Object Design Document

E-Balance

Riferimento	E-Balance_C17_ODD
Versione	1.0
Data	06/02/2024
Destinatario	F. Ferrucci, F. Palomba
Presentato da	Simone Cirma, Mario De Luca, Antonio Di Giorgio, Donato Folgieri, Daniela Palma, Emanuele Vitale
Approvato da	Matteo Ercolino, Simone Silvestri

Revision History

Data	Versione	Descrizione	Autori
03/01/2024	0.1	Prima stesura del documento	Antonio Di Giorgio
04/01/2024	0.2	Introduzione, Design Object Trade-Off e Linee guida documentazione	Simone Cirma, Mario De Luca, Emanuele Vitale
05/01/2024	0.3	Identificazione Packages	Simone Cirma , Antonio Di Giorgio, Donato Folgieri
05/01/2024	0.4	Identificazione Class Interfaces	Antonio Di Giorgio, Daniela Palma, Emanuele Vitale
05/01/2024	0.5	Identificazione Design Patterns	Antonio Di Giorgio, Daniela Palma, Emanuele Vitale
05/01/2024	0.6	Class Diagram	Simone Cirma, Mario De Luca, Donato Folgieri
06/01/2024	0.7	Revisione Class Interfaces	Antonio Di Giorgio, Daniela Palma
09/01/2024	0.8	Revisione documento	Antonio Di Giorgio, Daniela Palma, Emanuele Vitale
06/02/2024	1.0	Revisione e stesura finale del documento	Tutti i team member

Sommario

1. Introduzione

E-Balance è un sistema software che si propone di migliorare ed ottimizzare la gestione energetica dell'Università degli Studi di Salerno.
In questa prima sezione del documento, verranno descritti i trade-offs e le linee guida per la fase di implementazione, riguardanti la nomenclatura, la documentazione e le convenzioni sui formati.

1.01 Object Design Trade-Off

Trade-off	Descrizione
Affidabilità VS Costo di sviluppo	Il sistema deve assicurare un elevato grado di affidabilità, pertanto risulterà imprescindibile destinare oltre il 60% delle risorse finanziarie disponibili per garantire tale requisito fondamentale.
Affidabilità VS Tempo di sviluppo	Il sistema deve garantire un livello di affidabilità elevato, di conseguenza sarà necessario impiegare il 60% del tempo di sviluppo a disposizione per realizzare quest'aspetto.

1.02 Linee Guida per la Documentazione delle Interfacce

Le linee guida contengono un elenco di norme che gli sviluppatori sono tenuti a seguire nel corso della progettazione delle interfacce. Per la loro formulazione, ci si è ispirati alla convenzione Java conosciuta come Sun Java Coding Conventions [Sun, 2009].
Di seguito una lista di link alle convenzioni usate per definire le linee guida:

- Java Sun: https://checkstyle.sourceforge.io/sun_style.html
- HTML: https://www.w3schools.com/html/html5_syntax.asp

1.03 Definizioni, acronimi e abbreviazioni

Vengono riportati di seguito alcune definizioni presenti nel documento:

- **Package**: raggruppamento di classi, interfacce o file correlati;
- **Design pattern**: template di soluzioni a problemi ricorrenti impiegati per ottenere riuso e flessibilità;
- **Interfaccia**: insieme di signature delle operazioni offerte dalla classe;
- **View**: nel pattern MVC rappresenta ciò che viene visualizzato a schermo da un utente e che gli permette di interagire con le funzionalità offerte dalla piattaforma;
- **lowerCamelCase**: è la pratica di scrivere frasi in modo tale che ogni parola o abbreviazione nel o mezzo della frase inizi con una lettera maiuscola, senza spazi o punteggiatura intermedi;
- **UpperCamelCase**: è la pratica di scrivere frasi in modo tale che ogni parola o abbreviazione inizi o con una lettera maiuscola, senza spazi o punteggiatura intermedi.

1.04 Riferimenti

Di seguito una lista di riferimenti ad altri documenti utili durante la lettura:

- Documento di Statement of Work relativo a questo progetto. Link alla risorsa: SOW
- Requirements Analysis Document relativo a questo progetto. Link alla risorsa: RAD
- System Design Document relativo a questo progetto. Link alla risorsa: SDD

2. Packages

In questa sezione viene mostrata la suddivisione del sistema in package, in base a quanto definito nel documento di System Design. Tale suddivisione è motivata dalle scelte architetturali prese e ricalca la struttura di directory standard definita da Maven.

