Corso di Ingegneria del Software

Restaurant Management
Object Design
Versione 1.2

Data: 19/01/2019

Partecipanti al progetto

Nome	Matricola
Andrea Cipriano	512104874
Gianmarco Cringoli	512104778
Manuel Flora	512104628

Scritto da:	Andrea Cipriano, Gianmarco Cringoli, Manuel Flora

Revision History

Data	Versione	Descrizione	Autore
21/12/18	1.0	Design Pattern	Andrea Cipriano
21/12/18	1.0	Introduzione	Gianmarco Cringoli
21/12/18	1.0	Packages	Manuel Flora
22/12/18	1.1	Class Inteface	Andrea Cipriano
22/12/18	1.1	Class Inteface	Gianmarco Cringoli
22/12/18	1.1	Class Inteface	Manuel Flora
19/01/19	1.2	Revisione totale	Andrea Cipriano

Indice

1. Introduzione

Con l'obiettivo di ottimizzare la gestione delle attività ristorative di piccola-media dimensione, si vuole sviluppare un progetto che miri a semplificare le funzioni che il gestore di un ristorante e i camerieri svolgono quotidianamente. Il software, inoltre, è rivolto anche ai clienti dei ristoranti, i quali possono interagire con esso attraverso il web.

1.1. Object design trade-offs

1.1.1. Comprensibilità vs costi

Si preferisce aggiungere costi relativi alle ore/uomo dedicate per la documentazione, al fine di rendere il codice comprensibile sia alle persone non coinvolte nel progetto che alle persone coinvolte che non hanno lavorato a quella parte in particolare. Saranno introdotti commenti nel codice, per facilitarne la comprensione e la manutenzione.

1.1.2. Prestazioni vs Costi

Considerando che il budget a nostra disposizione non è elevato, il sistema sarà sviluppato utilizzando componenti free e open source, non saranno garantite alte prestazioni ma saranno comunque soddisfacenti per il normale utilizzo.

1.1.3. Costi vs Mantenimento

Il sistema può essere facilmente modificato ed implementato con nuove funzioni e corretto da errori, grazie all'uso di materiale open source e all'utilizzo di javadoc.

1.1.4. Interfaccia vs Easy-use

L'interfaccia, grazie ad una impostazione semplice e intuitiva, permette facile utilizzo (Easy-use) delle principali funzionalità del sistema anche per gli utenti meno esperti.

1.1.5. Memoria vs efficienza

Dato l'elevato carico di utenti che possono accedere al sistema, e dato che il sistema dovrà supportare un elevato numero di query, per non penalizzare le performance del sistema, verrà utilizzato un meccanismo di caching che permette la memorizzazione dei dati con accesso più frequente. Ciò sarà implementato attraverso l'utilizzo di Cookie.

1.1.6. Sicurezza vs costi

Dato il budget ridotto, non saranno utilizzate componenti esterne che garantiscano la massima sicurezza sui dati, ma verrà utilizzata una componente all'interno del linguaggio Java che permetta un grado di protezione soddisfacente.

1.1.7. Interfacce vs Tempo di risposta

Il tempo di risposta tra server e interfaccia è più che sufficiente a soddisfare le richieste da parte dell'utente in breve tempo.

1.2. Off-the-shelf component description

Per il progetto software che si vuole realizzare facciamo uso di componenti off-the-shelf, che sono componenti software disponibili sul mercato per facilitare la creazione del progetto. Per il sistema che si vuole realizzare faremo uso di un framework per la realizzazione dell'interfaccia grafica e di una libreria interna a Java per la crittografia dei dati.
Il framework scelto per l'interfaccia grafica è Bootstrap, un framework open source che contiene una raccolta di strumenti liberi per la creazione di siti e applicazioni per il Web. Questo contiene modelli di progettazione basati su HTML e CSS, per le varie componenti dell'interfaccia, come moduli, bottoni e navigazione, così come alcune estensioni opzionali di JavaScript.
Il tema Bootstrap scelto per la realizzazione e implementazione dell'interfaccia grafica, è stato reperito gratuitamente dal sito: https://shapebootstrap.net

1.3. Interface documentation guidelines

Per rendere il codice più estensibile e manutenibile, prima dell'implementazione della logica del sistema, è opportuno sottomettere le regole di implementazione, in modo che eventuali correzioni nella logica dell'applicazione possano essere apportate prima di imbattersi nella sintassi degli strumenti scelti.
Bootstrap: un toolkit open source utilizzato per lo sviluppo di progetti responsive sul web. Il suo utilizzo è previsto insieme a quello di HTML, CSS e JavaScript per realizzare la struttura base della grafica.
Commenti: i commenti di implementazione sono un mezzo per chiarire il codice o per esplicare una particolare implementazione. Sarà, inoltre, generato il javadocs con l'utilizzo della funzione di Eclipse (Generate Javadoc).
Metodi: i metodi devono essere preceduti da un commento, o più precisamente da una documentazione che riporti l'obiettivo che si vuole raggiungere. Inoltre bisogna giustificare le eventuali decisioni particolari o i calcoli commentandole.
Dichiarazioni: le dichiarazioni saranno posizionate all'inizio del blocco del codice. Non dichiarare le variabili al loro primo uso, può portare incomprensioni per il programmatore e rendere la manutenibilità del codice più complessa.
Indentazione: l'indentazione deve essere effettuata con un TAB e, a prescindere dal linguaggio usato per la produzione del codice, ogni istruzione deve essere opportunamente indentata.
Parentesi: a prescindere dalle istruzioni che seguono un IF o ciclo FOR e WHILE, è necessario riportare il blocco di istruzioni tra parentesi graffe.
Script Javascript: Gli script che svolgono funzioni distinte dal funzionamento di una pagina, dovrebbero essere collocati in file separato.
Le funzioni e oggetti Javascript devono essere preceduti da un commento in stile Javadoc.

