Università degli Studi di Salerno
Corso di Ingegneria del Software

UniClass
Object Design Document
Versione 1.1

Data: 17/12/2024

Coordinatore del progetto:

Nome	Matricola
Giuseppe Sabetta	0512117895

Partecipanti:

Nome	Matricola
Giuseppe Sabetta	0512117895
Sara Gallo	0512117262
Saverio D'Avanzo	0512118330
Gerardo Antonio Cetrulo	0512117856

Scritto da:	Giuseppe Sabetta (GS), Sara Gallo (SG), Saverio D'Avanzo (SD), Gerardo Antonio Cetrulo (AC)

Revision History

Data	Versione	Descrizione	Autore
17/12/2024	1.1	Correzione formattazione, cambio colori, miglioramento mapping, modifica interfacce	GS, SG, SD, AC
16/12/2024	1.0	Object Design Document	GS, SG, SD, AC

Sommario

1. Introduzione

1.1. Object Design trade-offs

Ci sono varie considerazioni da seguire, dunque alcuni compromessi da valutare per raggiungere l'ottimalità del sistema.
I trade off sono I seguenti:

- **Flessibilità vs** Stabilità: La flessibilità è fondamentale per un sistema in crescita come UniClass. Una piattaforma destinata a studenti universitari deve essere in grado di adattarsi a esigenze mutevoli, come nuove funzionalità (es. integrazione di calendari personalizzati) o modifiche strutturali (es. cambi di orario).
  Tuttavia, l'adozione della flessibilità comporta un rischio maggiore di introdurre bug, dato che modifiche frequenti possono destabilizzare il sistema.
  Per bilanciare i rischi, si implementa:
  - **Testing:** per rilevare tempestivamente i bug
  - **Feature toggles**: abilitare/disabilitare funzionalità in base alla maturità del codice
  - **Versioning delle API**: garantire retrocompatibilità durante l'aggiunta di nuove funzionalità
- Prestazioni **vs Manutenibilità**: La manutenibilità è essenziale per un progetto con una lunga prospettiva di utilizzo e aggiornamento, come UniClass. Questo tipo di piattaforma deve essere facile da aggiornare per incorporare nuove richieste senza compromettere l'intero sistema.
  La manutenibilità però potrebbe comportare una leggera perdita di prestazioni.
  Per garantire comunque delle buone prestazioni, si procede:
  - **Utilizzando il paradigma MVC**: per migliorare la chiarezza del codice, riducendo il rischio di errori durante la manutenzione
  - **Refactoring regolare**: ottimizzando parti del codice critico senza compromettere la struttura generale

- Costo **vs Scalabilità**: Il sistema UniClass parte con soluzioni economiche e, in caso di traffico intenso e grande affluenza al servizio, UniClass verrà trasferito su una piattaforma cloud.

- **Personalizzazione vs** Standardizzazione: Il sistema UniClass è una piattaforma completamente personalizzabile e ad-hoc (per quanto riguarda le funzionalità introdotte nel **RAD**), dato che la stessa piattaforma deve essere in grado di permettere una navigabilità studiata in base alle preferenze didattiche dell'utente. Personalizzazioni troppo avanzate, però, possono aumentare la complessità del sistema, rendendo più difficili i test e il debug. Per non rendere il sistema troppo complesso, si procederà per:
  - **Personalizzazione modulare**: offrire opzioni preconfigurate e scalabili per evitare configurazioni eccessivamente granulari
  - **Feature preview**: introdurre gradualmente nuove opzioni di personalizzazione per testarne l'efficacia
- Tempo di Esecuzione **vs Memoria**: Ottimizzare per la memoria è una scelta pragmatica per un sistema con grandi quantità di dati (es. orari, prenotazioni, profili). Una buona gestione della memoria riduce i costi operativi e migliora la stabilità complessiva. Tuttavia, tempi di esecuzione più lunghi potrebbero compromettere l'esperienza utente.
  Per rendere il sistema comunque efficiente si utilizzerà:
  - **Database ottimizzato**: utilizzando indici e partizionamento per migliorare le query senza impatti negativi sulla memoria

TRADE OFF	
Flessibilità	Stabilità
Manutentibilità	Prestazioni
Scalabilità	Costo
Personalizzazione	Standardizzazione
Memoria	Tempo di Esecuzione

1.2. Interface Document Guidelines

In questa sezione avremo un insieme di regole da utilizzare nella progettazione delle interfacce:

- Principio di Segregazione delle Interfacce: Assicurarsi che per il cliente non ci siano metodi non necessari. Ogni interfaccia deve

essere **specifica** e aderire a un solo scopo.

- Le trasformazioni devono essere effettuate in isolamento: (da modificare, per ogni trasformazione mettere un trattino)
- Gestione delle eccezioni:
    - Per ogni funzionalità, catturare le eccezioni previste o crearne di nuove coerenti con l'errore.
    - Stampare l'eccezione e importarla in un log file.
    - Evitare l'uso generico di *catch (Exception e)* senza gestione dettagliata.
- Validazione dati input: controllare valori nulli, formati errati o valori fuori range per metodi dove il dominio degli input è variabile/dinamico
- Principio di Liskov: Seguire un'ereditarietà rigorosa. Le sottoclassi devono essere sostituibili alle super-classi senza modificare il codice client. Nessun metodo deve violare le aspettative degli sviluppatori/client.
- Parentesi graffe {}: dopo l'inizio di un metodo e finire un rigo dopo l'ultima riga di codice
- Parametri nei metodi:
    - In presenza di più parametri per un metodo, inserire (parametro1, parametro2, ...), mettendo la virgola subito dopo il parametro e uno spazio per l'eventuale parametro successivo
    - Usare nomi generici durante la scrittura iniziale per semplificare la documentazione
- Commenti nei metodi:
    - Utilizzare // per spiegare istruzioni complesse o di difficile comprensione.
    - Aggiungere commenti multilinea /* */ all'inizio di ogni classe con una spiegazione dettagliata di scopo, responsabilità e autore e data dell'ultima modifica. Un esempio:

