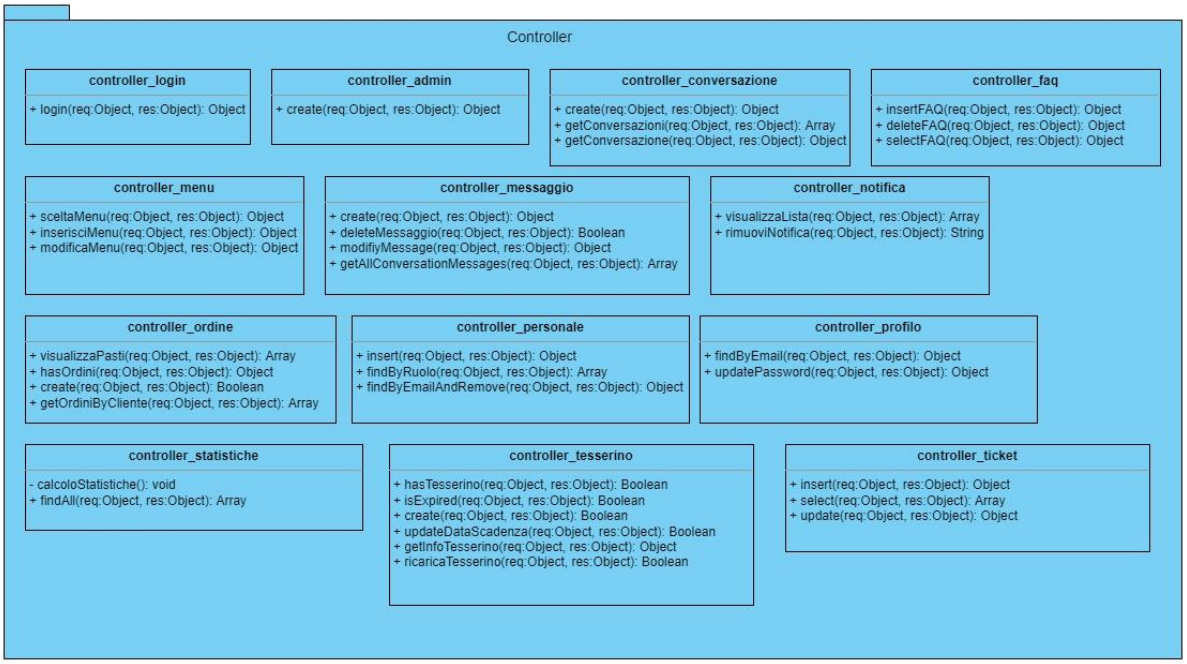


2.3 Controller

Il package Controller riceve, tramite il package View, i comandi dell'utente. Esso è formato dai moduli:

- controller_login - si occupa della gestione del login. Comunica con i model Cliente, Admin, Personale. Comunica con le view login.js.
- controller_tesserino - si occupa della gestione del tesserino e delle operazioni ad esso correlate. Comunica con i model Tesserino e con le view richiestaTesserino.js, rinnovoTesserino.js, ricaricaTesserino.js, visualizzaSaldo.js.
- controller_profilo - si occupa della gestione gestione del profilo, quindi di modificare la password e recuperare le informazioni del profilo da visualizzare. Comunica con i model Cliente, Admin e Personale. Comunica con le view profile.js e modificaPassword.js.