1.4. Definitions, acronyms, and abbreviations

HTML: acronimo di Hyper Text Markup Language, è un linguaggio di markup utilizzato per la creazione di documenti destinati al web.
CSS: acronimo di Cascading Style Sheets, è un linguaggio usato per definire la formattazione di documenti HTML.
Framework: software di supporto allo sviluppo web.
Javascript: linguaggio di scripting orientato agli oggetti e agli eventi , comunemente utilizzato nella programmazione Web lato client che viene interpretato dal browser.
Eclipse: ambiente di sviluppo integrato multi-linguaggio e multi-piattaforma.
Javadoc: è un applicativo incluso nel Java Development Kit utilizzato per la generazione automatica della documentazione del codice sorgente scritto in linguaggio Java.
Off-The-Shelf: servizi esterni di cui viene fatto utilizzo da terzi.

1.5. References

- B.Brugge, A.H. Dutoit, Object Oriented Software Engineering- Using UML, Patterns and Java, Prentice Hall.
- https://www.bruegge.in.tum.de
- https://shapebootstrap.net
- SDD Restaurant_Management

2. Design pattern

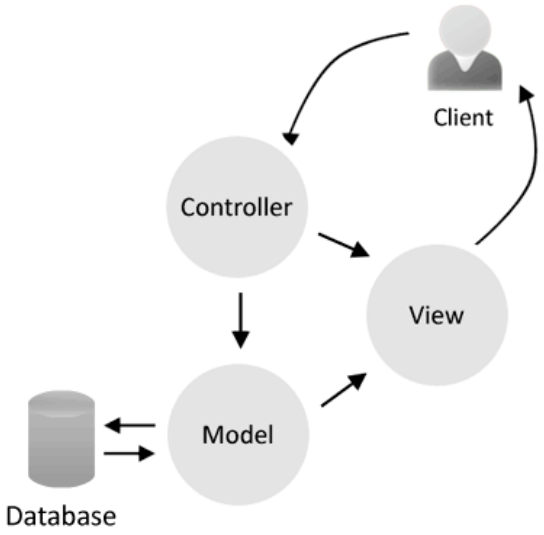
Di seguito, vengono descritti i design pattern ritenuti idonei all'implementazione del sistema proposto.

2.1. Model - View - Controller

Siccome in fase di system design si è stabilito che il sistema proposto presenta l'architettura Model - View - Controller (o MVC), in fase di implementazione è previsto l'utilizzo dell'omonimo pattern.
Il pattern MVC viene utilizzato in un contesto dove l'applicazione deve fornire un'interfaccia grafica costituita da più schermate che mostrano vari dati all'utente, i quali devono risultare aggiornati in qualunque momento. Tale pattern viene spesso utilizzato nei casi in cui l'applicazione presenti una natura modulare basata sulle responsabilità, al fine di ottenere un sistema basato sulle componenti.
La soluzione fornita da questo pattern risulta essere particolarmente adatta al sistema proposto, poiché prevede che l'applicazione debba separare i componenti software che implementano le funzionalità di business dai componenti che implementano la logica di presentazione e di controllo, i quali utilizzano tali funzionalità.
I componenti previsti dal pattern MVC sono descritti di seguito nel dettaglio:

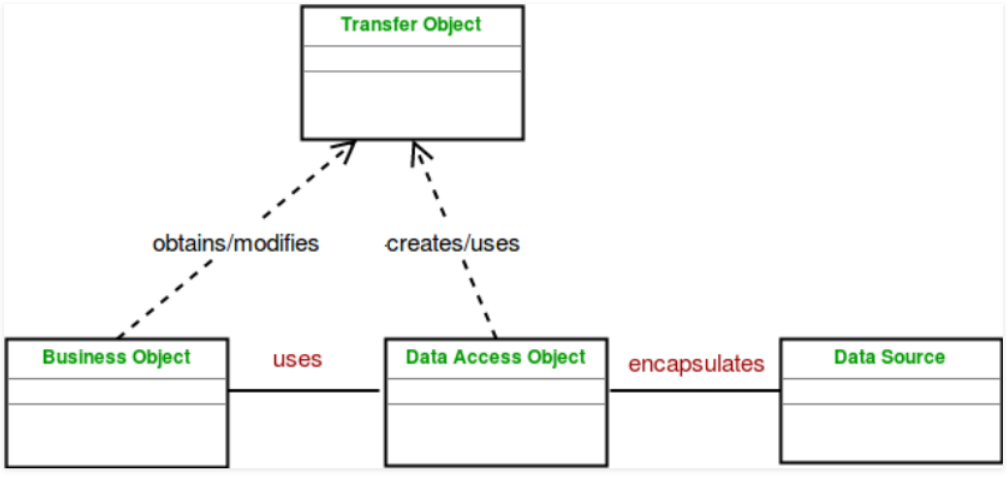
- Il **Model** definisce le regole di business per l'interazione con i dati, esponendo alla View ed al Control rispettivamente le funzionalità per l'accesso e l'aggiornamento dei dati.