      /*
      * Classe: GestioneOrari
      * Scopo: Gestisce la creazione, modifica e visualizzazione degli orari
      * Autore Ultima Modifica: [Nome]
      * Data: [Data]
      */

- Gestione della spaziatura:
    - Aumentare la leggibilità con un'adeguata gestione degli spazi
    - Aggiungere una riga vuota tra metodi, blocchi logici e all'interno di costrutti lunghi
- Lunghezza delle righe: Limitare ogni riga a 80-120 caratteri per facilitare la lettura del codice, soprattutto su schermi piccoli o IDE con più finestre aperte.
- Metodi chiari e concisi: Ogni metodo deve avere una sola responsabilità. Non devono essere presenti metodi monolitici con più scopi.
- Nomi Descrittivi:
    - Usare nomi di metodi e classi auto-esplicativi, come *calcolaOrario()*, etc.
    - Evitare acronimi o abbreviazioni non comprensibili facilmente.
- Documentazione delle interfacce:
    - Ogni interfaccia deve avere una breve descrizione del suo scopo, dei suoi metodi e di cosa ritornano. Bisogna usare **Javadoc**.
    - Le interfacce non devono dipendere da classi concrete. Utilizzare tipi astratti o generici dove possibile.
    - Inserire solo costanti (final static) all'interno delle interfacce, evitando implementazioni dirette, per fini di sicurezza e usabilità.
- Indentazione Standard: usare una tabulazione uniforme per indentare il codice. Non sono permessi spazi per l'indentazione.
- Nomenclatura uniforme:
    - Classi e Interfacce: PascalCase
    - Metodi: camelCase
    - Costanti: UPPERCASE
- Import: importare solo le classi necessarie. Evitare gli `import .*`
- Gestione del dead code: non sono permesse implementazioni di metodi non utilizzati o non commentati

1.3. Design Patterns

Nello sviluppo del sistema **UniClass**, l'adozione dei **design patterns** si rivela essenziale per affrontare in maniera efficace i compromessi individuati durante la progettazione del sistema. I design patterns forniscono soluzioni collaudate a problemi ricorrenti, permettendo di

migliorare **manutenibilità**, **scalabilità** e **flessibilità** del codice, senza introdurre complessità eccessiva. In un contesto caratterizzato da esigenze contrastanti, come **flessibilità** vs stabilità o prestazioni vs **manutenibilità**, l'uso dei pattern garantisce una struttura chiara e robusta, facilitando l'integrazione di nuove funzionalità e la gestione dei cambiamenti. Attraverso approcci standardizzati e modulari, come l'implementazione del paradigma **MVC** e la personalizzazione modulare, possiamo mantenere un equilibrio ottimale tra personalizzazione e stabilità del sistema, migliorando al contempo l'esperienza di sviluppo e utilizzo della piattaforma.

Pattern Strutturali:

- **Facade**: Il Facade Pattern è un design pattern strutturale che fornisce un'interfaccia semplificata a un gruppo complesso di classi, librerie o moduli. L'obiettivo principale è nascondere la complessità del sistema sottostante, offrendo agli utenti (o agli altri moduli del sistema) un punto di accesso unificato e intuitivo.
  In pratica, il Facade Pattern funge da "facciata" che media tra un sistema complesso e il codice che lo utilizza, mantenendo il codice client più pulito e facile da leggere.
  UniClass prevede una serie di funzionalità che coinvolgono più moduli e sottosistemi (gestione orari, prenotazioni, messaggistica, profili, ecc.). Senza una Facade, il codice client (ad esempio il controller di un'interfaccia utente) dovrebbe interagire direttamente con ogni modulo, aumentando:
  - *La complessità del codice*.
  - *Le dipendenze tra i moduli*.
  - *La difficoltà nella manutenzione e nella gestione degli errori*.

  Utilizzare un facade pattern nella piattaforma porterebbe:
  - *Semplificazione dell'accesso alle funzionalità*
  - *Isolamento dei sottosistemi*
  - *Migliore manutenibilità*
  - *Supporto per la modularità*

Pattern Comportamentali:

- **Observer**: Definisce una dipendenza uno-a-molti tra oggetti,

in modo che quando uno cambia stato, tutti i suoi osservatori vengano notificati automaticamente. Conosciuto anche come Publish and Subscribe. Favorisce una comunicazione asincrona e reattiva, separando le classi coinvolte per migliorare modularità e flessibilità.
UniClass prevede diverse funzionalità che beneficiano di aggiornamenti dinamici e notifiche automatiche. L'Observer Pattern è ideale per gestire questi casi:

- *Sistema di notifiche (messaggistica broadcast): Un professore o un coordinatore invia un messaggio broadcast. Gli studenti registrati al corso vengono notificati automaticamente.*
- *Stato delle aule (libere/occupate): Cambiamenti dell'orario notificano le aule interessate in tempo reale.*
- *Le modifiche agli orari delle lezioni notificano l'agenda dello studente e gli orari settimanali.*

I vantaggi ricavati dall'utilizzo di questo pattern sono:

- *L'aggiornamento in tempo reale: gli studenti ricevono notifiche immediate senza dover controllare manualmente*
- *Modularità e flessibilità: Separazione tra il modulo che genera gli eventi (es. modifica degli orari) e i moduli che reagiscono agli eventi (notifiche agli studenti).*
- *Riduzione delle dipendenze: Il subject (es. modulo orari) non deve sapere chi sono i suoi observers, migliorando la manutenibilità.*