- controller_ordine - si occupa della gestione degli ordini e delle operazioni ad essi correlati, quindi il pagamento, la prenotazione, e la visualizzazione. Comunica con i model Ordine, Pasto. Comunica con le view listaOrdini.js, pagamentoPasto.js, sceltaPasti.js
- controller_menu - si occupa della gestione del menù, quindi dell'inserimento, della modifica e della visualizzazione. Comunica con i model Menu. Comunica con le view VisualizzazioneMenu.js
- controller_conversazione - si occupa della gestione delle conversazioni, quindi della loro creazione e visualizzazione. Comunica con i model Cliente e Personale. Comunica con la view chat.js
- controller_messaggio - si occupa della gestione dei messaggi, quindi della loro creazione, visualizzazione, modifica ed eliminazione. Comunica con i model Cliente e Personale. Comunica con la view chat.js
- controller_personale - si occupa della gestione del personale Adisu e degli operatori mensa, quindi del loro inserimento, della loro rimozione e visualizzazione. Comunica con i model Personale. Comunica con le view inserimentoPersonale.js, RimozionePersonale.js, VisualizzazioneListaPersonale.js
- controller_statistiche - si occupa della gestione delle statistiche settimanali, quindi della loro visualizzazione e della generazione automatica settimanale. Comunica con i model Statistiche, Pasto. Comunica con le view statisticheSettimanali.js
- controller_ticket - si occupa della gestione dei ticket, quindi del suo inserimento nel database dopo la compilazione, della risoluzione e del recupero di tutti i ticket non risolti. Comunica con il model Ticket e con le view compilazioneTicket.js, risoluzioneTicket.js, visualizzazioneTicket.js
- controller_faq - si occupa della gestione delle faq, quindi del loro inserimento, modifica e rimozione. Comunica con il model FAQ e con le view inserimentoFAQ.js, modificaFAQ.js, visualizzazioneFAQ.js
- controller_notifiche - si occupa della gestione delle notifiche, quindi della loro rimozione e visualizzazione. Comunica con il model Notifica. Comunica con le view notifiche.html e index.html(alla generazione di una notifica l'icona delle notifiche in index.html avrà un numero indicante le notifiche non lette).



3. Class Interfaces

Nome classe	loginControl
Descrizione	Questa classe gestisce le funzionalità di autenticazione e di uscita dal sistema.
Pre-condizione	**context** loginControl: login(String email, String password); **pre**: email != null && password != null && (Utente.email == email && Utente.password == password)
Post-condizione	
Invarianti	

Nome classe	tesserinoControl
Descrizione	Questa classe si occupa della gestione del tesserino e delle operazioni ad esso correlate.
Pre-condizione	**context** tesserinoControl: controlloRichiestaTesserino(Object richiesta); **pre**: richiesta != null

	context tesserinoControl: controlloMetodoPagamento(Number importo, String metodoPagamento); **pre**: importo > 0 && metodoPagamento != null **context** tesserinoControl: : ricaricaTesserino(String id, Number importo, String metodoPagamento); **pre**: id != null && importo > 0 && metodoPagamento != null && collection<Tesserino> -> includes(tesserino) **context** tesserinoControl: inserimentoTesserino(Object richiesta); **pre**: richiesta != null && collection<Tesserino> -> excludes(tesserino) **context** tesserinoControl: rinnovoTesserino(Object richiesta); **pre**: richiesta != null && collection<Tesserino> -> includes(tesserino) **context** tesserinoControl: ricaricaTesserino(String id, Number importo, String metodoPagamento); **pre**: id != null && importo > 0 && metodoPagamento != null && collection<Tesserino> -> includes(Tesserino)
Post-condizione	**context** tesserinoControl: : ricaricaTesserino(String id, Number importo, String metodoPagamento); **post**: Tesserino.saldo = Tesserino.saldo + importo
Invarianti	

Nome classe	**profiloControl**
Descrizione	Questa classe si occupa della gestione del profilo.
Pre-condizione	**context** profiloControl: getProfilo(String id, String tipoUtente); **pre:** id != null && tipoUtente != null

	context profiloControl: modificaPassword(String id, String tipoUtente, String passwordAttuale, String nuovaPassword); **pre**: tipoUtente != "Cliente" && id != null && passwordAttuale == Cliente.password && nuovaPassword != null
Post-condizione	**context** profiloControl: modificaPassword (String id, String tipoUtente, String passwordAttuale, String nuovaPassword); **post**: Cliente.password == nuovaPassword
Invarianti	