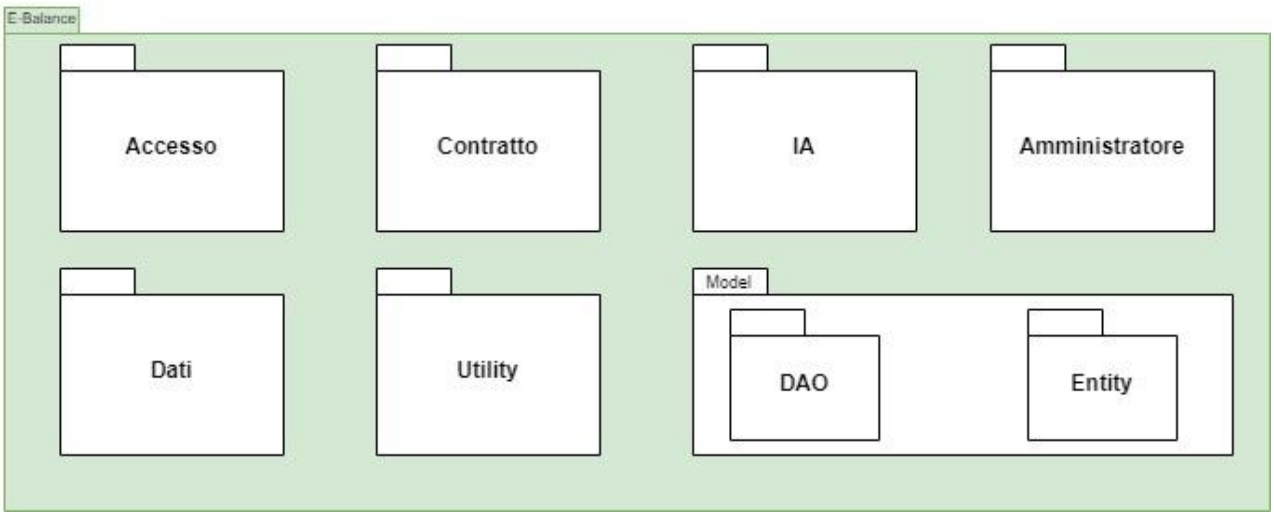
- **.idea**
- **.mvn**, contiene tutti i file di configurazione per Maven
- **src**, contiene tutti i file sorgente
    - **main**
        - **java**, contiene le classi Java relative alle componenti Control e Model
        - **webapp**
            - **css**, composta dai fogli di stile
            - **js**, composta dai file JavaScript
            - **static**, contenente pagine jsp che non cambiano il loro contenuto in base ai dati persistenti
            - **img**
            - **WEB-INF**
                - **jsp**, composta da tutte le pagine dinamiche relative alle View.
    - **test**, contiene tutto il necessario per il testing
        - **java**, contiene le classi Java per l'implementazione del testing
- **target**, contiene tutti i file prodotti dal sistema di build di Maven

2.01 Package E-Balance

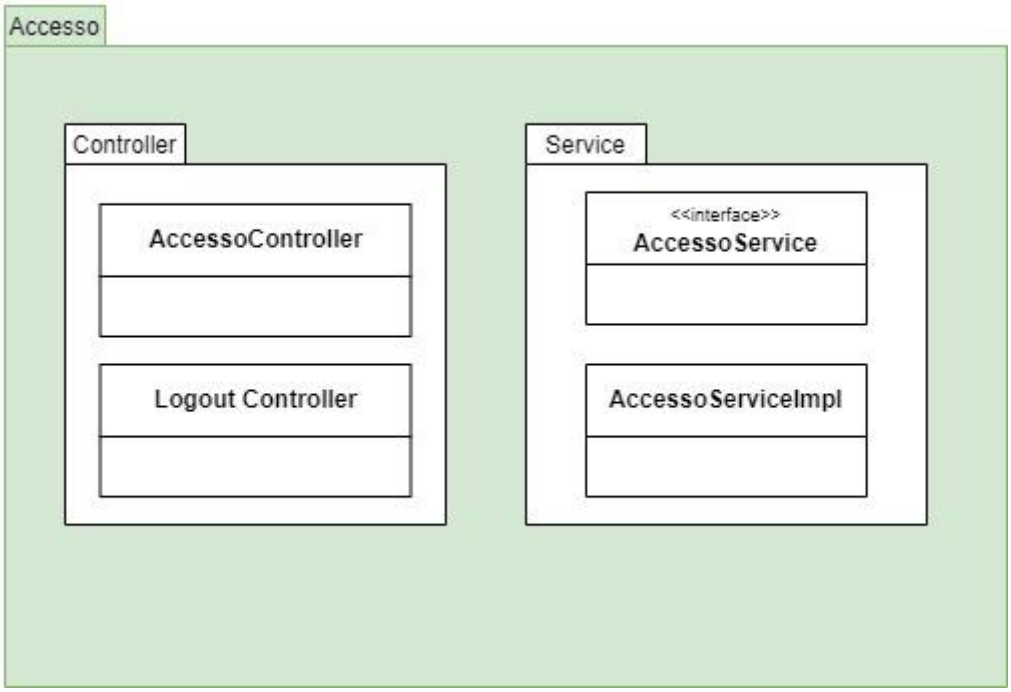
Nella presente sezione si mostra la struttura del package principale di E-Balance. La struttura generale è stata ottenuta a partire da due principali scelte:

1. Creare un package separato per ogni sottosistema, contenente le classi service e controller del sottosistema, ed eventuali classi di utilità usate unicamente da esso.
2. Creare un package separato per le classi del model, contenente le classi entity e i DAO per l'accesso al DB. Tale scelta è stata presa vista l'elevata complessità del database di E-Balance che prevede numerose relazioni tra le entità. Si è quindi preferito tenere tutto in un package separato e collegato a tutti gli altri package dei sottosistemi.