- **Strategy:** Lo Strategy Pattern consente di definire una famiglia di algoritmi, incapsulandoli separatamente per permettere di selezionarli dinamicamente.
  L'obiettivo principale è separare il comportamento (l'algoritmo) dal contesto in cui viene utilizzato.
  Invece di codificare un algoritmo direttamente all'interno di una classe, questa delega il comportamento a un oggetto di tipo "strategia". Ciò garantisce maggiore flessibilità e facilità di estensione.
  UniClass presenta molte funzionalità che richiedono la scelta dinamica di un algoritmo o comportamento basato su criteri specifici. Lo Strategy Pattern è ideale per gestire

questa complessità in modo flessibile.
In UniClass sarà utilizzato per:

- *Ordinamento degli orari e delle aule: Algoritmi per ordinare gli orari (per giorno, per corso, per aula). Ordinamento delle aule libere (per capienza, per distanza, per prossima disponibilità).*
- *Filtri personalizzati: Filtrare aule libere basandosi su criteri scelti dall'utente (capienza minima, attrezzature disponibili, ecc.).*
- *Gestione delle prenotazioni: Diversi algoritmi per allocare le aule in base a specifiche regole (priorità del corso, numero di studenti, ecc.).*

L'utilizzo di questo pattern porta a:

- *Flessibilità: Aggiungere nuovi criteri o algoritmi (es. un nuovo tipo di filtro per le aule) è semplice: basta creare una nuova strategia senza toccare il codice esistente.*
- *Manutenzione più semplice: Ogni algoritmo è incapsulato nella propria classe, rendendo il codice più leggibile e facile da modificare.*
- *Riduzione della complessità: Elimina la necessità di grandi blocchi condizionali (if/else o switch), migliorando la chiarezza del codice.*
- *Cambiamento dinamico del comportamento: Gli utenti possono modificare i criteri o i filtri durante l'uso della piattaforma, senza richiedere modifiche strutturali.*

- **State:** Lo State Pattern è un design pattern comportamentale che permette a un oggetto di modificare il proprio comportamento dinamicamente in base al suo stato interno. L'oggetto delega il comportamento specifico a una serie di classi che rappresentano ciascuno stato.
  Il comportamento dell'oggetto non è codificato staticamente, ma varia a seconda dello stato corrente, consentendo una maggiore modularità e flessibilità.
  UniClass prevede diverse funzionalità che potrebbero beneficiare della gestione dinamica degli stati. Lo State Pattern è ideale per implementare comportamenti che cambiano in base al contesto o alla situazione dell'utente o del sistema.
  Nella piattaforma sarà utilizzato per:

- *Gestione dello stato di autenticazione:*
  Stato non autenticato: Mostrare solo funzionalità limitate (es. login).
  Stato autenticato: Consentire l'accesso a funzionalità avanzate (prenotazioni, notifiche).
- *Prenotazioni di aule:*
  Stato libera: L'aula è disponibile per la prenotazione.
  Stato prenotata: L'aula è bloccata da un altro utente.
  Stato occupata: Aula già utilizzata per una lezione in corso.
- *Gestione degli appelli:*
  Stato aperto: Gli studenti possono iscriversi.
  Stato chiuso: L'appello non è più disponibile.

I benefici che avrà UniClass da ciò:

- *Eliminazione di condizioni complesse: Codice più pulito e leggibile, senza if/else annidati o switch.*
- *Modularità: Ogni stato ha la propria logica separata, migliorando la manutenibilità.*
- *Facilità di estensione: Aggiungere nuovi stati è semplice: basta creare una nuova classe senza modificare il codice esistente.*
- *Cambiamento dinamico del comportamento: Il sistema può adattarsi facilmente alle esigenze agli eventi del sistema.*

Pattern per la gestione della memoria

- **Flyweight:** Il Flyweight Pattern è un design pattern strutturale che consente di ridurre il consumo di memoria condividendo il più possibile dati comuni tra oggetti simili, invece di crearne nuove istanze per ciascun oggetto. L'idea principale è riutilizzare oggetti esistenti per rappresentare dati immutabili o condivisibili, risparmiando risorse.
  Si suddivide in due tipi di stato:
  1. Stato intrinseco:
  Dati che non cambiano e possono essere condivisi tra più oggetti.
  2. Stato estrinseco:
  Dati specifici dell'istanza, che non possono essere condivisi

e devono essere forniti dall'esterno.
La piattaforma UniClass potrebbe gestire una grande quantità di dati ripetuti, rendendo il Flyweight Pattern particolarmente utile per ottimizzare la memoria e migliorare le prestazioni.
Utilizzo in UniClass:

- *Gestione delle Aule: Le informazioni di base sulle aule (es. nome, posizione, capienza) possono essere condivise come stato intrinseco.*
  Dettagli specifici come orario di utilizzo o prenotazioni correnti possono essere forniti come stato estrinseco.
- *Visualizzazione degli orari: Ogni cella della tabella degli orari potrebbe rappresentare un Flyweight:*
  Stato intrinseco: Informazioni sulla lezione (es. corso, docente).
  Stato estrinseco: Dettagli dinamici come data e orario.
- *Gestione delle notifiche:*
  Le notifiche possono condividere dati comuni (es. tipo di messaggio, destinatario generico) e gestire separatamente gli stati unici (es. testo specifico o timestamp).
- *Cache per gli utenti autenticati:*
  Informazioni comuni agli utenti (es. tipo di ruolo, permessi di accesso) possono essere riutilizzate tramite Flyweight, mentre i dati specifici (es. nome, ID) sono gestiti separatamente.

