

Relazione finale del progetto di

Informatica III

Modulo di Progettazione e algoritmi

Prof. Patrizia Scandurra

cod. 38068

Componenti del gruppo:

Alessandro Mazzola, matr. 1053121

Luca Parimbelli, matr. 1053142

Andrea Marinò, matr. 1053230

Luca Ghislotti, matr. 1052975

Laurea Magistrale in Ingegneria Informatica

Università degli Studi di Bergamo

A.A. 2020/2021

I Anno

Contents

| | |
|---|-----------|
| Introduzione al progetto | 4 |
| Tecnologie e metodi | 6 |
| 1 ITERAZIONE 0 | 9 |
| 1.1 Requirements engineering | 10 |
| 1.2 Analisi dei requisiti | 11 |
| 1.3 Use case stories | 11 |
| 1.3.1 Use case stories: Personale universitario | 12 |
| 1.3.2 Use case stories: Ristoratore | 15 |
| 1.3.3 Use case stories: Amministratore | 15 |
| 1.3.4 Use case stories: Sistema di Gestione tessera | 16 |
| 1.3.5 Use case stories: Sistema di Autenticazione | 17 |
| 1.4 Use case diagram | 18 |
| 1.4.1 Fully dressed description | 20 |
| 1.5 Analisi delle specifiche | 33 |
| 1.5.1 Introduzione alle specifiche | 33 |
| 1.5.2 Specifiche funzionali | 33 |
| 1.6 Analisi dell'architettura | 35 |
| 1.6.1 Deployment diagram - informal | 35 |
| 1.6.2 Deployment diagram - UML | 37 |
| 2 ITERAZIONE 1 | 38 |
| 2.1 Component diagram - single component | 39 |
| 2.2 Component diagram delle interfacce | 40 |

| | | |
|----------|---|-----------|
| 2.3 | Interface definition | 40 |
| 2.4 | Data class diagram | 40 |
| 3 | ITERAZIONE 2 | 44 |
| 3.1 | Implementazione | 45 |
| 3.2 | Selezione funzioni implementate | 45 |
| 3.3 | Component diagram White-Box | 46 |
| 3.4 | Sequence diagram per visualizzazione | 48 |
| 3.5 | Class diagram dei metodi per visualizzazione | 49 |
| 3.6 | Tecnologie utilizzate | 50 |
| 3.6.1 | MongoDB Atlas | 50 |
| 3.6.2 | Robo 3T | 50 |
| 3.6.3 | Java | 50 |
| 3.7 | MongoDB: API di connessione | 51 |
| 3.8 | Pseudocodice metodo: <code>GetCanteenETA()</code> | 51 |
| 3.8.1 | Analisi di complessità metodo: <code>GetCanteenETA()</code> | 52 |
| 3.9 | Testing API di visualizzazione | 53 |
| 3.10 | Casi di test | 56 |
| 3.11 | Analisi statica del codice (STAN4J) | 57 |
| 3.11.1 | Legame tra Abstractness e Instability | 57 |
| 3.11.2 | Map View | 58 |
| 3.11.3 | Composition View | 59 |
| 3.12 | Conclusioni iterazione 2 | 60 |
| 4 | ITERAZIONE 3 | 62 |
| 4.1 | Selezione funzioni implementate | 63 |

| | | |
|----------|--|-----------|
| 4.2 | Class diagram dei metodi per inserimento | 63 |
| 4.3 | Component diagram White-Box | 64 |
| 4.4 | Sequence diagram per inserimento | 65 |
| 4.5 | Casi di test | 66 |
| 4.6 | Conclusioni iterazione 3 | 68 |
| 5 | ITERAZIONE FINALE | 69 |
| 5.1 | Interfacce grafiche | 70 |
| 5.2 | Conclusioni finali | 73 |
| 5.2.1 | API esposte | 73 |
| 5.2.2 | Repository GitHub | 74 |
| 5.2.3 | Sviluppi futuri | 74 |
| 5.3 | Manuale utente | 75 |
| 6 | STAN4J | 76 |

Introduzione al progetto

Il progetto nasce dalla rielaborazione di una traccia di un esame di stato che richiede "la realizzazione di un sistema per la gestione del servizio mensa di un grande campus" (Fig. 1). La traccia è stata adeguatamente estesa e rielaborata in modo da poter realizzare un sistema informativo più flessibile. In particolare, si intende realizzare un sistema non specifico che, dopo un'iniziale fase di configurazione, può essere utilizzato all'interno di una qualunque organizzazione con lo scopo di gestire le (multiple) mense dei propri dipendenti. Le funzionalità che il sistema informativo dovrà implementare sono descritte tramite lo "*Use case diagram*" presentato nei prossimi paragrafi.

Uno degli obiettivi fondamentali di questo progetto è quello di sviluppare il software in questione utilizzando una metodologia agile: tramite l'utilizzo di questa metodologia si abbattano i costi e i tempi di sviluppo oltre che a migliorare la qualità del prodotto finale. Nel paragrafo successivo verrà spiegato con cura in che cosa consiste questo metodo e quali sono i benefici.

Parole chiave

- Sistema informatico distribuito
- Metodologia agile
- UML Diagrams
- API REST

TRACCIA N. 1

Si progetti un sistema informativo per la gestione del servizio mensa di un grande campus universitario. Il campus dispone di diverse mense dove gli studenti possono recarsi a colazione, a pranzo, e a cena. Non tutte le mense sono aperte a cena e ognuna ha un proprio giorno di chiusura. Ogni mensa ha orari diversi, un certo numero di posti a sedere, un numero massimo di coperti per pasto. Ogni mensa ha un giorno della settimana in cui i pasti costano 25% in meno. Alcune mense servono pasti specifici per vegetariani, della cucina internazionale e della cucina mediterranea.