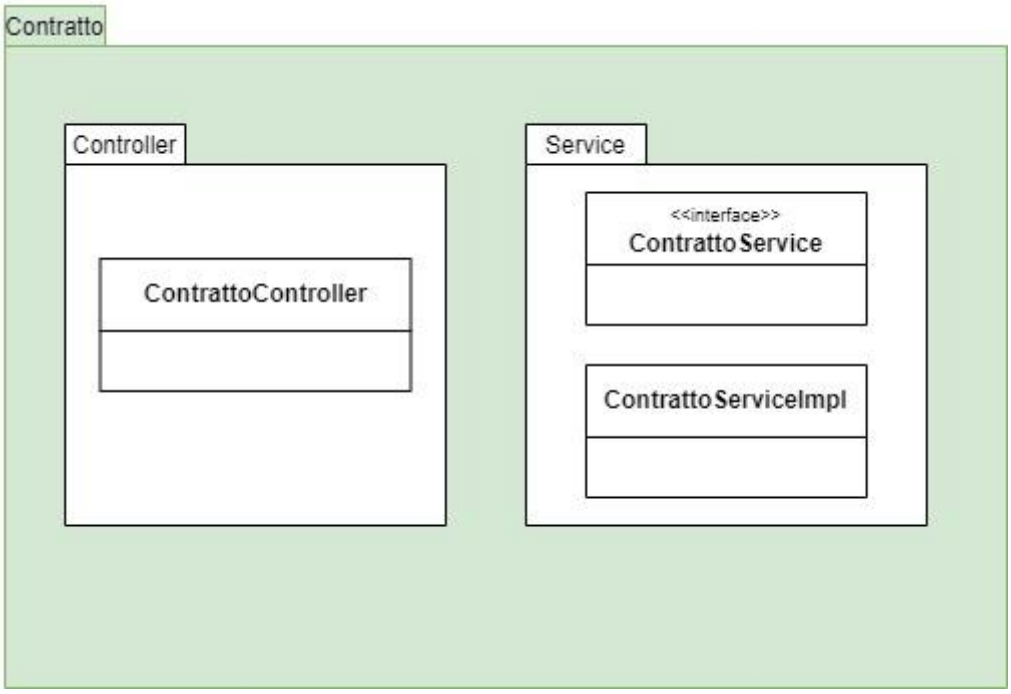
Per ciò che concerne la dipendenza tra i packages, la suddivisione precedentemente illustrata a portato alla creazione di una relazione tra il package model e tutti gli altri package del sistema.