Grazie all'utilizzo di questo pattern avremo:

- *Risparmio di memoria: Riduce l'allocazione di risorse duplicate, utile soprattutto se ci sono molte entità con dati ripetuti (es. aule o lezioni simili).*
- *Efficienza: Migliora le prestazioni condividendo oggetti già esistenti, riducendo il tempo di creazione e distruzione.*
- *Scalabilità: Permette alla piattaforma di gestire un numero elevato di utenti, dati o richieste senza aumentare drasticamente il consumo di memoria.*
- *Separazione dello stato: Permette alla piattaforma di gestire un numero elevato di utenti, dati o richieste senza aumentare drasticamente il consumo di*

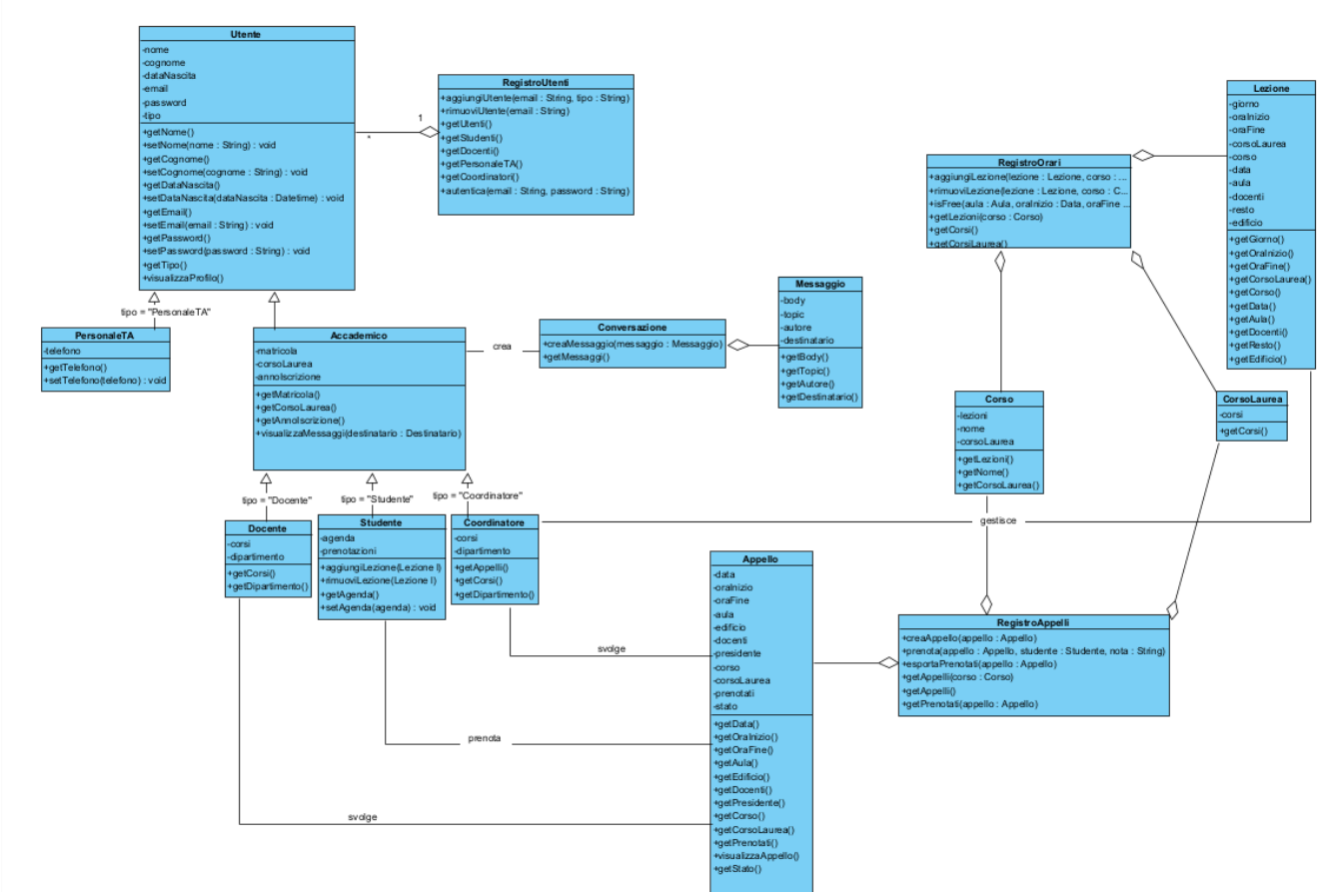
memoria.

1.4. Definition, acronyms and abbreviations

In questo documento, sono state utilizzate diverse abbreviazioni e termini che richiedono la specifica della definizione per aumentare la comprensione del documento.

Acronimo/Abbreviazione	Definizione
MVC	Paradigma di progettazione software che separa le componenti di Model, View e Control
Javadoc	Strumento di documentazione incluso nel JDK Java che genera html per descrizione delle specifiche di classi e metodi
JDK	Ambiente di sviluppo di applicazioni in linguaggio Java
DAO	Pattern progettuale per astrarre e incapsulare l'accesso ai dati di un'applicazione, fornendo interfacce per operazioni CRUD
CRUD	Rappresentazione delle quattro operazioni fondamentali sui Database, Create, Retrieve, Update, Delete

1.5. Mapping a Oggetti



1.6. References

- RequirementsAnalysisDocument_UniClass.pdf
- SystemDesignDocument_UniClass.pdf
- Object Oriented Software Engineering using UML, Patterns and Java Third Edition – Bruegge, Dutoit