Ogni studente è dotato di una tessera mensa, e di una carta a punti prepagata per comprare i pasti. Ogni mensa richiede una "quantità minima di punti" (per l'acquisto del pasto più economico) diversa. Uno studente può mangiare ad una certa mensa se la sua carta ha punti a sufficienza. L'ingresso è vietato anche quando una mensa è completa: il numero di persone nella mensa è pari al numero di posti a sedere. L'addetto deve avvisare lo studente ogni volta che i punti residui sulla tessera non sono più sufficienti a consumare un pasto, neppure il più economico. Le tessere possono essere ricaricate. Il minimo sono 50 punti, il massimo 1000 punti.

Il sistema deve essere in grado di monitorare l'occupazione delle mense in "tempo reale". A tale scopo dei monitor informativi dell'università devono poter guidare gli studenti indicando le mense meno affollate e visualizzando altre informazioni utili sulle mense, quali, ad esempio, le mense più convenienti perché applicano per quel giorno lo sconto sui pasti, le mense ancora aperte, la "quantità minima di punti", ecc. Nell'accezione più ampia, le informazioni sulle mense devono poter essere visualizzate anche da dispositivi mobili, quindi viene definita un'interfaccia tipo web service per accedervi. Alla fine della giornata il sistema deve anche produrre un consuntivo sull'uso delle diverse mense al fine di consentire il monitoraggio e il miglioramento della qualità del servizio.

Fig. 1: Traccia di riferimento per lo sviluppo del progetto

Tecnologie e metodi

Modello AGILE

Come anticipato prima, l'utilizzo della metodologia agile è stata di fondamentale importanza per lo sviluppo del software. Infatti, con questo metodo siamo riusciti a gestire variabili (come: tempo, risorse e qualità) che con altre metodologie non saremmo riusciti a controllare. Nasce come bisogno di contrapporsi con i modelli tradizionali come il modello a spirale e il modello a cascata. Questa nuova metodologia è un modo di pensare il software in modo leggero e ci permette di raggiungere l'agilità con l'utilizzo di best-practice. I quattro valori agili importanti sono:

- Persone e interazioni sono più importanti del processo e dei tools utilizzati.
- Un software funzionante è più importante di una chiara documentazione.
- La collaborazione del cliente va al di là del contratto stipulato.
- La capacità di rispondere a cambiamenti è una caratteristica fondamentale. Può succedere spesso che vengano imposte delle modifiche al prodotto: bisogna adottare un processo in grado di apportare continui cambiamenti a ciò che stiamo realizzando.

CoCoME

Per lo sviluppo di questo progetto abbiamo seguito come riferimento le tecniche e i modelli visti a lezione con il caso di studio CoCoME. A differenza di CoCoME, con il consiglio della prof. Scandurra, abbiamo introdotto l'utilizzo delle API REST al posto di JMS (Java Messaging System).

Toolchain

Per la realizzazione del software verranno utilizzati i seguenti strumenti:

- Modellazione:
 - **Use case diagram, deployment diagram, component diagram, class diagram, sequence diagram, datatype datagram:** Astah UML, strumento open source con licenza studente, molto utile e veloce per comporre diagrammi.
- Implementazione software:
 - **Linguaggio di programmazione:** Java, linguaggio di programmazione.
 - **IDE:** Eclipse, un ambiente di sviluppo integrato per il linguaggio java.
 - **API Development:** Eclipse e Postman
 - **DBMS:** MongoDB Atlas
 - **Interfaccia grafica:** Window Builder 1.9.4 di Java Swing
- Analisi del software:
 - **Analisi statica:** Stan4J
 - **Analisi dinamica:** JUnit
- Documentazione, versioning e organizzazione team:
 - **Documentazione:** LaTex, con scrittura tramite l'editor online Overleaf.
 - **Versioning:** GitHub, servizio di hosting per progetti software.
 - **Materiale:** Google Drive, servizio file sharing collaborativo.

- **Riunioni team:** Skype, servizio di messaggistica istantanea e videochiamate.

1 ITERAZIONE 0

1.1 Requirements engineering

In questa fase si è svolta l'analisi dei requisiti partendo dalle Use Case Stories: questa metodologia di analisi dei casi d'uso ci ha permesso di scandire i possibili scenari all'interno del problema. Il team ha svolto questa fase attraverso una sessione di brainstorming i cui risultati sono visibili in questo paragrafo. Da ricordare che, per definizione di processo agile, le Use Case Stories descritte sono state utilizzate come input per le fasi successive dove sono state analizzate e raffinate con cura.

1.2 Analisi dei requisiti

Per effettuare l'analisi dei requisiti è stato necessario immedesimarsi nei panni di un potenziale cliente per poter prendere in considerazione tutti i possibili scenari e tutti i requisiti congeniali alla realizzazione del nostro progetto.

Per fare ciò siamo ricorsi all'uso delle Use Case Stories, tecnica molto efficiente e largamente utilizzata nell'ambito del software design, grazie alla quale siamo riusciti a focalizzare volta per volta la nostra attenzione su un attore diverso coinvolto nel sistema, di modo da poter comprendere a fondo le sue necessità in termini di funzionalità dell'applicativo da progettare.