- Il **Control** realizza la corrispondenza tra l'input dell'utente e i processi eseguiti dal Model, oltre a selezionare le schermate della View richieste ed implementare la logica di controllo dell'applicazione.
- La **View** si occupa della logica di presentazione dei dati. In genere, le View adottano una strategia "push Model", così come prevista dal design pattern dell'Observer. Tuttavia, siccome il presente modello MVC fa riferimento ad un'architettura J2EE, le view che vengono implementate con delle JSP restituiscono GUI costituite da contenuti statici in HTML, non in grado di eseguire delle operazioni sul Model. Per questo motivo, si è deciso di utilizzare la strategia "pull Model", che differisce in alcuni aspetti all'Observer. Questa strategia, infatti, prevede che la View richieda aggiornamenti quando lo ritiene opportuno e deleghi al Control sia l'esecuzione dei processi richiesti dall'utente (dopo averne catturato gli input), sia la scelta delle



eventuali schermate da presentare.

2.2. Data Access Object



Il pattern Data Access Object (o DAO) è utilizzato per separare l'API di accesso ai dati a basso livello dai servizi di business ad alto livello. In generale, un Pattern DAO segue lo schema seguente.

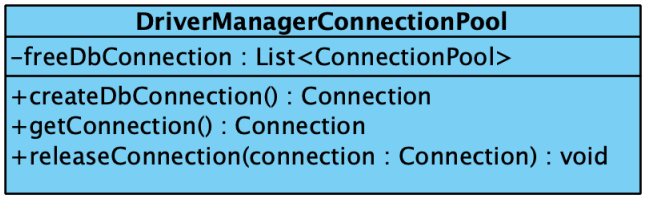
Le componenti previste dal pattern DAO sono descritte di seguito:

- il Business Object rappresenta il client che richiede l'accesso ai dati sorgente così da poterli archiviare. Un Business Object può essere implementato come un bean di sessione, un bean di entità o un altro oggetto Java, oltre a prevedere un servlet o un bean ausiliario che acceda all'origine dei dati.
- Il Data Access Object (DAO) astrae la funzionalità di accesso ai dati sottostanti al Business Object per consentire l'accesso trasparente alla sorgente dei dati. In questo caso, il Business Object delega il caricamento dei dati e memorizza le operazioni sui Data Access Object.
- Il Data Source rappresenta la sorgente dei dati, in questo caso un database relazionale.
- Il Transfer Object rappresenta un oggetto di trasferimento utilizzato come un vettore di dati. Il DAO potrebbe utilizzare un Trasfer Object per restituire i dati al client oppure riceve i dati dal client in un Trasfer Object per aggiornare i dati presenti nel database.

Nel sistema proposto, i Business Object sono rappresentati da Servlet che si servono di Bean e DAO per ricevere e inviare dati da e verso il client. In particolare, i DAO sono impegnati nell'invocazione dei metodi CRUD, mentre il DataSource, rappresentato dalla classe singleton DriverManagerConnectionPool (il cui pattern viene illustrato in seguito) ha il compito di gestire la connessione al database che viene poi utilizzata da tutti i DAO per la lettura, la scrittura, l'aggiornamento e l'eliminazione dei dati persistenti nel DBMS MySQL. Per finire, i Transfer Object consistono di tutti gli oggetti Bean che rappresentano le informazioni persistenti che vengono scambiate tra database e client.