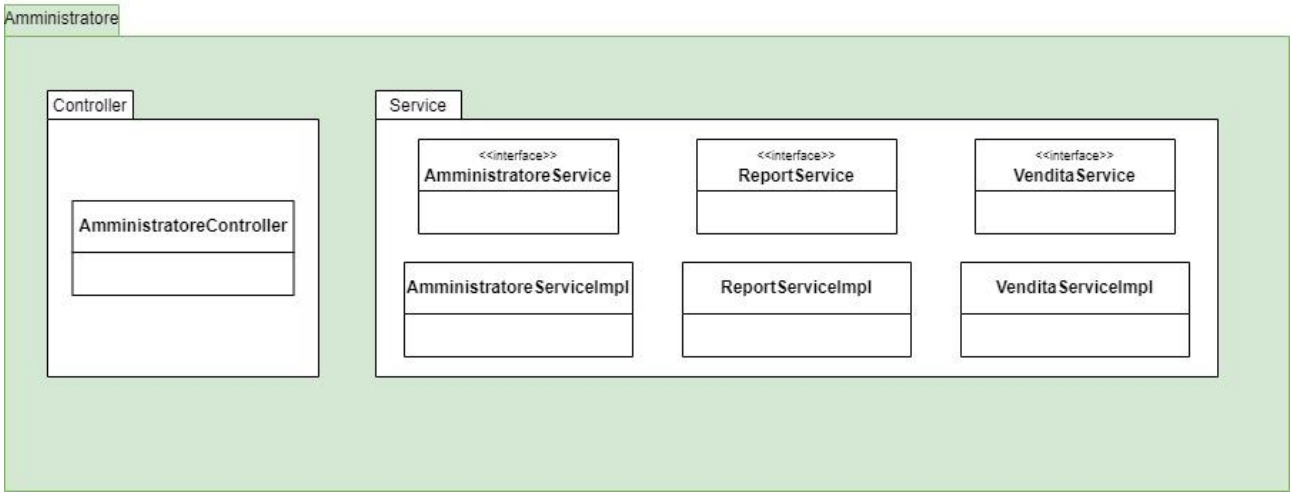
2.01.01 Package Accesso



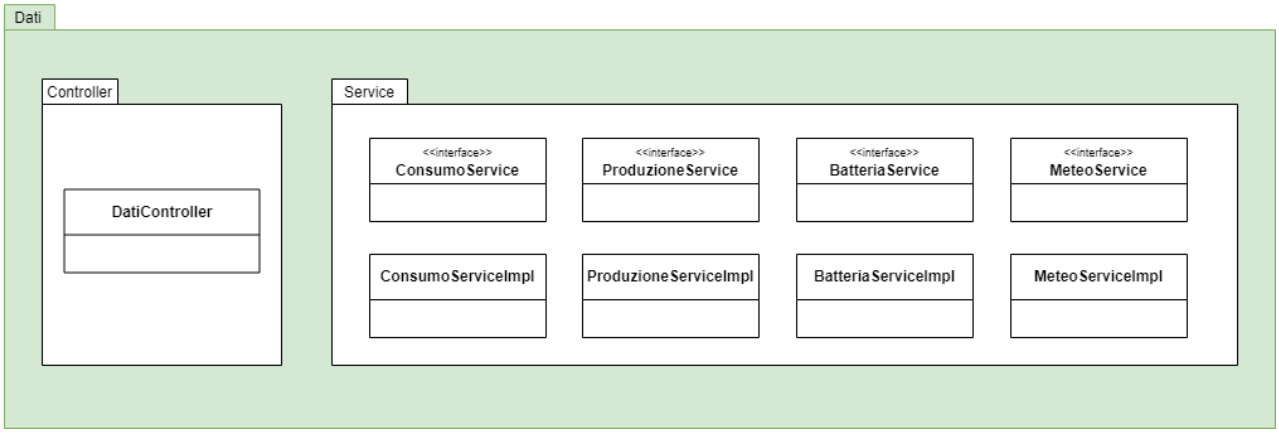
2.01.02 Package Contratto



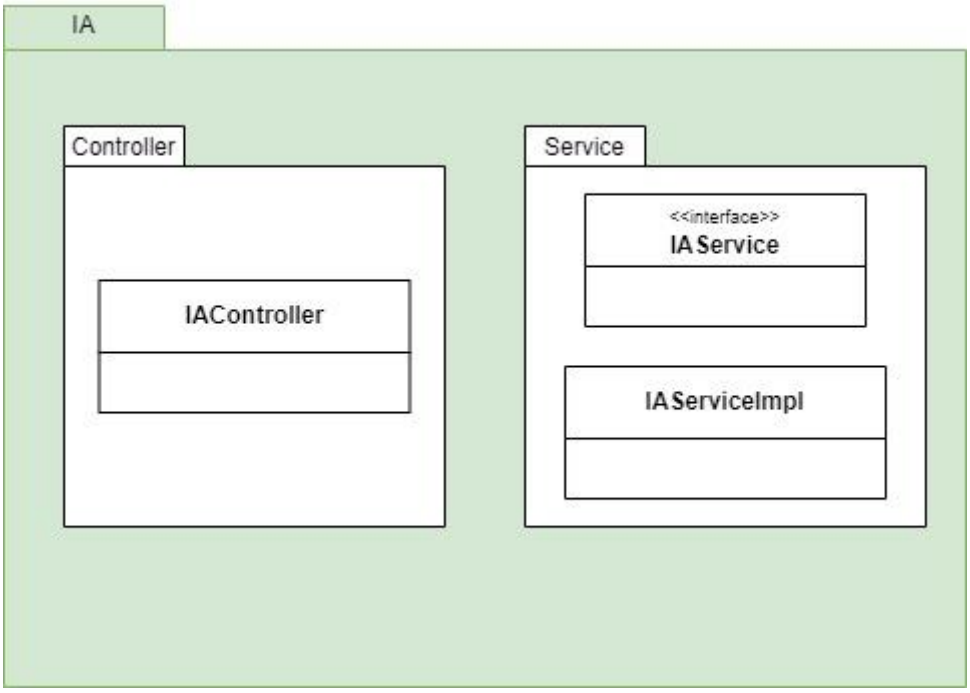
2.01.03 Package Amministratore



2.01.04 Package Dati



2.01.05 Package IA



3. Class interfaces

Di seguito saranno presentate le interfacce di ciascun package.

3.01 Package Accesso

Nome Classe	AccessoService
Descrizione	Questa classe permette di gestire le operazioni relative al login da parte dell'amministratore.
Metodi	+login(String email, String pass): AmministratoreBean
Invariante di classe	/

Nome Metodo	+login(String email, String password)
Descrizione	Questo metodo consente all'amministratore di accedere al sistema.
Pre-condizione	/
Post-condizione	**context**: AccessoService::login(email, password) **post**: isSuper(loggedUser) || isRegistrato(loggedUser)==true

3.02 Package Amministratore

3.02.01 AmministratoreService

Nome Classe	AmministratoreService
Descrizione	Questa classe permette di gestire le operazioni relative al profilo dell'amministratore.
Metodi	+verificaSuperAdmin() : boolean +visualizzaAmministratori : List<AmministratoreBean> +aggiornaAmministratore(AmministratoreBean amministratore) : AmministratoreBean +aggiungiAmministratore(AmministratoreBean amministratore) +getById(int id) : AmministratoreBean +rimuoviAmministratore(int idAmministratore) +verificaPresenzaEmail(String email) : boolean
Invariante di classe	/

Nome Metodo	+verificaSuperAdmin()
Descrizione	Questo metodo consente di verificare se colui che ha loggato è il super-admin
Pre-condizione	/
Post-condizione	**context**: AmministratoreService::verificaSuperAdmin() **post**: isSuper(loggedUser) == true

Nome Metodo	+visualizzaAmministratori()
Descrizione	Questo metodo consente di visualizzare gli amministratori registrati nel sistema
Pre-condizione	/
Post-condizione	/

Nome Metodo	+aggiornaAmministratore(AmministratoreBean amministratore)
Descrizione	Questo metodo consente di aggiornare i dati di un profilo amministratore.
Pre-condizione	/
Post-condizione	/

Nome Metodo	+aggiungiAmministratore(AmministratoreBean amministratore)
Descrizione	Questo metodo consente al Super-Admin di aggiungere un nuovo amministratore.
Pre-condizione	**context**: AmministratoreService:: aggiungiAmministratore(amministratore) **pre**: not visualizzaAmministratori(null).contains(amministratore) && isSuper(loggedUser) == true
Post-condizione	**context**: AmministratoreService:: aggiungiAmministratore(amministratore) **post**: visualizzaAmministratori(null).contains(amministratore) && visualizzaAmministratori(null).size() = @pre. visualizzaAmministratori(null).size() +1