Per prima cosa però, in via del tutto preliminare, è necessario sottolineare una precondizione indispensabile ai fini dell'utilizzo del software stesso: ogni utente del sistema mense del campus, al momento della sua iscrizione al servizio, riceve una tessera fisico-virtuale senza la quale non avrebbe la possibilità di pagare i pasti che desidera consumare. Infatti, il sistema progettato, prevede l'utilizzo di un metodo di pagamento interno basato su punti che vengono debitamente convertiti dal denaro grazie ad un ente bancario esterno che ne gestisce tutti gli aspetti.

1.3 Use case stories

L'elaborazione delle use case stories ha richiesto due fasi distinte. In una prima fase, è stata stilata una lista divisa per attori delle funzionalità richieste da ciascuno di essi. In una seconda fase, le use case stories sono state raggruppate per ogni attore in "macro-gruppi". Questo è stato necessario per poter identificare i diversi use case da inserire successivamente nello use case diagram.

Gli attori (primari e secondari) coinvolti nel sistema sono:

- personale universitario (primario)

- amministratore (primario)
- ristoratore (primario)
- sistema di gestione tessera (secondario)
- sistema di autenticazione (secondario)

1.3.1 Use case stories: Personale universitario

L'attore primario coinvolto nel sistema è stato da noi identificato come tutto il personale universitario a cui sono rivolte le main functionalities del sistema progettato.

Di seguito sono elencate le Use Case Stories a lui dedicate:

VISUALIZZAZIONE INFORMAZIONI MENSE

- Come personale universitario, voglio poter visualizzare le mense aperte (colazione/pranzo/cena) su dashboards sparse in università e su app mobile
- Come personale universitario, voglio poter visualizzare il menu del giorno nella mensa selezionata
- Come personale universitario, voglio poter visualizzare le offerte giornaliere (sconti)
- Come personale universitario, voglio poter visualizzare la disponibilità di posti in real time
- Come personale universitario, voglio poter visualizzare un estimated time per la prossima disponibilità se pieno
- Come personale universitario, voglio poter visualizzare il tempo di attesa media (coda) per l'acquisto di un pasto

- Come personale universitario, voglio poter visualizzare se un piatto non è più disponibile
- Come personale universitario, voglio poter visualizzare i prezzi dei vari piatti e possibili offerte “combo menu”
- Come personale universitario, voglio poter visualizzare le mense divise per categorie (vegetariane/internazionale/mediterranea etc.)

VISUALIZZAZIONE TESSERA

- Come personale universitario, voglio poter visualizzare il saldo residuo su tessera
- Come personale universitario, voglio poter bloccare una tessera in caso di smarrimento

GESTIONE DASHBOARD PERSONALI

- Come personale universitario, voglio poter gestire una lista di mense preferite
- Come personale universitario, voglio poter filtrare le mense sulla base di preferenze circa il cibo offerto (es. vegetariano, calorie medie)
- Come personale universitario, voglio poter visualizzare “i miei ordini passati”
- Come personale universitario, voglio poter creare una wishlist di cibi che vorrei provare e essere notificato della loro disponibilità/giornata in cui c’è un’offerta per quei cibi
- Come personale universitario, voglio poter riorganizzare la dashboard su app delle mense giornaliere in base a criteri personali (es. ordine crescente di prezzo etc.)
- Come personale universitario, voglio poter visualizzare grafici circa i miei consumi

ACQUISTA PASTO

- Come personale universitario, voglio poter pagare un pasto tramite tessera
- Come personale universitario, voglio poter essere notificato in caso di punti minori della soglia minima

ENTRATA/USCITA DALLA MENSA

- Come personale universitario, voglio poter entrare/uscire dalla mensa passando la tessera a un tornello

LOG-IN/LOG-OUT

- Come personale universitario, voglio poter fare il log-in/log-out al mio account

RICARICA PUNTI TESSERA

- Come personale universitario, voglio poter ricaricare i punti sulla tessera in presenza e online

GESTIONE PROFILO UTENTE

- Come personale universitario, voglio poter modificare i setting relativi alle notifiche
- Come personale universitario, voglio poter gestire informazioni circa il mio account (modificare foto e altre informazioni)
- Come personale universitario, voglio poter ricevere notifiche circa offerte giornaliere

1.3.2 Use case stories: Ristoratore

Il secondo attore coinvolto nel sistema è il ristoratore, le cui Use Case Stories sono elencate di seguito:

ACQUISTA PASTO

- Come ristoratore, voglio poter visualizzare il saldo punti studente
- (Come ristoratore, voglio poter visualizzare e interagire rapidamente con piatti disponibili) → requisito non funzionale
- Come ristoratore, voglio poter selezionare i piatti nel vassoio dello studente al momento dell'ordine
- Come ristoratore, voglio poter produrre lo scontrino
- Come ristoratore, voglio poter essere avvisato quando la tessera dello studente ha punti con saldo inferiore alla soglia minima
- Come ristoratore, voglio poter effettuare il pagamento tramite tessera studente

GESTIONE GESTORI MENSE

- Come ristoratore, voglio poter fornire informazioni utili riguardo alla mia mensa

1.3.3 Use case stories: Amministratore

Il terzo attore coinvolto nel sistema è l'amministratore, le cui Use Case Stories sono elencate di seguito:

PROFILAZIONE MENSA

- Come amministratore, voglio poter registrare una serie di informazioni relative a un nuovo gestore mensa nel sistema del campus
- Come amministratore, voglio poter eliminare un gestore mensa
- Come amministratore, voglio poter effettuare il log-in/log-out dal sistema
- Come amministratore, voglio poter effettuare un test di accesso al sistema informativo di un nuovo gestore

LOG-IN/LOG-OUT

- Come amministratore, voglio poter fare il log.in/Log-out al mio account

1.3.4 Use case stories: Sistema di Gestione tessera

Il quarto attore coinvolto è il sistema di gestione tessera, che nonostante interagisca con il sistema è un attore esterno. Di seguito le sue Use Case Stories:

RICARICA PUNTI TESSERA

- Come sistema di gestione tessera, voglio poter tener traccia dei saldi punti di tutte le tessere che fanno parte dell'organizzazione universitaria.
- Il sistema di gestione tessera deve essere in grado di autorizzare una transazione di pagamento.