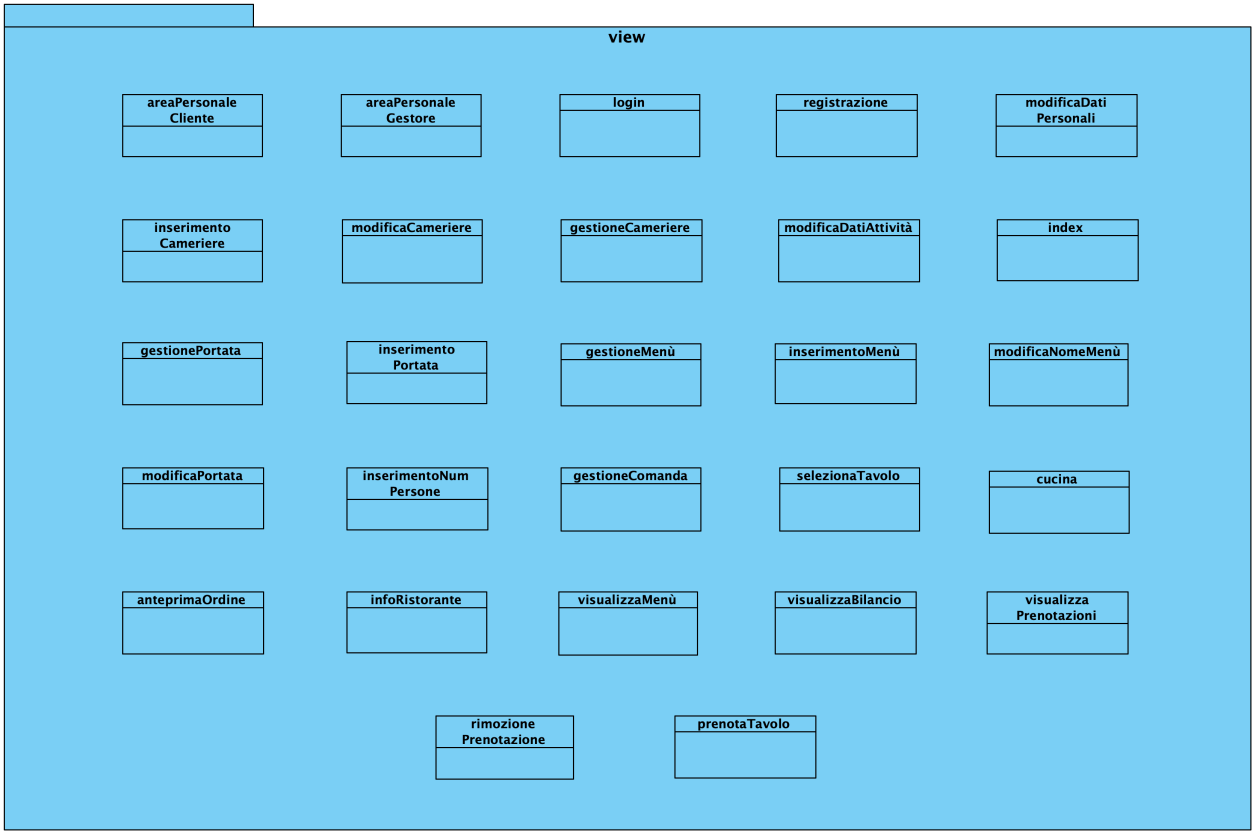
2.3. Singleton

Il pattern Singleton viene utilizzato quando si vuole garantire di avere un unico punto di accesso. Ad esempio, esso viene utilizzato quando si desidera avere un solo Window Manager oppure una sola Coda di Stampa oppure un unico accesso al database.
Un oggetto Singleton viene inizializzato nel momento in cui la classe viene invocata attraverso la definizione di un oggetto statico. La visibilità del suo costruttore viene modificata da public a private, così che non sia possibile istanziare la classe dall'esterno e fare in modo che soltanto la classe può istanziare sé stessa.