Nome Metodo	+getById(int id)
Descrizione	Questo metodo consente ricercare un amministratore per id.
Pre-condizione	**context:** AmministratoreService::getById(id) **pre**: id != null
Post-condizione	/

Nome Metodo	+rimuoviAmministratore(int id)
Descrizione	Questo metodo consente di rimuovere un amministratore.
Pre-condizione	**context**: AmministratoreService::rimuoviAmministratore (id) **pre**: visualizzaAmministratori(null).contains(getById(id))
Post-condizione	**context**: AmministratoreService::rimuoviAmministratore(id) **post**: not visualizzaAmministratori(null).contains(getById(id)) && visualizzaAmministratori(null).size() = @pre. visualizzaAmministratori(null).size() -1

Nome Metodo	+verificaPresenzaEmail(String email)
Descrizione	Questo metodo consente di verificare se un email è già utilizzata per un amministratore.
Pre-condizione	**context**: AmministratoreService::verificaPresenzaEmail (email) **pre**: email != null
Post-condizione	/

3.02.02 ReportService

Nome Classe	ReportService
Descrizione	Questa classe permette di gestire le operazioni relative al report da parte dell'amministratore.
Metodi	+visualizzaReport() : List<ReportBean> +aggiungiReport(ReportBean bean) +generaReport(Date dataInizio, Date dataFine, String servletPath, HttpSession session): ReportBean
Invariante di classe	/

Nome Metodo	+visualizzaReport()
Descrizione	Questo metodo permette all'amministratore di visualizzare l'elenco dei report generati in passato.
Pre-condizione	/
Post-condizione	/

Nome Metodo	+aggiungiReport(ReportBean report)
Descrizione	Questo metodo permette al sistema di aggiungere il report appena generato al database.
Pre-condizione	/
Post-condizione	/

Nome Metodo	+generaReport(Date dataInizio, Date dataFine, String servletPath, HttpSession session)
Descrizione	Questo metodo permette all'amministratore di generare e visualizzare un report.
Pre-condizione	/
Post-condizione	/

3.02.03 VenditaService

Nome Classe	VenditaService
Descrizione	Questa classe permette di gestire le operazioni relative alla vendita da parte dell'amministratore.
Metodi	+getVendite(Date dataInizio, Date dataFine) : List<VenditaBean> +getRicavoTotalePerData(Date dataInizio, Date dataFine) : float +effettuaVendita(int idAmministratore)
Invariante di classe	/

Nome Metodo	+getVendite(Date dataInizio, Date dataFine)
Descrizione	Questo metodo consente all'amministratore ottenere le vendite effettuate per intervallo di date.
Pre-condizione	**context**: VenditaService::getVendite (dataInizio, dataFine) **pre :** dataInizio != null && dataFine != null
Post-condizione	/

Nome Metodo	+getRicavoTotalePerData(Date dataInizio, Date dataFine)
Descrizione	Questo metodo consente all'amministratore di ottenere il ricavo totale per intervallo di date.
Pre-condizione	**context**: VenditaService:: getRicavoTotalePerData (dataInizio, dataFine) **pre :** dataInizio != null && dataFine != null
Post-condizione	/

Nome Metodo	+effettuaVendita(int idAmministratore)
Descrizione	Questo metodo consente all'amministratore di vendere l'energia in eccesso immagazzinata.
Pre-condizione	**context**: VenditaService::effettuaVendita(idAmministratore) **pre**: idAmministratore != null
Post-condizione	**context**: VenditaService::effettuaVendita(idAmministratore) **post**: getNumVendite_postEffettuaVendita = getNumVendite_preEffettuaVendita +1

3.03 Package Contratto

Nome Classe	ContrattoService
Descrizione	Questa classe permette di gestire le operazioni relative al contratto del sistema.
Metodi	+aggiornaContratto(ContrattoBean contratto): ContrattoBean +aggiungiContratto (ContrattoBean contratto) +visualizzaContratto(): ContrattoBean +visualizzaStoricoContratti(): List<ContrattoBean> +verificaPrimoContratto() : boolean +getContrattoAttivo(Date dataInizio, Date dataFine) : ContrattoBean +ottieniPrezzoVendita() : float
Invariante di classe	/

Nome Metodo	+aggiornaContratto(ContrattoBean contratto)
Descrizione	Questo metodo consente di aggiornare i dati del contratto attuale.
Pre-condizione	/
Post-condizione	/

Nome Metodo	+aggiungiContratto(ContrattoBean contratto)
Descrizione	Questo metodo consente agli amministratori di aggiungere un nuovo contratto.
Pre-condizione	**context**: ContrattoService:: aggiungiContratto(contratto) **pre**: not visualizzaStoricoContratti(null).contains(contratto)
Post-condizione	**context**: ContrattoService:: aggiungiContratto(contratto) **post**: visualizzaStoricoContratti (null).contains(contratto) && visualizzaStoricoContratti (null).size() = @pre. visualizzaStoricoContratti (null).size() +1

Nome Metodo	+visualizzaContratto()
Descrizione	Questo metodo restituisce il contratto attivo
Pre-condizione	/
Post-condizione	/

Nome Metodo	+visualizzaStoricoContratti()
Descrizione	Questo metodo restituisce la lista dello storico dei contratti registrati.
Pre-condizione	/
Post-condizione	/