2. Packages

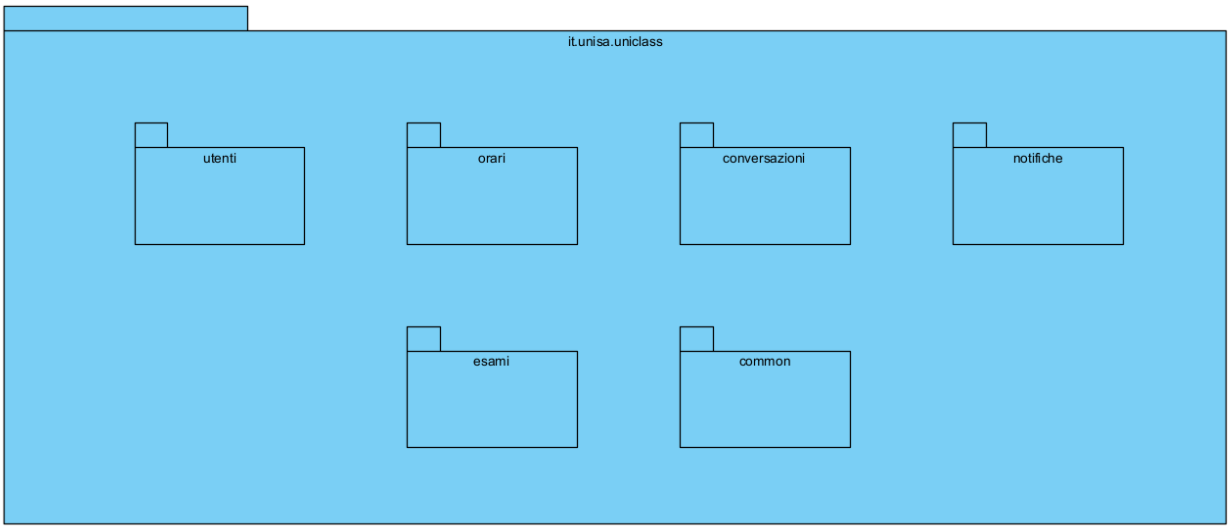
Il sistema UniClass presenta un packaging basato sul layering e partitioning visibile dall'SDD, seguendo il paradigma MVC, dividendo ogni sottosistema in model, view e control. Il model rappresenta l'oggetto in sé, i services rappresentano i pattern progettuali DAO per la comunicazione tra i model e le repository persistenti e il controller per l'HTTPServlet. I sottosistemi sono orari, utenti, conversazioni, notifiche, esami, mentre un ultimo package importante sarà

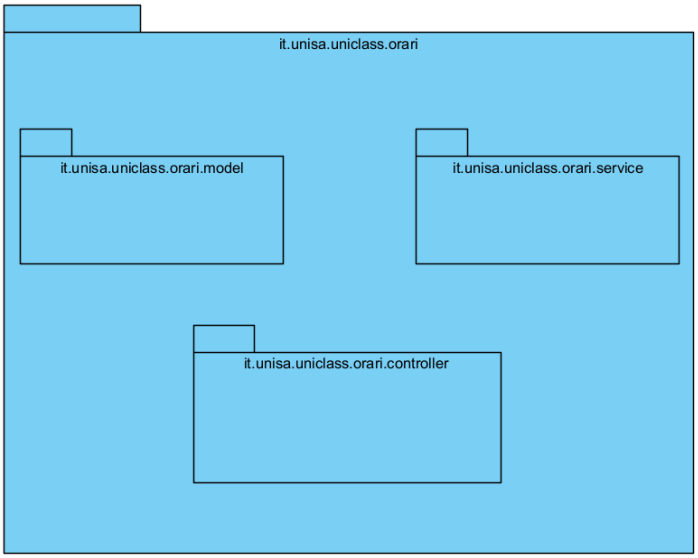
common, contenente utility (come design patterns o funzioni ausiliarie), exception (eccezioni ad-hoc), config (file di configurazione dell'ambiente), security (per controlli e filtri presenti sull'ambiente) e controller, per servlet comuni.

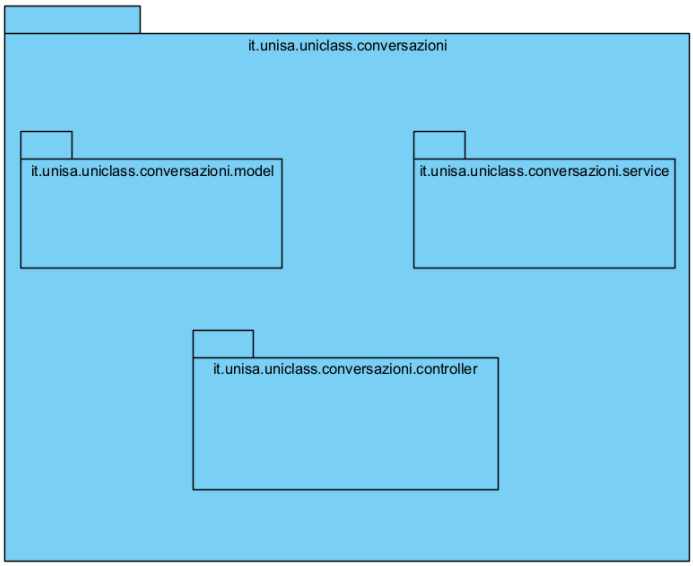
Packages:

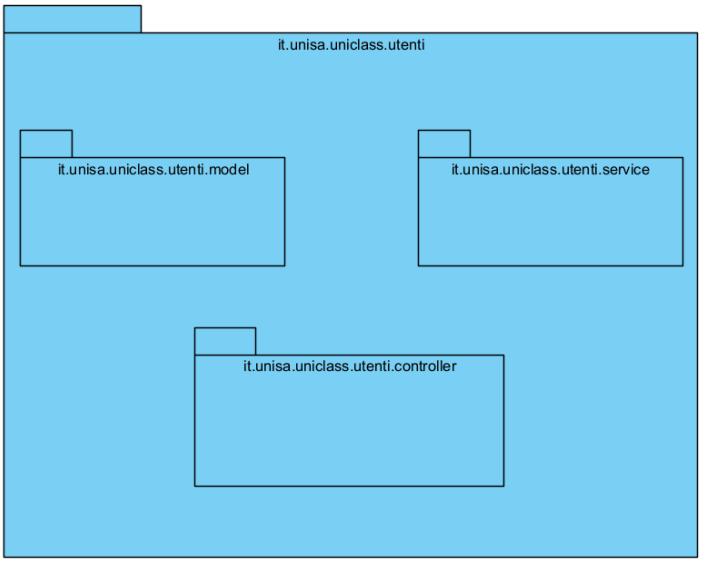
```
it.unisa.uniclass
    ├── orari
    │   ├── model
    │   ├── service
    │   └── controller
    ├── utenti
    │   ├── model
    │   ├── service
    │   └── controller
    ├── conversazioni
    │   ├── model
    │   ├── service
    │   └── controller
    ├── notifiche
    │   ├── model
    │   ├── service
    │   └── controller
    ├── esami
    │   ├── model
    │   ├── service
    │   └── controller
    └── common
        ├── utils
        ├── exceptions
        ├── config
        └── security
```