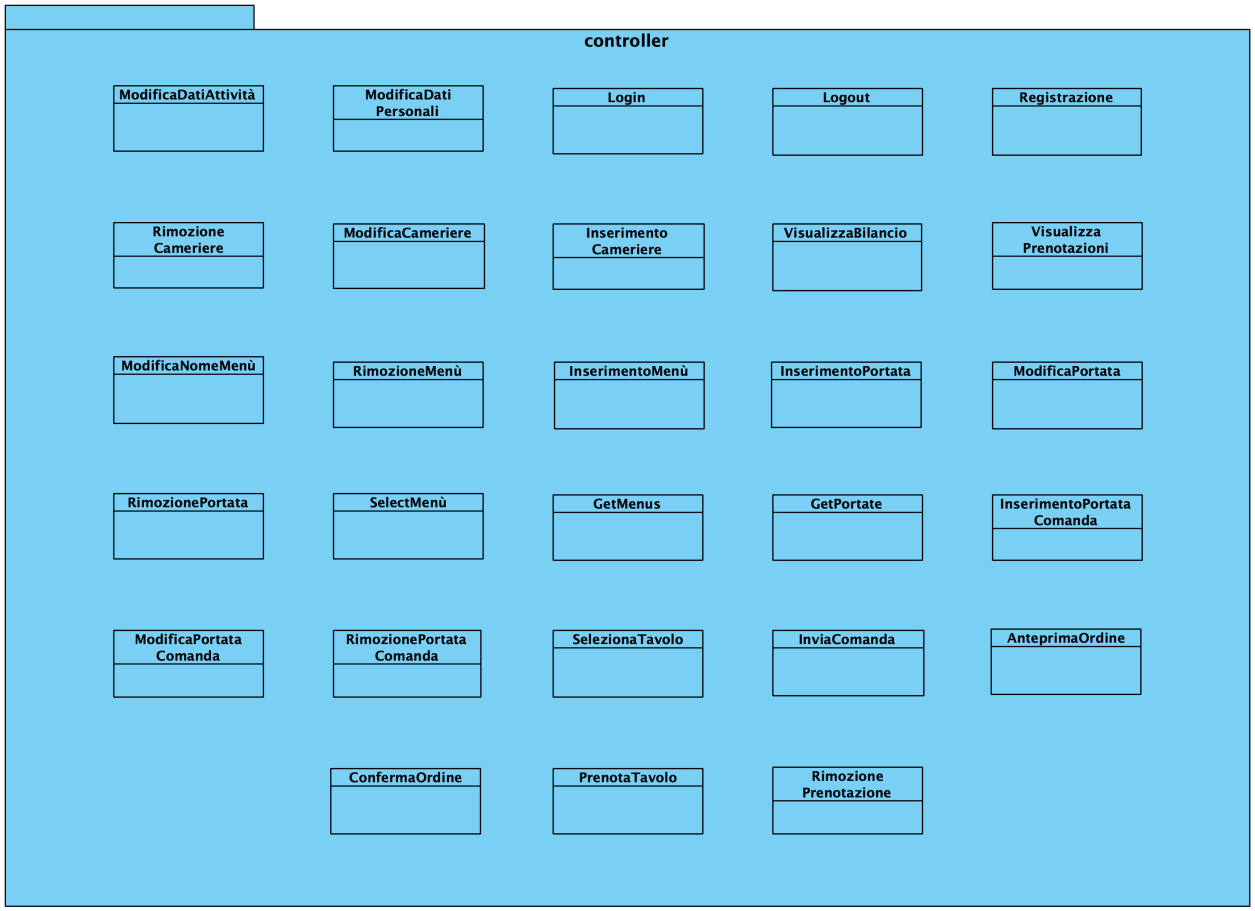


3. Packages

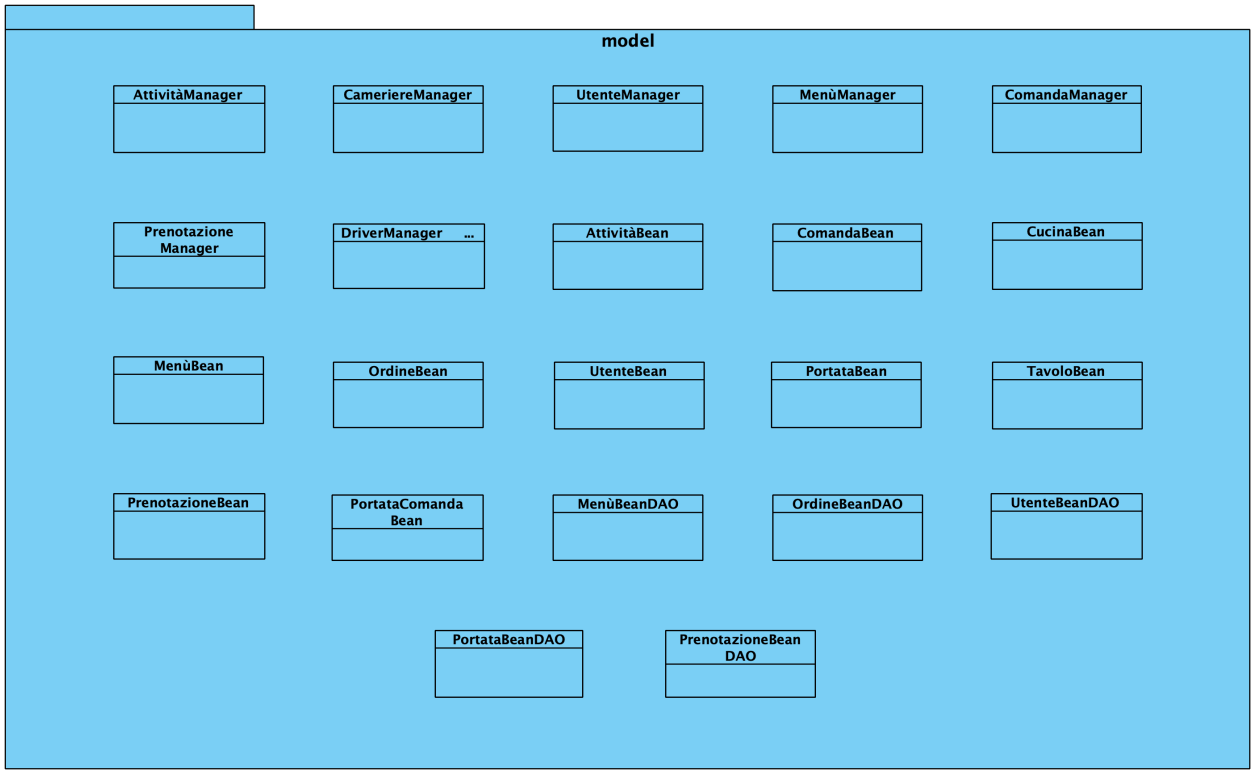
3.1. View



3.2. Controller



3.3. Model



4. Class interfaces

Alcune classi non sono state descritte, in quanto presentano solo metodi getter e setter:

- AttivitàBean
- MenùBean
- PortateBean
- PrenotazioneBean
- PortataComandaBean
- OrdineBean
- ComandaBean
- CucinaBean

4.1. UtenteManager

Nome	**UtenteManager.**
Descrizione	Questa classe contiene i metodi utili per l'utente.