Nome Metodo	+verificaPrimoContratto()
Descrizione	Questo metodo verifica se è presente almeno un contratto nel sistema.
Pre-condizione	/
Post-condizione	/

Nome Metodo	+getContrattoAttivo(Date dataInizio, Date dataFine)
Descrizione	Questo metodo restituisce il contratto attivo per il periodo indicato.
Pre-condizione	**context**: ContrattoService:: getContrattoAttivo(dataInizio, dataFine) **pre**: dataInizio != null && dataFine != null
Post-condizione	/

Nome Metodo	+ottieniPrezzoVendita()
Descrizione	Questo metodo restituisce il prezzo di vendita per il contratto attivo.
Pre-condizione	/
Post-condizione	/

3.04 Package Dati

3.04.01 ConsumoService

Nome Classe	ConsumoService
Descrizione	Questa classe permette di gestire le operazioni alla gestione e visualizzazione di tutti i dati relativi al consumo.
Metodi	+ottieniConsumiEdifici(float consumoEdifici []) : float[] +visualizzaStoricoConsumi() : List<ArchivioConsumoBean> +getConsumoPerData(Date dataInizio, Date dataFine) : float
Invariante di classe	/

Nome Metodo	+ottieniConsumiEdifici(float consumoEdifici [])
Descrizione	Ottiene i consumi degli edifici per il periodo corrente.
Pre-condizione	**context**: ConsumoService:: ottieniConsumoEdifici(consumoEdifici) **pre**: not consumoEdifici.isEmpty();
Post-condizione	/

Nome Metodo	+visualizzaStoricoConsumi()
Descrizione	Restituisce lo storico dei consumi nel sistema.
Pre-condizione	/
Post-condizione	/

Nome Metodo	+getConsumoPerData(Date dataInizio, Date dataFine)
Descrizione	Ottiene il consumo totale per un periodo specificato.
Pre-condizione	**context**: ConsumoService:: getConsumoPerData(dataInizio, dataFine) **pre**: dataInizio != null && dataFine != null
Post-condizione	/

3.04.02 ProduzioneService

Nome Classe	ProduzioneService
Descrizione	Questa classe permette di gestire le operazioni alla gestione e visualizzazione di tutti i dati relativi alla produzione.
Metodi	+ottieniProduzioneProdotta(float[] produzioneSorgente) : float[] +ottieniProduzioneSEN() : float +ottieniTipoSorgente() : List<TipoSorgenteBean> +energiaRinnovabileProdottaPerData(Date dataInizio, Date dataFine) : float
Invariante di classe	/

Nome Metodo	+ottieniProduzioneProdotta(float[] produzioneSorgente)
Descrizione	Ottiene la produzione di energia prodotta da diverse sorgenti.
Pre-condizione	**context**: ProduzioneService:: ottieniProduzioneProdotta(produzioneSorgente) **pre**: not produzioneSorgente.isEmpty();
Post-condizione	/

Nome Metodo	+ottieniProduzioneSEN()
Descrizione	Ottiene la produzione di energia dal Servizio Elettrico Nazionale (SEN).
Pre-condizione	/
Post-condizione	/

Nome Metodo	+ottieniTipoSorgente()
Descrizione	Ottiene i tipi di sorgenti disponibili per la produzione di energia.
Pre-condizione	/
Post-condizione	/

Nome Metodo	+energiaRinnovabileProdottaPerData(Date dataInizio, Date dataFine)
Descrizione	Ottiene l'energia rinnovabile prodotta nel sistema nel periodo specificato.
Pre-condizione	**context**: ProduzioneService:: energiaRinnovabileProdottaPerData(dataInizio, dataFine) **pre**: dataInizio != null && dataFine != null
Post-condizione	/

3.04.03 BatteriaService

Nome Classe	BatteriaService
Descrizione	Questa classe permette di gestire le operazioni alla gestione e visualizzazione di tutti i dati relativi alle batterie.
Metodi	+ottieniPercetualeBatteria() : float +ottieniNumBatterieAttive() : int +aggiornaBatteria(float energia, int numBatteria)
Invariante di classe	/

Nome Metodo	+ottieniPercetualeBatteria()
Descrizione	Ottiene la percentuale di carica della batteria nel sistema.
Pre-condizione	/
Post-condizione	/

Nome Metodo	+ottieniNumBatterieAttive()
Descrizione	Ottiene il numero di batterie attualmente attive nel sistema.
Pre-condizione	/
Post-condizione	/

Nome Metodo	+aggiornaBatteria(float energia, int numBatteria)
Descrizione	Aggiorna lo stato della batteria con l'energia fornita per un determinato numero di batterie.
Pre-condizione	**context**: BatteriaService:: aggiornaBatteria(energia, numBatteria) **pre**: energia != null && numBatteria != null
Post-condizione	/

3.04.04 MeteoService

Nome Classe	MeteoService
Descrizione	Questa classe permette di gestire le operazioni di gestione e visualizzazione di tutti i dati relativi al meteo.
Metodi	+getCondizioniMeteo() : List<MeteoBean> +getCondizioniSettimanali() : List<MeteoBean>
Invariante di classe	/

Nome Metodo	+getCondizioniMeteo()
Descrizione	Ottiene le condizioni meteo attuali.
Pre-condizione	/
Post-condizione	/