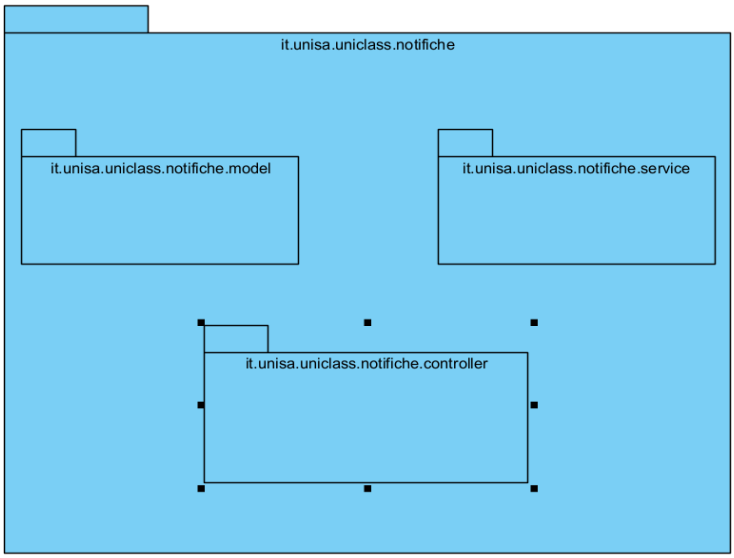
Package it.unisa.uniclass:

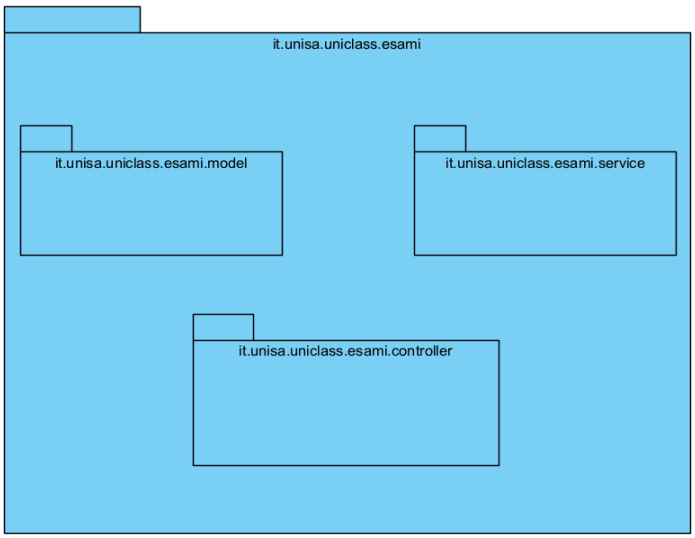


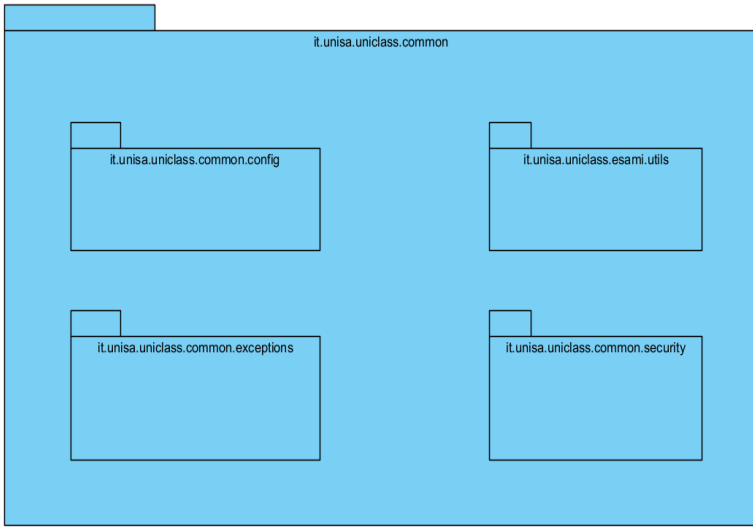

it.unisa.uniclass.orari
it.unisa.uniclass.orari.model
it.unisa.uniclass.orari.service
it.unisa.uniclass.orari.controller


it.unisa.uniclass.conversazioni
it.unisa.uniclass.conversazioni.model
it.unisa.uniclass.conversazioni.service
it.unisa.uniclass.conversazioni.controller


it.unisa.uniclass.utenti
it.unisa.uniclass.utenti.model
it.unisa.uniclass.utenti.service
it.unisa.uniclass.utenti.controller


it.unisa.uniclass.notifiche
it.unisa.uniclass.notifiche.model
it.unisa.uniclass.notifiche.service
it.unisa.uniclass.notifiche.controller


it.unisa.uniclass.esami
it.unisa.uniclass.esami.model
it.unisa.uniclass.esami.service
it.unisa.uniclass.esami.controller


it.unisa.uniclass.common
it.unisa.uniclass.common.config
it.unisa.uniclass.esami.utils
it.unisa.uniclass.common.exceptions
it.unisa.uniclass.common.security

3. Class Interfaces Glossary

Javadoc è uno strumento di documentazione fornito con il JDK (Java Development Kit) che consente di generare documentazione in formato HTML direttamente dal codice sorgente Java. Funziona analizzando i commenti strutturati (detti anche "doc comments") inseriti sopra classi, metodi, campi e costruttori.
Questi commenti sono delimitati da /** ... */ e seguono una sintassi specifica che include tag come @param, @return, @throws, etc.
Per ogni package sarà possibile avere un'**interfaccia** con i metodi presenti nel servizio in questione.