Nome classe	**ordinePastoControl**
Descrizione	Questa classe si occupa della gestione degli ordini e delle operazioni ad essi correlati.
Pre-condizione	**context** ordinePastoControl: inserisciPrenotazionePasto(String idCliente, Array IdPasti, Boolean boolPasto, Data dataOrdine); **pre**: idCliente != null && IdPasti != null && boolPasto != null && dataOrdine != null && collection<Ordine> -> excludes(ordine) **context** ordinePastoControl: pagamentoPasto(String idPasto, Number prezzo); **pre**: idPasto != null && collection<Ordine> -> includes(ordine) && Ordine.prezzo == prezzo && Tesserino.saldo >= prezzo **context** ordinePastoControl: getPrenotazioni(String idCliente); **pre**: idCliente != null **context** ordinePastoControl: getInfoOrdine(String idOrdine); **pre**: idOrdine != null **context** ordinePastoControl: generaQR(String idPasto); **pre:** idPasto != null && collection<Ordine> -> includes(ordine)
Post-condizione	**context** ordinePastoControl: inserisciPrenotazionePasto(String idCliente, Array idPasti, Boolean boolPasto, Data dataOdierna); **post**: collection<Ordine> -> includes(ordine)

	context ordinePastoControl: pagamentoPasto(String idPasto, Number prezzo); **post**: Tesserino.saldo = Tesserino.saldo – prezzo
Invarianti	

Nome classe	**menuControl**
Descrizione	Questa classe si occupa della gestione del menu.
Pre-condizione	**context** menuControl: getMenu(); **context** menuControl: inserimentoMenu(Object menu); **pre**: menu != null && collection<Menu> -> excludes(menu) **context** menuControl: modificaMenu(Object menu); **pre**: menu != null && collection<Menu> -> includes(menu)
Post-condizione	**context** menuControl: inserimentoMenu(Object menu); **post**: collection<Menu> -> includes(menu) **context** menuControl: modificaMenu(Object menu); **post**: collection<Menu> -> includes(menu)
Invarianti	

Nome classe	**chatControl**
Descrizione	Questa classe si occupa della gestione della chat.
Pre-condizione	**context** chatControl: invioMessaggio(String idMittente, String idDestinatario, String testo, Date dataInvio); **pre**: idMittente != null && idDestinatario != null && testo != null && dataInvio != null && collection<Messaggio> -> excludes(messaggio)

	context chatControl: getChat(String idCliente, String idPersonale); **pre**: idCliente != null && idPersonale != null && collection<Chat> -> includes(chat) * da decidere* **context** chatControl: getChats (String idUtente); **pre**: idUtente != null **context** chatControl: rimuoviMessaggio(String idMessaggio); **pre**: idMessaggio != null && collection<Messaggio> -> includes(messaggio) **context** chatControl: modificaMessaggio(String idMessaggio, String nuovoTesto); **pre**: idMessaggio != null && nuovoTesto != null && collection<Messaggio> -> includes(messaggio)
Post-condizione	**context** chatControl: invioMessaggio(String idMittente, String idDestinatario, String testo, Date dataInvio); **post**: collection<Messaggio> -> includes(messaggio) **context** chatControl: rimuoviMessaggio(String idMessaggio); **post**: collection<Messaggio> -> excludes(messaggio) **context** chatControl: modificaMessaggio(String idMessaggio, String nuovoTesto); **post**: collection<Messaggio> -> includes(messaggio)
Invarianti	

Nome classe	**personaleControl**
Descrizione	Questa classe si occupa della gestione del personale ADISU.
Pre-condizione	**context** personaleControl: inserisciPersonale(Object personale); **pre**: personale != null && collection<Personale> -> excludes(personale) **context** personaleControl: getInfo(String idPersonale);

	pre: idPersonale != null && collection<Personale> -> includes(personale) **context** personaleControl: rimuoviPersonale(String idPersonale); **pre**: idPersonale != null && collection<Personale> -> includes(personale) **context** personaleControl: getLista(String ruolo); **pre**: ruolo != null
Post-condizione	**context** personaleControl: inserisciPersonale(Object personale); **post**: collection<Personale> -> includes(personale) **context** personaleControl: rimuoviPersonale(String idPersonale); **post**: collection<Personale> -> excludes(personale)
Invarianti	