ACQUISTO PASTO

- Come sistema di gestione tessera, voglio poter autorizzare la transazione dell'acquisto di un pasto/piatto da parte di un utente.

1.3.5 Use case stories: Sistema di Autenticazione

Il quinto e ultimo attore coinvolto è il sistema di autenticazione, anch'esso esterno.

Di seguito le sue Use Case Stories:

LOG-IN/LOG-OUT

- Come sistema autenticazione, voglio permettere a tutto il personale universitario di potersi identificare ogni volta che il sistema del campus lo richiede.

1.4 Use case diagram

Partendo dalle *use case stories*, è stato redatto lo *use case diagram*. Tramite questo diagramma vengono specificati tutti i requisiti funzionali che il sistema dovrà implementare. Qui di seguito viene riportato il diagramma elaborato (Fig. 2):

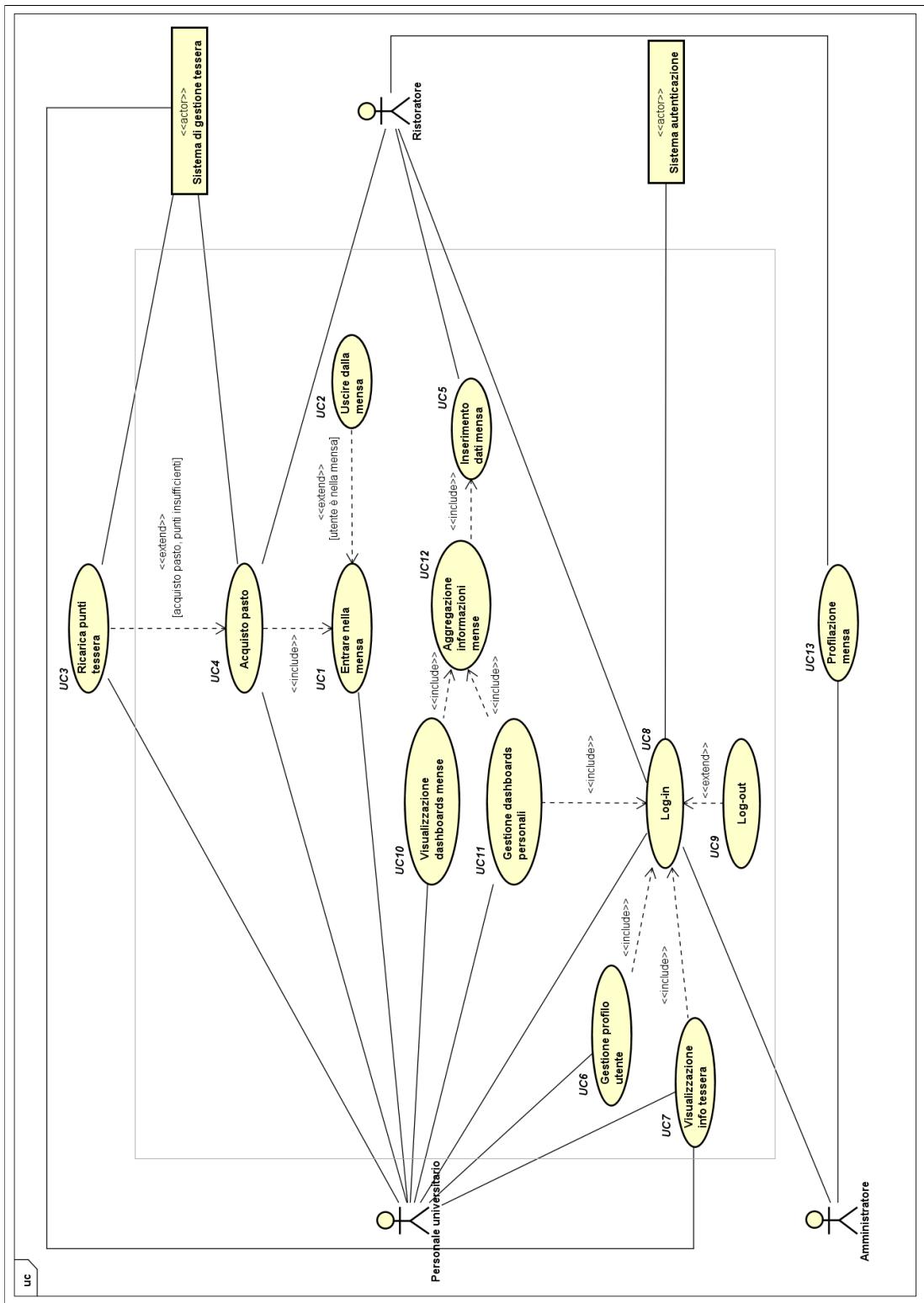


Fig. 2: Use Case Diagram

1.4.1 Fully dressed description

Per ogni use case, è stata redatta una descrizione completa (*fully dressed description*) di ogni possibile scenario coinvolto. Ogni scenario descrive una storia di interazione con il sistema, sia essa con esito positivo o negativo. In quest'ultimo caso, viene specificata anche una possibile risoluzione. Gli scenari sono stati raffinati di volta in volta con il procedere delle diverse iterazioni. Le *fully dressed description* di ogni use case sono presentate nelle figure delle pagine seguenti.