Signature dei metodi	login(String username, String password);
	modificaDatiPersonali(String nome, String cognome, String password);
	registrazione(String username, String password, String nome, String cognome);

Pre-condizioni	**context** UtenteManager::login(username, password)
	username != null && password != null;
	context CamerieреManager::modificaDatiPersonali(username, password, nome, cognome);
	username != null && password != null && nome != null && cognome != null;
	context UtenteManager::registrazione(nome, cognome, username, password)
	getUtente(username) = null;

Post-condizioni	**context** UtenteManager::login(username, password)
	getUtente(username, password) != null;
	context UtenteManager::modificaDatiPersonali(username, password, nome, cognome);
	old_password = password && old_nome = nome && old_cognome = cognome;
	context UtenteManager::registrazione(nome, cognome, username, password)
	utente.getNumUtenti() + 1;

Invariante	

4.2.MenùManager

Nome	**MenùManager.**
Descrizione	Questa classe contiene tutti i metodi per gestire i menù.
Signature dei metodi	inserimentoMenù(String nome, GregorianCalendar data) modificaMenù(int id_menù, String new_nome) rimuoviMenù(int id_menù) inserimentoPortata(String nome, double prezzo, String tipo, String descrizione) modificaPortata(int id_portata, String new_nome, double new_prezzo, String new_tipo, String new_descrizione) rimozionePortata(int id_portata)
Pre-condizioni	**context** MenùManager::inserimentoMenù(nome, data) getMenù(nome) = null **context** MenùManager::modificaMenù(id_menù, new_nome) getMenù(id_menù) != null **context** MenùManager::rimozioneMenù(id_menù) getMenù(id_menù) != null **context** MenùManager::inserimentoPortata(nome, prezzo, tipo, descrizione) getPortata(nome) != null **context** MenùManager::modificaPortata(nome, prezzo, tipo, descrizione) getPortata(id_portata) != null

Post-condizioni	**context** MenùManager::inserimentoMenù(nome, data)
	getNumMenù() = getNumMenù() + 1;
	context MenùManager::modificaMenù(id_menù)
	getNomeMenù() = new_nome;
	context MenùManager::rimozioneMenù(id_menù)
	getNumMenù() = getNumMenù() – 1;
	context MenùManager::inserimentoPortata(nome, prezzo, tipo, descrizione)
	getNumPortate() = getNumPortate() + 1;
	context MenùManager::modificaPortata(id_portata, new_nome, new_prezzo, new_tipo, new_descrizione)
	getPortata(id_portata).getNome() = new_nome &&
	getPortata(id_portata).getPrezzo() = new_prezzo &&
	getPortata(id_portata).getTipo() = new_tipo &&
	getPortata(id_portata).getDescrizione() == new_descrizione;

Invariante	

4.3. AttivitàManager

Nome	**AttivitàManager.**
Descrizione	Questa classe contiene i metodi utili per l'attività(ristorante).

Signature dei metodi	modificaDatiAttività(String nome, String indirizzo);
	modificaDatiPersonali(String nome, String cognome, String password);
	registrazione(String username, String password, String nome, String cognome);

Pre-condizioni	**context** UtenteManager: login(username, password) username != null && password != null **context** UtenteManager: modificaDatiPersonale(nome, cognome, password) nome != null && cognome != null && password != null **context** UtenteManager: registrazione(nome, cognome, username, password) nome != null && cognome != null && username != null && password != null
Post-condizioni	**context** UtenteManager: modificaDatiPersonale(nome, cognome, password) **context** UtenteManager: registrazione(nome, cognome, username, password) UtenteBean utente;
Invariante	

4.4. CamerieреManager

Nome	**CamerieреManager**.
Descrizione	Questa classe contiene i metodi utili per l'attività(ristorante).
Signature dei	login(String username, String password);

metodi	modificaDatiPersonali(String nome, String cognome, String password); registrazione(String username, String password, String nome, String cognome);
Pre-condizioni	**context** UtenteManager: login(username, password) username != null && password != null **context** UtenteManager: modificaDatiPersonale(nome, cognome, password) nome != null && cognome != null && password != null **context** UtenteManager: registrazione(nome, cognome, username, password) nome != null && cognome != null && username != null && password != null
Post-condizioni	**context** UtenteManager: modificaDatiPersonale(nome, cognome, password) **context** UtenteManager: registrazione(nome, cognome, username, password) UtenteBean utente;
Invariante	

4.5. ComandaManger

Nome	**ComandaManager.**
Descrizione	Questa classe contiene i metodi utili per gestire l'ordinazione di un tavolo.

Signature dei metodi	selezionaTavolo(int numTavolo; int numPersone);
	inserimentoPortataComanda(Comanda c, PortataComanda p);
	modificaPortataComanda(Comanda c, int idPortataComanda, boolean consegnato, int quantità, String note);
	rimozionePortataComanda(Comanda c, int idPortataComanda);
	inviaComanda(Comanda c);
	anteprimaOrdine(Comanda c);
	confermaOrdine()

Pre-condizioni	**context** ComandaManager::selezionaTavolo(int numTavolo; int numPersone);
	numPersone > 0;
	context ComandaManager::inserimentoPortataComanda(Comanda c, PortataComanda p);
	c != null && p != null;
	context ComandaManager::modificaPortataComanda(Comanda c, int idPortataComanda, boolean consegnato, int quantità, String note);
	c != null && idPortataComanda > 0
	context ComandaManager::rimozionePortataComanda(Comanda c, int idPortataComanda);
	getNumPortateComanda() > 0;
	context ComandaManager::inviaComanda(Comanda c);
	getNumPortateComanda() > 0 && getPortataComandaIsConsegnata() = false;
	context ComandaManager::anteprimaOrdine(Comanda c);
	getNumPortateComandaConsegnate() > 0;
	context ComandaManager::confermaOrdine()
	getNumPortateComandaConsegnate() = getNumPortateComanda();