Nome classe	**statisticheControl**
Descrizione	Questa classe si occupa della gestione delle statistiche settimanali.
Pre-condizione	**context** statisticheControl: getStatistica(Date dataInizio, Date dataFine); **pre**: dataInizio != null && dataFine != null **context** statisticheControl: calcoloStatistiche(); **pre**: collection<StatisticheSettimanali> -> notEmpty() **context** statisticheControl: salvaStatistiche(Object statistiche); **pre**: statistiche != null && collection<StatisticheSettimanali> -> excludes(statistiche)
Post-condizione	**context** statisticheControl: salvaStatistiche(Object statistiche);

	post: collection<StatisticheSettimanali> -> includes(statistiche)
Invarianti	

Nome classe	ticketControl
Descrizione	Questa classe si occupa della gestione dei ticket.
Pre-condizione	**context** ticketControl: inserisciTicket(String titolo, String problema, Date data, String idMittente); **pre**: titolo != null && data != null && problema != null && idMittente != null && collection<Ticket> -> excludes(ticket) **context** ticketControl: getTicket(String id); **pre:** id != null **context** ticketControl: risolviTicket(String idTicket, String soluzione); **pre**: idTicket != null && soluzione != null
Post-condizione	**context** ticketControl: inserisciTicket(String titolo, String problema, Date data, String idMittente); **post**: collection<Ticket> -> includes(ticket)
Invarianti	

Nome classe	faqControl
Descrizione	Questa classe si occupa della gestione delle FAQ.
Pre-condizione	**context** faqControl: inserisciFAQ(Object faq); **pre**: faq != null && collection<FAQ> -> excludes(FAQ)

	context faqControl: modificaFAQ(String idFaq, Object FAQ); **pre**: idFaq != null && faq != null && collection<FAQ> -> includes(FAQ) **context** faqControl: rimuoviFAQ(String idFaq); **pre**: idFaq != null && collection<FAQ> -> includes(FAQ)
Post-condizione	**context** faqControl: inserisciFAQ(Object faq); **post**: collection<FAQ> -> includes(FAQ) **context** faqControl: modificaFAQ(String idFaq, Object faq); **post**: collection<FAQ> -> includes(FAQ) **context** faqControl: rimuoviFAQ(String idFaq); **post**: collection<FAQ> -> excludes(FAQ)
Invarianti	

Nome classe	**notificheControl**
Descrizione	Questa classe si occupa della gestione delle notifiche.
Pre-condizione	**context** notificheControl: generaNotifica(Object notifica); **pre**: notifica != null && collection<Notifica> -> excludes(notifica) **context** notificheControl: getNotifiche (String idUtente); **pre**: idUtente != null **context** notificheControl: rimuoviNotifica(String idNotifica); **pre**: idNotifica != null && collection<Notifica> -> includes(notifica)
Post-condizione	**context** notificheControl: generaNotifica(Object notifica); **post**: collection<Notifica> -> includes(notifica) **context** notificheControl: rimuoviNotifica(String idNotifica);

	post: idNotifica != null && collection<Notifica> -> excludes(notifica)
Invarianti	

4. Design Pattern Con Class Diagram

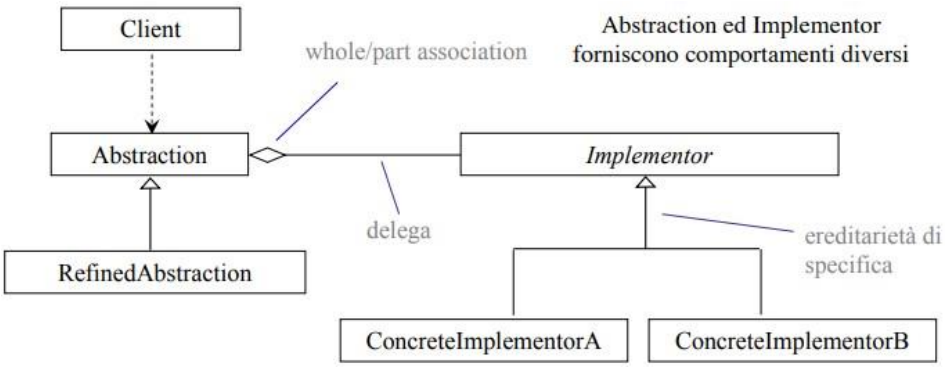
4.1 Bridge Pattern

Nome e classificazione: Bridge pattern.
Il Bridge pattern fa parte dei Structural Pattern.