| ITEM | VALUE |
|--------------------|--|
| UseCase | Entrare nella mensa |
| Summary | Il sistema conta il numero di ingresso degli utenti |
| Actor | Personale universitario |
| Precondition | |
| Postcondition | Il sistema ha decrementato i posti disponibili della mensa. |
| Base Sequence | 1. L'utente si presenta al tornello 2. Il sistema verifica la disponibilità dei posti nella mensa. 3. Il sistema conferma la disponibilità dei posti nella mensa. [E1: non disponibilità posti] 4. Il sistema attiva il tornello e permette all'utente di entrare. 5. Il sistema decremente il numero dei posti disponibili nella mensa. |
| Branch Sequence | |
| Exception Sequence | E1: 3. Il sistema rileva che la mensa non ha posti disponibili. 4. Il sistema avvisa l'utente che la mensa è piena. 5. Ritorna al punto 1 del "Base sequence". |
| Sub UseCase | |
| Note | Sistema (generico) = sistema del campus |

Fig. 3: UC1: Entrare nella mensa

| ITEM | VALUE |
|--------------------|---|
| UseCase | Uscire dalla mensa |
| Summary | Il sistema conta il numero delle uscite degli utenti |
| Actor | Personale universitario |
| Precondition | L'utente deve essere entrato nella mensa |
| Postcondition | Il sistema ha incrementato i posti disponibili della mensa. |
| Base Sequence | 1. L'utente si presenta al tornello. 2. Il sistema incrementa i posti disponibili della mensa. |
| Branch Sequence | |
| Exception Sequence | |
| Sub UseCase | |
| Note | Sistema (generico) = sistema del campus |

Fig. 4: UC2: Uscire dalla mensa

| ITEM | VALUE |
|--------------------|---|
| UseCase | Ricarica punti tessera |
| Summary | Conversione denaro in punti tessera |
| Actor | Personale universitario, Sistema di gestione tessera |
| Precondition | |
| Postcondition | L'utente ha incrementato il numero di punti sulla propria tessera |
| Base Sequence | <ol style="list-style-type: none"> 1. L'utente manifesta la volontà di ricaricare la tessera e lo comunica al sistema di gestione tessera. 2. Il sistema di gestione chiede all'utente la quantità di punti da ricaricare con la corrispettiva quantità di denaro. 3. L'utente inserisce la quantità di punti desiderata e seleziona un metodo di pagamento (fisico/virtuale). 4. Il sistema verifica i dati inseriti. 5. I dati inseriti sono corretti. [E1: dati non corretti] 6. Il sistema effettua la transazione e aggiorna il saldo punti dell'utente. |
| Branch Sequence | |
| Exception Sequence | <p>E1:</p> <ol style="list-style-type: none"> 5. I dati inseriti non sono corretti. 6. Ritorna al punto 3 del "Base Sequence" |
| Sub UseCase | |
| Note | |

Fig. 5: UC3: Ricarica punti tessera

| ITEM | VALUE |
|--------------------|---|
| UseCase | Acquisto pasto |
| Summary | L'utente acquista il pasto presso la mensa. |
| Actor | Personale universitario, Ristoratore, Sistema di gestione tessera |
| Precondition | L'utente deve aver effettuato l'accesso all'area mensa. |
| Postcondition | L'utente ha completato la consumazione del pasto ed esce dalla mensa. |
| Base Sequence | <p>1. L'utente si mette in coda per acquistare il cibo.</p> <p>2. L'utente preleva il cibo e lo dispone sul proprio vassoio.</p> <p>3. L'utente giunge in cassa e mostra la propria tessera al cassiere.</p> <p>4. Il cassiere passa la tessera sul lettore.</p> <p>5. Il lettore comunica con il sistema di gestione tessera e riceve da esso il saldo punti dell'utente.</p> <p>6. Il cassiere seleziona i cibi sul vassoio del cliente.</p> <p>7. Il sistema verifica se il saldo disponibile è sufficiente.</p> <p>8. Il sistema conferma che il saldo disponibile è sufficiente. [E1: saldo insufficiente]</p> <p>9. Il ristoratore comunica tramite client le informazioni riguardanti l'acquisto.</p> <p>10. Il sistema riceve dal client le informazioni aggiornate delle rimanenze dei piatti.</p> |
| Branch Sequence | B1: <p>9. L'utente passa allo use case "Ricarica punti tessera"</p> <p>10. Ritorna al punto 4. della "Base Sequence"</p> |
| Exception Sequence | E1: <p>7. Il saldo dei punti disponibili sulla tessera non è sufficiente.</p> <p>8. Il cassiere annulla l'ordine. [B1: annulla ordine, ricarica tessera].</p> |
| Sub UseCase | Entrare nella mensa |
| Note | Quando non si specifica a che "sistema" si fa riferimento, si intende il "sistema del campus" che si sta sviluppando con questi UML diagram. |