Post-condizioni	**context** ComandaManager::selezionaTavolo(int numTavolo; int numPersone);
	Tavolo t != null;
	context ComandaManager::inserimentoPortataComanda(Comanda c, PortataComanda p);
	getNumPortateComanda() + 1;
	context ComandaManager::modificaPortataComanda(Comanda c, int idPortataComanda, boolean consegnato, int quantità, String note);
	getIdPortataComanda = idPortataComanda && consegnato = true &&
	old_quantità = quantità && old_note = note;
	context ComandaManager::rimozionePortataComanda(Comanda c, int idPortataComanda);
	getNumPortateComanda() - 1 ;
	context ComandaManager::inviaComanda(Comanda c);
	inviaPortate(cucina, portateNonConsegnate);
	context ComandaManager::anteprimaOrdine(Comanda c);
	getNumPortateComandaConsegnate > 0;
	context ComandaManager::confermaOrdine()
	Ordine o= new Ordine && setNumPer(numPersone) &&
	setPortateConsegnate(portateConsegnate) && destroy.tavolo;

Invariante	numPersone > 0;

4.6. PrenotazioneManager

Nome	**PrenotazioneManager.**
Descrizione	Questa classe contiene tutti i metodi per gestire le prenotazioni.
Signature dei metodi	prenotaTavolo(String username, int num_coperti, GregorianCalendar data, String descrizione) rimozionePrenotazione(int id_prenotazione) visualizzaPrenotazioni(String username)

Pre-condizioni	**context** PrenotazioneManager::prenotaTavolo(username, num_coperti, data, descrizione)
	getNumCoperti() > 0 && getData() >= getDataOggi();
	context PrenotazioneManager::rimozionePrenotazione(id_prenotazione)
	getNumPrenotazioni(username) >= 1 &&
	getPrenotazione(id_prenotazione) != null;

Post-condizioni	**context** PrenotazioneManager::prenotaTavolo(username, num_coperti, data, descrizione)
	getNumPrenotazioni(username) = getNumPrenotazioni(username) + 1;
	context PrenotazioneManager::rimozionePrenotazione(id_prenotazione)
	getNumPrenotazioni(username) = getNumPrenotazioni(username) - 1;

Invariante	

4.7. UtenteBeanDAO

Nome	**UtenteBeanDAO.**
Descrizione	Questa classe contiene i metodi che permettono all'utente d'interagire con il Database.
Signature dei metodi	doSave(String username, String password, String nome, String cognome) doRetrieveByKey(String username) doUpdate(String username, String password, String nome, String cognome) doDelete(String username) doRetrieveAll()

Pre-condizioni	**context** UtenteBeanDAO::doSave(username, password, nome, cognome)
	doRetrieveByKey(username) = null;
	context UtenteBeanDAO::doRetrieveByKey(username)
	getUtente(username) != null;
	context UtenteBeanDAO::doUpdate(username, password, nome, cognome)
	doRetrieveByKey(username) != null && password != null && nome != null && cognome != null;
	context UtenteBeanDAO::doDelete(username)
	doRetrieveByKey(username) != nul;

Post-condizioni	**context** UtenteBeanDAO::doSave(username, password, nome, cognome)
	getNumUtenti() + 1;
	context UtenteBeanDAO::doRetrieveByKey(username)
	utente = doRetrieveByKey(username) && utente != null;
	context UtenteBeanDAO::doDelete(username)
	getNumUtenti() - 1;

Invariante	

4.8. MenùBeanDAO

Nome	**MenùBeanDAO.**
Descrizione	Questa classe contiene i metodi utili per gestire i dati persistenti del menù.
Signature dei metodi	doSave(int idMenù, String nome, GregorianCalendar data); doRetrieveByKey(int idMenù); doUpdate(int idMenù, String nome); doDelete(int idMenù); doRetrieveAll();

Pre-condizioni	**context** MenùBeanDAO::doSave(idMenù, nome, data); getIdMenù(idMenù) = null && idMenù > 0 && nome != null && (data!= null && data >= getDataOggi()); **context** MenùBeanDAO::doRetrieveByKey(idMenù); getMenù() != null; **context** MenùBeanDAO::doUpdate(idMenù, nome); getIdMenù(idMenù) != null && nome != null; **context** MenùBeanDAO::doDelete(idMenù); getIdMenù(idMenù) != null;
Post-condizioni	**context** MenùBeanDAO::doSave(idMenù, nome, data); getNumMenù() + 1; **context** MenùBeanDAO::doRetrieveByKey(idMenù); menù = getMenù(idMenù) != null; **context** MenùBeanDAO::doUpdate(idMenù, nome); menù = getMenù(idMenù); menù.getNome = nome; **context** MenùBeanDAO::doDelete(idMenù); getNumMenù() - 1;
Invariante	

4.9. PortataBeanDAO

Nome	**PortataBeanDAO.**
Descrizione	Questa classe contiene i metodi utili per gestire i dati persistenti della portata.