Nome Metodo	+getCondizioniSettimanali()
Descrizione	Ottiene le condizioni meteo settimanali.
Pre-condizione	/
Post-condizione	/

3.05 Package IA

Nome Classe	IAService
Descrizione	Questa classe permette di gestire le operazioni relative ai parametri IA.
Metodi	+visualizzaParametri() : List<ParametriIABean> +visualizzaInterazioneParametri() : List<InteragisceBean> +ottieniParametriAttivi() : List<InteragisceBean> +aggiornaPianoPersonalizzato(String preferenzaSorgente, int percentualeUtilizzoPannelli, int percentualeUtilizzoSEN, String sortableListData) +aggiornaPianoAttivo(String piano, int idAmministratore)
Invariante di classe	/

Nome Metodo	+visualizzaParametri()
Descrizione	Questo metodo consente di visualizzare la lista dei parametri IA.
Pre-condizione	/
Post-condizione	/

Nome Metodo	+visualizzaInterazioneParametri()
Descrizione	Restituisce la lista delle interazioni tra parametri IA e le sorgenti.
Pre-condizione	/
Post-condizione	/

Nome Metodo	+ottieniParametriAttivi()
Descrizione	Restituisce la lista dei parametri IA attivi.
Pre-condizione	/
Post-condizione	/

Nome Metodo	+aggiornaPianoPersonalizzato(String preferenzaSorgente, int percentualeUtilizzoPannelli, int percentualeUtilizzoSEN, String sortableListData)
Descrizione	Aggiorna il piano personalizzato dell'amministratore.
Pre-condizione	**context**: IAService:: aggiornaPianoPersonalizzato(preferenzaSorgente, percentualeUtilizzoPannelli, percentualeUtilizzoSEN, sortableListData) **pre**: preferenzaSorgente != null && percentualeUtilizzoPannelli != null && percentualeUtilizzoSEN != null && sortableListData != null
Post-condizione	/

Nome Metodo	+aggiornaPianoAttivo(String piano, int idAmministratore)
Descrizione	Aggiorna il piano attivo dei Parametri IA effettuato da un amministratore specifico.
Pre-condizione	**context**: IAService:: aggiornaPianoAttivo(piano, idAmministratore) **pre**: piano!= null && idAmministratore!= null
Post-condizione	/

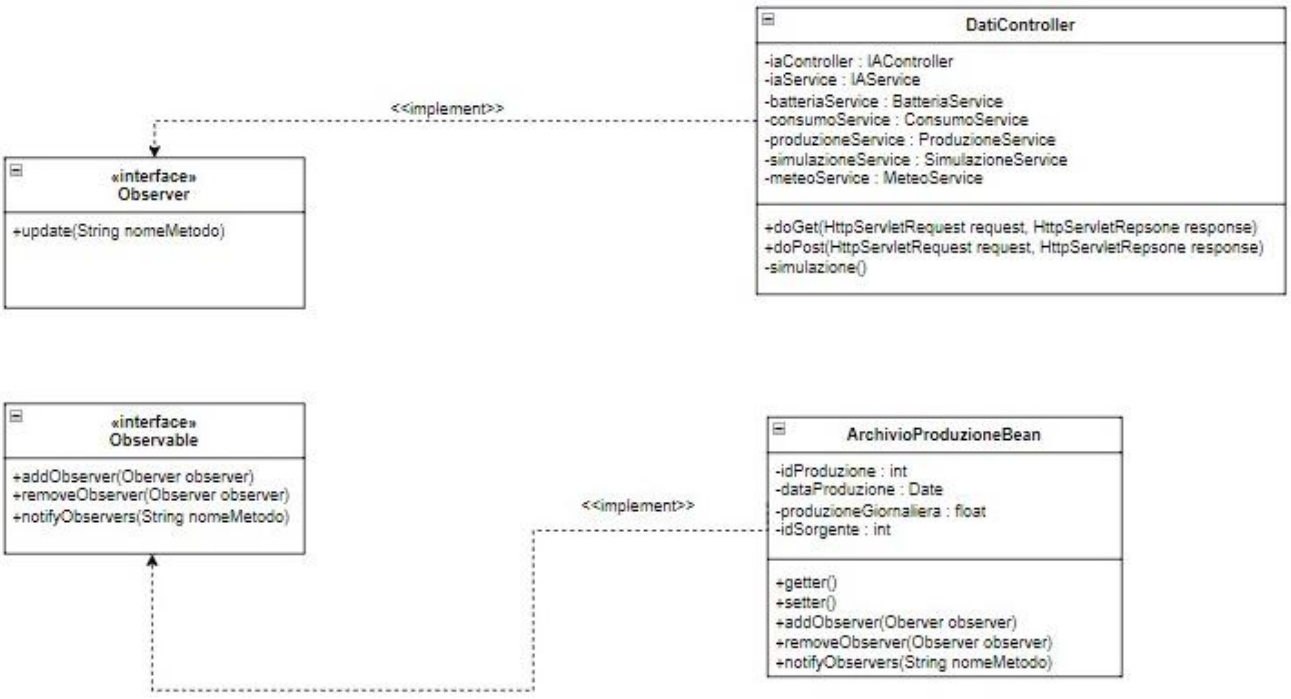
4. Design Pattern

Nella presente sezione si andranno a descrivere e dettagliare i design patterns utilizzati nello sviluppo dell'applicativo E-Balance. Per ogni pattern si darà:

- Una brevissima introduzione teorica.
- Il problema che doveva risolvere all'interno di E-Balance.
- Un grafico della struttura delle classi che implementano il pattern.