Fig. 6: UC4: Acquisto pasto

| ITEM | VALUE |
|--------------------|--|
| UseCase | Inserimento dati mensa |
| Summary | Il ristoratore inserisce tramite client i dati giornalieri relativi alla propria mensa. |
| Actor | Ristoratore |
| Precondition | Il ristoratore è loggato nel sistema. |
| Postcondition | Le informazioni sono disponibili al sistema del campus. |
| Base Sequence | <p>1. Il ristoratore aggiunge tramite client tutte le informazioni giornaliere relative alla mensa, quali:</p> <ul style="list-style-type: none"> - Pasti disponibili - Menu del giorno - Orario di apertura - Sconti e/o promozioni <p>2. Il ristoratore conferma l'inserimento dei dati.</p> |
| Branch Sequence | |
| Exception Sequence | |
| Sub UseCase | |
| Note | |

Fig. 7: UC5: Inserimento dati mensa

| ITEM | VALUE |
|--------------------|--|
| UseCase | Gestione profilo utente |
| Summary | |
| Actor | Personale universitario |
| Precondition | L'utente è loggato nel sistema. |
| Postcondition | Le modifiche effettuate dall'utente sono state registrate. |
| Base Sequence | <ol style="list-style-type: none"> 1. L'utente si reca nell'area dedicata alla gestione delle preferenze. 2. Il sistema mostra all'utente le preferenze dello specifico utente. 3. L'utente richiede la modifica di alcune preferenze. 4. Il sistema aggiorna le preferenze dell'utente. 5. Il sistema conferma l'avvenuta modifica all'utente. |
| Branch Sequence | |
| Exception Sequence | |
| Sub UseCase | Log-in |
| Note | Con preferenze si intende: foto profilo, notifiche da ricevere, etc. |

Fig. 8: UC6: Gestione profilo utente

| ITEM | VALUE |
|--------------------|---|
| UseCase | Visualizzazione info tessera |
| Summary | L'utente visualizza le informazioni relative alla propria tessera. |
| Actor | Personale universitario, Sistema di gestione tessera |
| Precondition | L'utente è loggato nel sistema. |
| Postcondition | |
| Base Sequence | 1. L'utente si reca nell'area dedicata alla gestione della tessera. 2. Il sistema del campus richiede le informazioni della tessera utente al "sistema di gestione tessera" 3. Il "sistema di gestione tessera" fornisce le informazione della tessera utente al sistema del campus 4. Il sistema del campus fornisce all'utente: nome, cognome, matricola, codice tessera e saldo residuo. 5. L'utente visualizza le informazioni. |
| Branch Sequence | |
| Exception Sequence | |
| Sub UseCase | Log-in |
| Note | |

Fig. 9: UC7: Visualizzazione info tessera

| ITEM | VALUE |
|--------------------|--|
| UseCase | Log-in |
| Summary | |
| Actor | Sistema autenticazione, Amministratore, Personale universitario, Ristoratore |
| Precondition | L'utente è registrato nel sistema. |
| Postcondition | L'utente è loggato nel sistema. |
| Base Sequence | 1. Il sistema richiede all'utente le informazioni di accesso. 2. L'utente inserisce le informazioni richieste 3. Il sistema comunica le informazioni di accesso al "sistema di autenticazione" 4. Il "sistema di autenticazione" verifica la correttezza delle informazioni inserite. 5. Le informazioni sono corrette. [E1: informazioni sbagliate, utente non presente] 6. Il "sistema di autenticazione" comunica la correttezza delle informazioni al sistema del campus. 7. L'utente viene loggato nel sistema. |
| Branch Sequence | |
| Exception Sequence | E1: 4. Le informazioni inserite non sono corrette. 5. Il "sistema di autenticazione" comunica la non correttezza delle informazioni al sistema del campus. 6. Ritorna al passo 1. della "Base Sequence". |
| Sub UseCase | |
| Note | |

Fig. 10: UC8: Log-in

| ITEM | VALUE |
|--------------------|---|
| UseCase | Log-out |
| Summary | L'utente viene deautenticato dal sistema. |
| Actor | Personale universitario |
| Precondition | L'utente è loggato nel sistema. |
| Postcondition | L'utente non è più loggato nel sistema. |
| Base Sequence | 1. L'utente richiede al sistema di effettuare il log-out. 2. Il sistema effettua il log-out. |
| Branch Sequence | |
| Exception Sequence | |
| Sub UseCase | |
| Note | |

Fig. 11: UC9: Log-out

| ITEM | VALUE |
|--------------------|--|
| UseCase | Visualizzazione dashboards mense |
| Summary | Visualizzazione di tutte le informazioni utili delle varie mense su dashboards collocate nel campus. |
| Actor | Personale universitario |
| Precondition | Si è verificato l'use case "Aggregazione informazioni mense" |
| Postcondition | |
| Base Sequence | <p>1. Il sistema del campus, dopo aver ricevuto l'aggregazione delle infomazione delle varie mense, riorganizza tali informazioni secondo determinati criteri.</p> <p>2. L'utente si reca davanti ad una delle dashboard presenti nel campus.</p> <p>3. Sulla dashboard, il sistema mostra per ciascuna mensa le seguenti informazioni:</p> <ul style="list-style-type: none"> - mense aperte in quel momento - menù del giorno - mense divise per categoria (vegetariana/internazionale/mediterranea etc.) - posti liberi disponibili in quel momento - tempo stimato per la prossima disponibilità per le mense piene - tempo di attesa medio per l'acquisto di un pasto (coda) - piatti disponibili ed esauriti - offerte giornaliere/sconti/combo menù |
| Branch Sequence | |
| Exception Sequence | |
| Sub UseCase | Aggregazione informazioni mense |
| Note | |