Signature dei metodi	doSave(int idPortata, String nome, String tipo, double prezzo, String descrizione);
	doRetrieveByKey(int idPortata);
	doUpdate(int idPortata, String nome, String tipo, double prezzo, String descrizione);
	doDelete(int idPortata);
	doRetrieveAll();

Pre-condizioni	**context** PortataBeanDAO::doSave(idPortata, nome, tipo, prezzo, descrizione);
	getIdPortata(idPortata) = null && idPortata > 0 && nome != null && tipo != null && prezzo != 0;
	context PortataBeanDAO::doRetrieveByKey(idPortata);
	getIdPortata() != null;
	context PortataBeanDAO::doUpdate(idPortata, nome, tipo, prezzo, descrizione);
	getIdPortata(idPortata) != null && nome != null && nome != null && tipo != null && prezzo != 0;
	context PortataBeanDAO::doDelete(idPortata);
	getIdPortata(idPortata) != null;

Post-condizioni	**context** PortataBeanDAO::doSave(idPortata, nome, tipo, prezzo, descrizione);
	getNumPortate() + 1;
	context PortataBeanDAO::doRetrieveByKey(idPortata);
	portata = getPortata(idPortata) != null;
	context PortataBeanDAO::doUpdate(idPortata, nome, tipo, prezzo, descrizione);
	portata = getPortata(idPortata);
	portata.getNome() = nome && portata.getTipo() = tipo &&
	portata.getPrezzo() = prezzo && portata.getDescr() = descrizione;
	context PortataBeanDAO::doDelete(idPortata);
	getNumPortate() - 1;

Invariante	

4.10. PrenotazioneBeanDAO

Nome	**PrenotazioneBeanDAO.**
Descrizione	Questa classe contiene i metodi utili per gestire i dati persistenti della prenotazione.

Signature dei metodi	doSave(int idPrenotazione, int numPersone, GregorianCalendar data, String descrizione);
	doRetrieveByKey(int idPrenotazione);
	doDelete(int idPrenotazione);
	doRetrieveAll();

Pre-condizioni	**context** PrenotazioneBeanDAO::doSave(idPrenotazione, numPersone, data, descrizione);
	getidPrenotazione(idPrenotazione) = null && idPrenotazione > 0 &&
	numPersone > 0 && (data != null && data >= getDataOggi()) &&
	descrizione != null;
	context PrenotazioneBeanDAO::doRetrieveByKey(idPrenotazione);
	getIdPrenotazione() != null;
	context PrenotazioneBeanDAO::doDelete(idPrenotazione);
	getIdPrenotazione(idPrenotazione) != null;

Post-condizioni	**context** PrenotazioneBeanDAO::doSave(idPortata, nome, tipo, prezzo, descrizione);
	getNumPrenotazioni() + 1;
	context PrenotazioneBeanDAO::doRetrieveByKey(idPrenotazione);
	prenotazione = getPrenotazione(idPrenotazione) != null;
	context PrenotazioneBeanDAO::doDelete(idPrenotazione);
	getNumPrenotaziioni() - 1;

Invariante	

4.11. OrdineBeanDAO

Nome	**OrdineBeanDAO.**
Descrizione	Questa classe contiene i metodi utili per gestire i dati persistenti dell'ordine.

Signature dei metodi	doSave(int idOrdine, int numPersone, GregorianCalendar data, double totale);
	doRetrieveByKey(int idOrdine);
	doRetrieveAll();

Pre-condizioni	**context** OrdineBeanDAO::doSave(idOrdine, numPersone, data, totale) getIdOrdine(idOrdine) = null && idOrdine > 0 && numPersone > 0 && (data != null && data >= getDataOggi()) && totale != 0; **context** OrdineBeanDAO::doRetrieveByKey(int idOrdine); getIdOrdine() != null;

Post-condizioni	**context** OrdineBeanDAO::doSave(idOrdine, numPersone, data, totale)
	getNumOrdini() + 1;
	context OrdineBeanDAO::doRetrieveByKey(int idOrdine);
	ordine = getPortata(idOrdine) != null;

Invariante	

4.12. DriverManagerConnectionPool

Nome	**DriverManagerConnectionPool**
Descrizione	Questa classe contiene i metodi per gestire le connessioni al DataBase.

Signature dei metodi	createDBConnection();
	getConnection();
	releaseConnection(Connection connection);

Pre-condizioni	**context** DriverManagerConnectionPool::createDBConnection();

Post-condizioni	**context** DriverManagerConnectionPool::createDBConnection()
	Connection new_connection = DriverManager.getConnection(string for connection to the DataBase);
	context DriverManagerConnectionPool::getConnection()
	Connection connection;
	If (list of Connection is not empty) connection = list.getConnection()
	else connection = new Connection();
	context DriverManagerConnectionPool::releaseConnection(connection)
	if (connection != null)
	getNumConnection() = getNumConnection() + 1

Invariante	

5. Glossario

Database: archivio in cui le informazioni sono organizzate in tabelle che consentono le ricerche e gli aggiornamenti.
Query: richiesta di dati o informazioni da una tabella o da una combinazione di tabelle del database.
Trade-offs: compromessi.
Package: pacchetto. Una collezione di classi e interfacce tra loro correlate.
Free: Gratuito.
Open source: software il cui codice sorgente è rilasciato con una licenza che lo rende modificabile da parte di chiunque.
ODD: Object Design Document.