3.1. Package Gestione Orari

Nome Interfaccia	**GestioneOrariInterface**
Scopo	**Interfaccia relativa al servizio della gestione degli orari in UniClass, per l'aggiunta, rimozione e modifica di una lezione all'interno di un corso, aggiunta o rimozione di corsi da seguire in un'agenda**
Invariante di classe	**context Studente inv:** **self.getAgenda()->forAll(l1,l2 | l1.getOraFine() <= l2.getOraInizio() or l2.getOraFine() <= l1.getOraInizio())** **context CorsoLaurea inv:** **self.getCorsi()->collect(c | c.getLezioni() -> asSet() -> forAll(l1,l2 | l1<>l2 or** **l1.getOraFine() <= l2.getOraInizio() or l2.getOraFine() <= l1.getOraInizio())**
Nome Metodo	+ aggiungiLezione(Lezione l, CorsoLaurea cl, Anno a, Resto r): int
Descrizione Metodo	- Il metodo permette di aggiungere una lezione all'interno di un corso, purchè non vada in conflitto con le lezioni già presenti del corso. - È possibile non utilizzare il parametro resto in caso di corsi con abbastanza partecipanti in una sola sezione.
Pre-condizioni	context GestioneOrariInterface::aggiungiLezione(Lezione l, CorsoLaurea cl, Anno a, Resto r) : int pre : l <> null pre : cl <> null pre : a <> null

Post-condizioni	context GestioneOrariInterface::aggiungiLezione(Lezione l, CorsoLaurea cl, Anno a, Resto r) : int post : cl.getLezioni() = @pre.cl.getLezioni() + 1 post : cl.getLezioni().include(l)
Nome Metodo	+ rimuoviLezione(Lezione l, CorsoLaurea cl, Anno a, Resto r) : void
Descrizione Metodo	- Il metodo permette di rimuovere una lezione all'interno di un corso. - È possibile non utilizzare il parametro resto in caso di corsi con abbastanza partecipanti in una sola sezione.
Pre-condizioni	context GestioneOrariInterface::rimuoviLezione(Lezione l, CorsoLaurea cl, Anno a, Resto r) : int pre : l <> null pre : cl <> null pre : a <> null
Post-condizioni	context GestioneOrariInterface::rimuoviLezione(Lezione l, CorsoLaurea cl, Anno a, Resto r) : int post : cl.getLezioni() = @pre.cl.getLezioni() - 1 post : cl.getLezioni().include(l)
Nome Metodo	+ isFree(aula : Aula, oraInizio : Data, oraFine : Data) : boolean
Descrizione Metodo	Questo metodo restituisce True se l'aula è libera nell'intorno previsto dai parametri, False altrimenti
Pre-condizioni	context RegistroOrari::isFree(aula : Aula, oraInizio : Date, oraFine : Date) : boolean pre: aula <> null pre: oraInizio <> null and 8<=oraInizio<=18:30 pre: oraFine <> null and 8:30<=oraFine<=19:30
Post-condizioni	context RegistroOrari::isFree(aula : Aula, oraInizio : Date, oraFine : Date) : boolean post: result = RegistroOrari.getLezioni() -> forAll(l | l.aula <> aula or l.oraFine<=oraInizio or l.oraInizio >= oraFine)
Nome Metodo	+ getLezioni(corso : Corso) : Collection<Lezione>
Descrizione Metodo	Questo metodo restituisce l'elenco delle lezioni presenti in un determinato corso
Pre-condizioni	context RegistroOrari::getLezioni(corso : Corso) : Collection<Lezione>

	pre: corso <> null
Post-condizioni	context RegistroOrari::getLezioni(corso : Corso) : Collection<Lezione> post: result = RegistroOrari.getLezioni()->select(l | l.getCorso() = corso)
Nome Metodo	+ getCorsi(corsoLaurea : CorsoLaurea) : Collection<Corso>
Descrizione Metodo	Questo metodo restituisce una collezione con tutti i corsi presenti in un corso di Laurea
Pre-condizioni	context RegistroOrari::getCorsi(corsoLaurea : CorsoLaurea) : Collection<Corso> pre: corsoLaurea <> null
Post-condizioni	context RegistroOrari::getCorsi(corsoLaurea : CorsoLaurea) : Collection<Corso> post: result = RegistroOrari.getCorsi() -> select(c | c.getCorsoLaurea() = corsoLaurea)
Nome Metodo	+ getCorsiLaurea() : Collection<CorsoLaurea>
Descrizione Metodo	Questo metodo restituisce una collezione con tutti i corsi di laurea presenti in ateneo
Pre-condizioni	context RegistroOrari::getCorsiLaurea() : Collection<CorsoLaurea> pre:
Post-condizioni	context RegistroOrari::getCorsiLaurea() : Collection<CorsoLaurea> post: result = RegistroOrari.getCorsiLaurea() -> asSet()