Fig. 12: UC10: Visualizzazione dashboards mense

| ITEM | VALUE |
|--------------------|--|
| UseCase | Gestione dashboards personali |
| Summary | L'utente stabilisce e organizza quali informazioni visualizzare circa i pasti offerti nelle mense. |
| Actor | Personale universitario |
| Precondition | L'utente è loggato nel sistema. Si è verificato l'use case "Aggregazione informazioni mense". |
| Postcondition | L'utente ha riorganizzato le informazioni da visualizzare in base alle proprie preferenze. |
| Base Sequence | <ol style="list-style-type: none"> 1. L'utente chiede di visualizzare un filtraggio personalizzato delle informazioni relative alle diverse mense. 2. Il sistema riceve la struttura dati aggregata di tutte le mense. 3. Il sistema riorganizza e filtra le informazioni sulla base delle preferenze espresse dall'utente. 4. Il sistema aggiorna le informazioni e salva le modifiche. |
| Branch Sequence | |
| Exception Sequence | |
| Sub UseCase | Aggregazione informazioni mense, Log-in |
| Note | Con filtraggio personalizzato, si intende: <ul style="list-style-type: none"> - lista di mense preferite - filtraggio in base alla tipologia di cibo offerto - visualizzazione ordini passati - creazione di wishlist - ordinare in base prezzo/mensa preferita etc. - grafici circa i consumi |

Fig. 13: UC11: Gestione dashboards personali

| ITEM | VALUE |
|--------------------|--|
| UseCase | Aggregazione informazioni mense |
| Summary | Acquisizione dati dai client presenti in ogni mensa nel campus. |
| Actor | |
| Precondition | |
| Postcondition | Il sistema (del campus) ha ricevuto e aggregato tutte le informazioni utili di ogni mensa presente nel campus. |
| Base Sequence | 1. Il sistema del campus richiede le informazioni delle mense al client di ogni mensa presente nel campus. 2. Il sistema del campus riceve le informazioni da ogni client. [E1: errore ricezione] 3. Il sistema del campus aggrega in un'unica struttura le informazioni ricevute . |
| Branch Sequence | |
| Exception Sequence | E1: 2. Il sistema del campus non riesce a ricevere le informazioni da uno o più client . 3. Ritorna al punto 1. della "Base Sequence". |
| Sub UseCase | Inserimento dati mensa |
| Note | Sistema (generico) = sistema del campus |

Fig. 14: UC12: Aggregazione informazioni mense

| ITEM | VALUE |
|--------------------|--|
| UseCase | Profilazione mensa |
| Summary | Registrazione/eliminazione/modifica di un ristoratore |
| Actor | Amministratore, Ristoratore |
| Precondition | |
| Postcondition | |
| Base Sequence | 1. Il ristoratore chiede di poter aggiungere/eliminare/modificare la propria mensa all'amministratore. 2. L'amministratore esegue le modifiche. |
| Branch Sequence | |
| Exception Sequence | |
| Sub UseCase | |
| Note | |

Fig. 15: UC13: Profilazione mensa

1.5 Analisi delle specifiche

1.5.1 Introduzione alle specifiche

In questo capitolo andremo a descrivere ed ad analizzare tutte le specifiche funzionali del software da sviluppare. Abbiamo deciso di assegnare ad ogni specifica una priorità che è scelta in base a quante altre specifiche dipendono da essa:

- **Alta:** Sono le specifiche che realizzano gli aspetti cruciali del sistema e che, in quanto tali, sono da implementare prima di procedere con quelle a bassa priorità.
- **Bassa:** Sono le specifiche marginali da cui non dipendono altre specifiche.

1.5.2 Specifiche funzionali

Le specifiche funzionali del programma sviluppato sono le seguenti:

- **Inserimento dati mensa:** l'applicazione deve permettere ai ristoratori di ogni mensa di aggiungere prima dell'apertura le informazioni giornaliere, quali: menu, orari, piatti, promozioni, sconti, posti disponibili.
- **Aggiornamento status mensa:** l'applicazione deve permettere ai ristoratori di aggiornare le informazioni relative alla propria mensa, quali: posti occupati, piatti ancora disponibili, etc. Più l'applicazione interagisce con i ristoratori e più permette agli utenti di visualizzare le informazioni delle mense aggiornate in tempo reale.
- **Aggregazione informazioni mense:** l'applicazione deve essere in grado di ricevere le informazioni inserite dai ristoratori e di aggregarle secondo diversi criteri aumentandone la leggibilità. Inoltre, in questa specifica, l'applicazione riceve dati sui posti disponibili e stima il tempo medio di attesa in real-time.