Scopo: I Bridge Pattern si occupano di separare un'astrazione da un'implementazione così che, se l'implementazione è differente, può essere sostituita, anche a runtime.

Applicabilità: usato per interfacciare un insieme di oggetti sia quando l'insieme non è ancora completamente noto, in alcune fasi come analisi, design, ecc, e sia quando c'è bisogno di estendere un sottosistema dopo che il sistema è stato consegnato ed è già in uso (estensione dinamica).

Struttura:



Partecipanti: in generale il Bridge pattern impiega i seguenti componenti:

- Una classe Abstraction: definisce l'interfaccia visibile al codice Client;
- Implementor: un'interfaccia astratta che definisce i metodi di basso livello disponibili ad Abstraction;
- Un'istanza di Abstraction mantiene un riferimento alla corrispondente istanza di Implementor;
- Abstraction e Implementor possono essere raffinate indipendentemente.

Conseguenze:

- Disaccoppiamento tra interfaccia ed implementazione:
  - Un'implementazione non è più legata in modo permanente ad un 'interfaccia;
  - L ' implementazione di un ' astrazione può essere configurata durante l'esecuzione;
  - La parte di alto livello di un sistema dovrà conoscere soltanto le classi Abstraction e Implementor;
- Maggiore estendibilità:
  - Le gerarchie Abstraction e Implementor possono essere estese indipendentemente.
- Mascheramento dei dettagli dell'implementazione ai client
  - I client non devono preoccuparsi dei dettagli implementativi

Utilizzo: poiché l'implementazione è suddivisa tra sottoteam e ci sono classi che fanno uso di metodi di altre classi, utilizzeremo implementazioni stub per non attendere l'implementazione concreta.

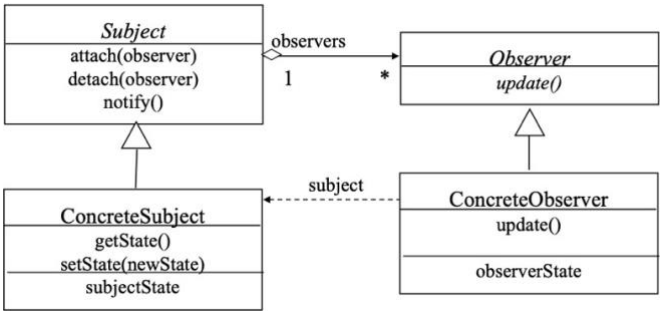
4.2 Observer Pattern

Nome e classificazione: Observer pattern.
L'observer pattern fa parte dei Behavioral Pattern.

Scopo: gli Observer Pattern si occupano di algoritmi e dell'assegnazione delle responsabilità tra oggetti che collaborano (Chi fa che cosa?). Caratterizzano flussi di controllo complessi, difficili da seguire a run time.

Applicabilità: definisce una dipendenza uno-a-molti tra gli oggetti in modo tale che quando un oggetto cambia stato, tutte le sue dipendenze vengano notificate e aggiornate automaticamente. Viene utilizzato per il mantenimento della coerenza in tutti gli stati ridondanti. Inoltre, viene utilizzato per l'ottimizzazione delle modifiche batch per mantenere la coerenza.

Struttura:



Partecipanti: in generale l'Observer pattern impiega le seguenti classi:

- Soggetto: una classe che fornisce interfacce per registrare o rimuovere gli observer.
- Soggetto Concreto: classe che contiene l'attributo "subjectState", il quale descrive lo stato del soggetto.
- Observer: classe che definisce un'interfaccia per tutti gli observer per ricevere le notifiche dal soggetto. È utilizzata come classe astratta per implementare i veri observer, ossia i ConcreteObserver.
- ConcreteObserver: questa classe mantiene un riferimento "subject" al Soggetto Concreto, per ricevere lo stato quando avviene una notifica. Inoltre, alla classe appartiene l'attributo "observerState", il quale contiene lo stato del ConcreteObserver.