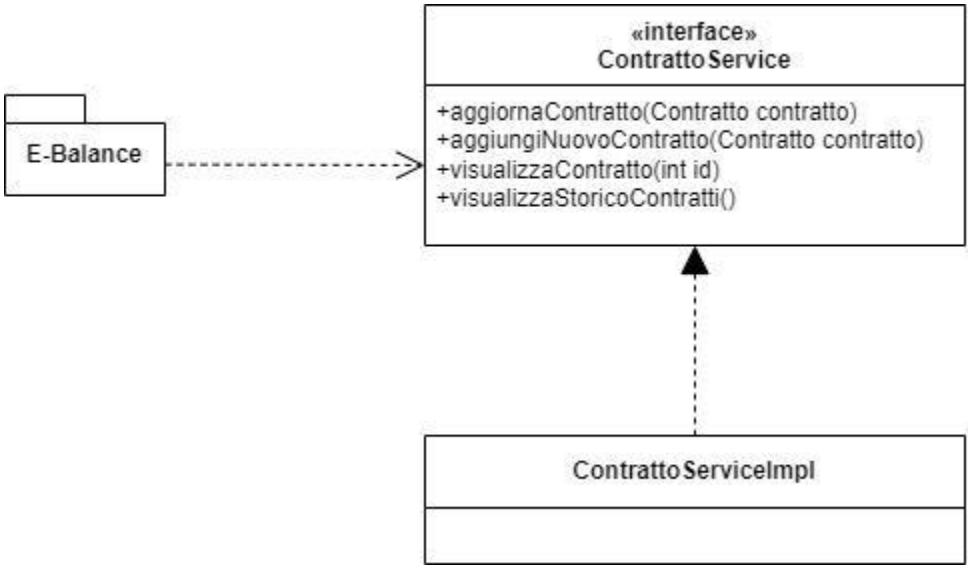
4.01 Observer

Il design pattern **Observer** è un pattern comportamentale che permette a un oggetto di mantenere una lista dei suoi "osservatori", che vengono informati automaticamente di qualsiasi cambiamento di stato del soggetto. Questo pattern è particolarmente utile quando un oggetto deve notificare una serie di altri oggetti su eventuali cambiamenti, senza conoscere i dettagli specifici degli osservatori. Nel nostro caso, ci troviamo di fronte ad un sistema in cui ci sono oggetti il cui stato può cambiare, e altri oggetti devono essere informati di tali cambiamenti senza essere direttamente accoppiati con gli oggetti che subiscono le modifiche.
Ad esempio, durante il monitoraggio delle condizioni meteorologiche, dove il soggetto è la stazione meteorologica che fornisce dati sulle condizioni atmosferiche mentre l'osservatore è il grafico in cui vengono aggiornate tali condizioni metereologiche. Ogni volta che la stazione meteorologica rileva un cambiamento nelle condizioni atmosferiche, notifica automaticamente l'osservatore.



4.02 Facade

Il design pattern **Facade** è un pattern strutturale che fornisce un'interfaccia unificata a un insieme di interfacce in un sistema. Questo pattern definisce un'interfaccia di alto livello che semplifica l'interazione con il sistema sottostante, rendendo più facile l'uso e la comprensione delle parti complesse del sistema. Questo approccio consente di mantenere una separazione tra l'implementazione interna, che potrebbe coinvolgere librerie, framework o insiemi di classi complessi, e l'interfaccia fornita all'esterno. Attraverso questa separazione, si ottiene un livello di disaccoppiamento elevato. Ciò significa che l'amministratore non ha bisogno di conoscere i dettagli specifici della complessa implementazione interna. Invece, può interagire con un'interfaccia semplificata e stabile, riducendo così la dipendenza da dettagli implementativi.
Questo disaccoppiamento offre manutenibilità poiché è possibile apportare modifiche all'implementazione interna senza influire sulle parti del sistema che utilizzano l'interfaccia esterna, aggiornabilità, poiché la possibilità di modificare l'implementazione interna senza modificare l'interfaccia esterna consente di aggiornare o migliorare le parti del sistema senza impattare gli utenti o le componenti che interagiscono con esso e, infine, flessibilità poiché l'utilizzo di un'interfaccia esterna permette una maggiore flessibilità nella scelta delle tecnologie o dei framework interni. Si possono adottare nuove librerie o apportare modifiche senza dover riscrivere l'interfaccia esterna.



5. Class Diagram

Per una maggiore leggibilità il Class Diagram è caricato sottoforma di pagina web statica HTML, raggiungibile a questo link.

6. Glossario

Sigla/Termine	Definizione
Controller	Classe che si occupa di gestire le richieste effettuate dal client.
Service	Classe che implementa la logica di business e viene utilizzata dal controller o da un altro sottosistema.
Model	Parte del design architetturale MVC che fornisce al sistema i metodi per accedere ai dati utili al sistema.
MVC	Model-View-Controller: design architetturale che permette di separare la logica di presentazione dalla logica di business alla base del sistema.
Observer Pattern	Design pattern usato come base per la gestione di eventi.
Farcade	Un oggetto che permette, attraverso un'interfaccia più semplice, l'accesso a sottosistemi che espongono interfacce complesse e molto diverse tra loro.