3.2 Package Gestione Utenti

Nome Interfaccia	**GestioneUtentiInterface**
Scopo	**Interfaccia relativa al servizio della gestione degli utenti in UniClass, per l'aggiunta e rimozione**
Invariante di classe	**//**
Nome Metodo	+ aggiungiUtente(email : String, tipo : String) :
Descrizione Metodo	Questo metodo aggiunge un nuovo utente al registro, specificando la sua email e il tipo dell'utente.
Pre-condizioni	context RegistroUtenti::aggiungiUtente(email : String, tipo : String) pre: email <> null and email.size() > 0

	pre: tipo <> null and tipo.size() > 0 pre: self.getUtenti()->forAll(u | u.getEmail() <> email)
Post-condizioni	context RegistroUtenti::aggiungiUtente(email : String, tipo : String) post: self.getUtenti()->exists(u | u.getEmail = email and u.getTipo = tipo) post: self.getUtenti().size() = @pre.self.getUtenti().size() + 1
Nome Metodo	+ rimuoviUtente(email : String)
Descrizione Metodo	Questo metodo rimuove un utente dal registro degli utenti con l'email indicata nel parametro
Pre-condizioni	context RegistroUtenti::rimuoviUtente(email : String) pre: email<>null and email.size() > 0 pre: self.getUtenti()->exists(u | u.getEmail() = email)
Post-condizioni	context RegistroUtenti::rimuoviUtente(email : String) post: self.getUtenti()->forAll(u | u.getEmail() <> email) post: self.getUtenti().size() = @pre.self.getUtenti().size() - 1
Nome Metodo	+ getUtenti() : Collection<Utente>
Descrizione Metodo	Questo metodo restituisce una collezione di tutti gli utenti presenti nel sistema.
Pre-condizioni	context RegistroUtenti::getUtenti() pre:
Post-condizioni	context RegistroUtenti::getUtenti() post: result = self.utenti->asSet()
Nome Metodo	+ getStudenti() : Collection<Studente>
Descrizione Metodo	Questo metodo restituisce una collezione di tutti gli utenti di tipo "Studente" presenti nel sistema.
Pre-condizioni	context RegistroUtenti::getStudenti() pre:
Post-condizioni	context RegistroUtenti::getStudenti() post: result = self.getUtenti() -> select(u | u.getTipo() = "Studente")
Nome Metodo	+ getDocenti() : Collection<Docente>
Descrizione Metodo	Questo metodo restituisce una collezione di tutti gli utenti di tipo "Docente" presenti nel sistema.
Pre-condizioni	context RegistroUtenti::getDocenti() pre:
Post-condizioni	context RegistroUtenti::getDocenti() post: result = self.getUtenti() -> select(u | u.getTipo()

	= “Docente”)
Nome Metodo	+ getCoordinatori() : Collection<Coordinatore>
Descrizione Metodo	Questo metodo restituisce una collezione di tutti gli utenti di tipo “Coordinatore” presenti all’interno del sistema.
Pre-condizioni	context RegistroUtenti::getCoordinatori() pre:
Post-condizioni	context RegistroUtenti::getCoordinatori() post: result = self.getUtenti() -> select(u | u.getTipo() = “Coordinatore”)
Nome Metodo	+ getPersonaleTA() : Collection<PersonaleTA>
Descrizione Metodo	Questo metodo restituisce una collezione di tutti gli utenti di tipo “PersonaleTA”, ovvero Personale Tecnico Amministrativo, presenti all’interno del sistema.
Pre-condizioni	context RegistroUtenti::getPersonaleTA() pre:
Post-condizioni	context RegistroUtenti::getPersonaleTA() post: result = self.getUtenti() -> select(u | u.getTipo() = “PersonaleTA”)

3.3 Package Gestione Appelli

Nome Interfaccia	**GestioneAppelliInterface**
Scopo	**Interfaccia relativa al servizio della gestione degli appelli in UniClass, per l’aggiunta, rimozione e prenotazioni**
Invariante di classe	
Nome Metodo	+ creaAppello(appello : Appello) : Appello
Descrizione Metodo	Questo metodo consente la creazione di un appello all’interno del sistema
Pre-condizioni	context RegistroAppelli::creaAppello(appello : Appello) pre: appello <> null
Post-condizioni	context RegistroAppelli::creaAppello(appello : Appello) post: self.getAppelli().size() = @pre.self.getAppelli().size() + 1 post: self.getAppelli() -> exists(a | a = appello)

Nome Metodo	+ prenota(appello : Appello, studente : Studente, nota : String)
Descrizione Metodo	Questo metodo consente la prenotazione lato studente ad un determinato appello con una nota facoltativa.
Pre-condizioni	context RegistroAppelli::prenota(appello : Appello, studente : Studente, nota : String) pre: appello <> null pre: studente <> null and RegistroStudenti::getStudenti() -> exists(s | s = studente) pre: self.getAppelli() -> exists(a | a = appello) pre: a.getStato() = “Aperto” pre: not appello.getPrenotati()->includes(studente)
Post-condizioni	context RegistroAppelli::prenota(appello : Appello, studente : Studente, nota : String) post: appello.getPrenotati()->includes(studente) post: appello.getPrenotati().size() = @pre.appello.getPrenotati().size() + 1
Nome Metodo	+ esportaPrenotati(appello : Appello) : csv
Descrizione Metodo	Questo metodo consente l’esportazione in un file excel dei prenotati di un determinato appello.
Pre-condizioni	context RegistroAppelli::esportaPrenotati(appello : Appello) pre: appello <> null and self.getAppelli() -> includes(appello)
Post-condizioni	Post: result = csv
Nome Metodo	+ getAppelli(corso : Corso) : Collection<Appello>
Descrizione Metodo	Questo metodo consente di restituire una collezione di tutti gli appelli presenti in un determinato corso
Pre-condizioni	context RegistroAppelli::getAppelli(corso : Corso) : Collection<Appello> pre: corso <> null and RegistroOrari::getCorsi() -> includes(corso)
Post-condizioni	context RegistroAppelli::getAppelli(corso : Corso) : Collection<Appello> post: result = self.getAppelli() -> select(a | a.getCorso() = corso)
Nome Metodo	+ getAppelli() : Collection<Appello>
Descrizione Metodo	Questo metodo consente di restituire una collezione di tutti gli appelli di tutti i corsi presenti in Ateneo.

Pre-condizioni	context RegistroAppelli::getAppelli() : Collection<Appello> pre:
Post-condizioni	context RegistroAppelli::getAppelli() : Collection<Appello> post: result = self.appelli