Conseguenze: l'Observer pattern, garantisce che quando un oggetto cambia stato, un numero aperto di oggetti dipendenti vengano aggiornati automaticamente, oltre al fatto che deve essere possibile che un oggetto possa notificare un numero aperto di altri oggetti. L'Observer pattern può causare però perdite di memoria, noto come "problema dell'ascoltatore scaduto", perché nell'implementazione di base si richiede sia la registrazione esplicita, sia la cancellazione esplicita, come nel Dispose Pattern. Questo perché il soggetto contiene forti riferimenti agli observer, mantenendoli in vita.

Utilizzo: all'interno del nostro software verrà utilizzato per gestire la dipendenza tra gli ordini ed i pasti. Infatti, per ogni nuovo ordine, dovrà essere incrementato il numero di ordinazioni per i pasti ordinati. Ci sarà utile, inoltre, anche per la gestione del sistema di messaggistica.

5. Glossario

MongoDB: è un DBMS non relazionale, orientato ai documenti. Classificato come un database di tipo NoSQL, MongoDB si allontana dalla struttura tradizionale basata su tabelle dei database relazionali in favore di documenti in stile JSON con schema dinamico.

DBMS (Database Management System): è un sistema software progettato per consentire la creazione, la manipolazione e l'interrogazione efficiente di database, per questo detto anche "gestore o motore del database", è ospitato su architettura hardware dedicata oppure su semplice computer.

NoSQL: i database NoSQL sono appositamente realizzati per modelli di dati specifici e hanno schemi flessibili per creare applicazioni moderne. Si discostano dai Database di tipo relazionale, infatti l'espressione "NoSQL" fa riferimento al linguaggio SQL, che è il più comune linguaggio di interrogazione dei dati nei database relazionali, qui preso a simbolo dell'intero paradigma relazionale.

JSON (JavaScript Object Notation): è un semplice formato per lo scambio di dati. È basato sul linguaggio JavaScript, ma ne è indipendente. Viene usato in AJAX come alternativa a XML/XSLT.

Node.js: è un runtime system di JavaScript Open source multipiattaforma orientato agli eventi per l'esecuzione di codice JavaScript, costruita sul motore JavaScript V8 di Google Chrome. Node.js consente di utilizzare JavaScript anche per scrivere codice da eseguire lato server, ad esempio per la produzione del contenuto delle pagine web dinamiche prima che la pagina venga inviata al Browser dell'utente.

Modulo: sono file con estensione .js che contengono codice JavaScript, Node.js viene fornito con una serie di moduli principali che implementano funzionalità di base.

Bootstrap: è una raccolta di strumenti free per la creazione di siti e applicazioni per il Web. Essa contiene modelli di progettazione basati su HTML e CSS, sia per la tipografia, che per le varie componenti dell'interfaccia, come moduli, pulsanti e navigazione, così come alcune estensioni opzionali di JavaScript.

CamelCase: la notazione a cammello, o in inglese CamelCase, è la pratica nata durante gli anni 70 di scrivere parole composte o frasi unendo tutte le parole tra loro, ma lasciando le loro iniziali maiuscole. Si può distinguere in un lowerCamelCase, in cui la prima lettera della prima parola viene lasciata minuscola, o in UpperCamelCase, in cui la prima lettera della prima parola è maiuscola.

JavaScript: è un linguaggio di scripting orientato agli oggetti e agli eventi, comunemente utilizzato nella programmazione Web lato client (recentemente esteso anche al lato server) per la creazione, in siti web e applicazioni web, di effetti dinamici interattivi tramite funzioni di script invocate da eventi innescati a loro volta in vari modi dall'utente sulla pagina web in uso.

HTML (HyperText Markup Language): è un linguaggio di markup nato per la formattazione e impaginazione di documenti ipertestuali disponibili nel web 1.0, oggi è utilizzato principalmente per il disaccoppiamento della struttura logica di una pagina web.

CSS (Cascading Style Sheets): è un linguaggio usato per definire la formattazione di documenti HTML, XHTML e XML ad esempio nei siti web e relative pagine web.

Trade-off: il Trade-off è una situazione che implica una scelta tra due possibilità, in cui la perdita di valore di una costituisce un aumento di valore in un'altra.