- **Visualizzazione dashboard campus:** l'applicazione, una volta ricevute le informazioni, le elabora e le mostra sulle dashboard universitarie sparse per il campus.
- **Visualizzazione dashboard app:** l'applicazione, una volta ricevute le informazioni, le elabora e le mostra sulle dashboard consultabili dall'app. A differenza della specifica precedente, quest'ultima realizza un'interfaccia interattiva che permette all'utente di filtrare a suo piacimento le informazioni da visualizzare.
- **Registrazione di un nuovo utente:** l'applicazione deve permettere a un nuovo cliente di registrarsi al sistema. Importante: l'area personale è riservata al solo personale universitario.
- **LogIn/LogOut:** il software deve permettere agli utenti di effettuare il logIn e il logOut all'applicazione.
- **Visualizzazione punti tessera:** l'applicazione deve permettere all'utente che ha effettuato il logIn di visualizzare i propri punti tessera residui.
- **Visualizzazione informazioni utente:** l'applicazione, al momento dell'acquisto, deve mostrare al personale della mensa i punti tessera residui dell'utente che vuole acquistare il pasto.

| Codice | Nome | Priorità | Implementato |
|--------|-------------------------------------|----------|--------------|
| R1 | Inserimento dati mensa | A | SI |
| R2 | Aggiornamento status mensa | A | SI |
| R3 | Aggregazione informazioni mense | A | NO |
| R4 | Visualizzazione dashboard campus | A | SI |
| R5 | Visualizzazione dashboard app | A | NO |
| R6 | Registrazione di un nuovo utente | B | NO |
| R7 | LogIn/LogOut | B | NO |
| R8 | Visualizzazione informazioni utente | B | NO |

Tab. 1: Requisiti funzionali

1.6 Analisi dell'architettura

Per quanto riguarda l'architettura, sono stati realizzati due deployment diagram: il primo, meno formale, è stato utile al team per creare una idea generale dell'intero ecosistema; il secondo, più formale, è servito invece per mostrare nel dettaglio l'allocazione delle componenti software su quelle hardware.

1.6.1 Deployment diagram - informal

Il deployment diagram informale mette in evidenza i dispositivi che sono stati considerati nell'architettura e permette di fornire una visualizzazione ad alto livello delle componenti del sistema e della loro interazione.

Come si può notare nella Fig. 16, l'architettura è formata da un server centrale che interagisce in primo luogo con il database (nel quale sono salvati i dati necessari al software) e in secondo luogo con le altre componenti hardware:

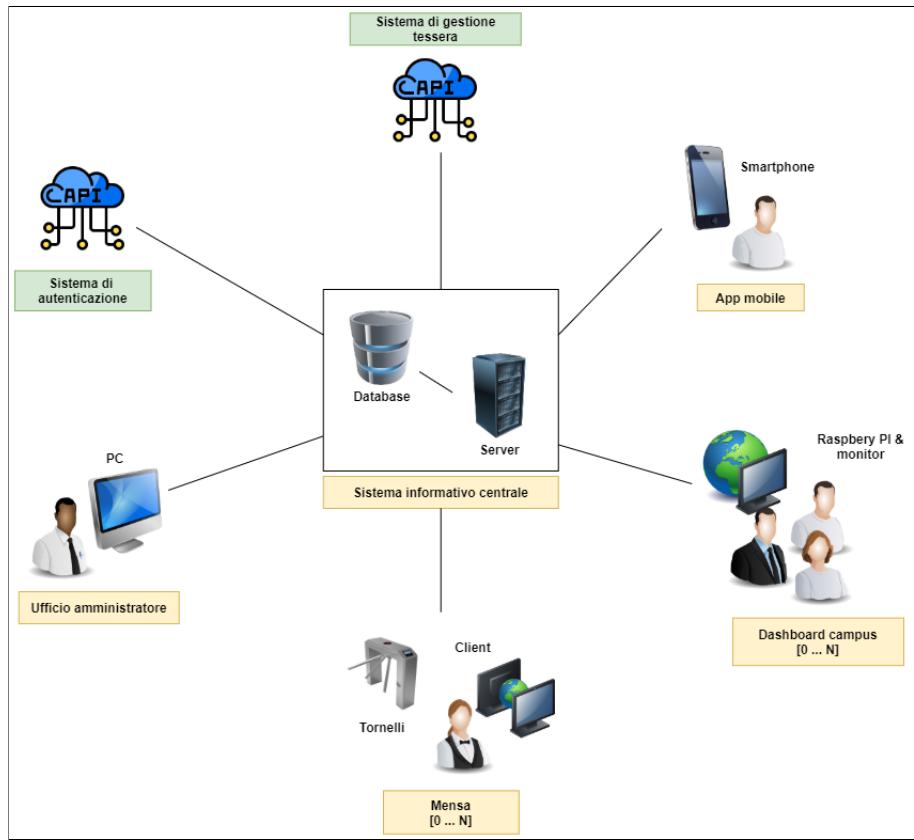


Fig. 16: Deployment diagram - informale

- **Mense**, dalle quali riceve da ciascun client mensa le informazioni inerenti agli orari di apertura, il menù, i prezzi e le promozioni del giorno e, in modo periodico, le informazioni aggiornate in tempo reale riguardanti i piatti ancora disponibili e il numero di posti a sedere liberi.
- **Dashboard campus**, alle quali invia le informazioni debitamente aggregate e riorganizzate per la visualizzazione statica.
- **App mobile**, alle quali invia le informazioni debitamente aggregate e riorganizzate per la visualizzazione dinamica modificabile tramite filtri locali.
- **Ufficio amministratore**, che ha la possibilità di gestire i ristoratori delle mense registrate al servizio.

- **Sistema di gestione tessera**, che gestisce le transazioni per la ricarica o la decurtazioni dei punti tessera degli utenti.
- **Sistema di autenticazione**, che gestisce l'autenticazione dei membri del personale universitario, dei ristoratori registrati al servizio e degli amministratori di sistema.

1.6.2 Deployment diagram - UML

Il deployment diagram formale (Fig. 17) rappresenta le stesse informazioni del diagramma precedente, ma espresse più formalmente. In particolare viene specificata la distinzione tra i device e i components, precisando la tecnologia comunicativa utilizzata.

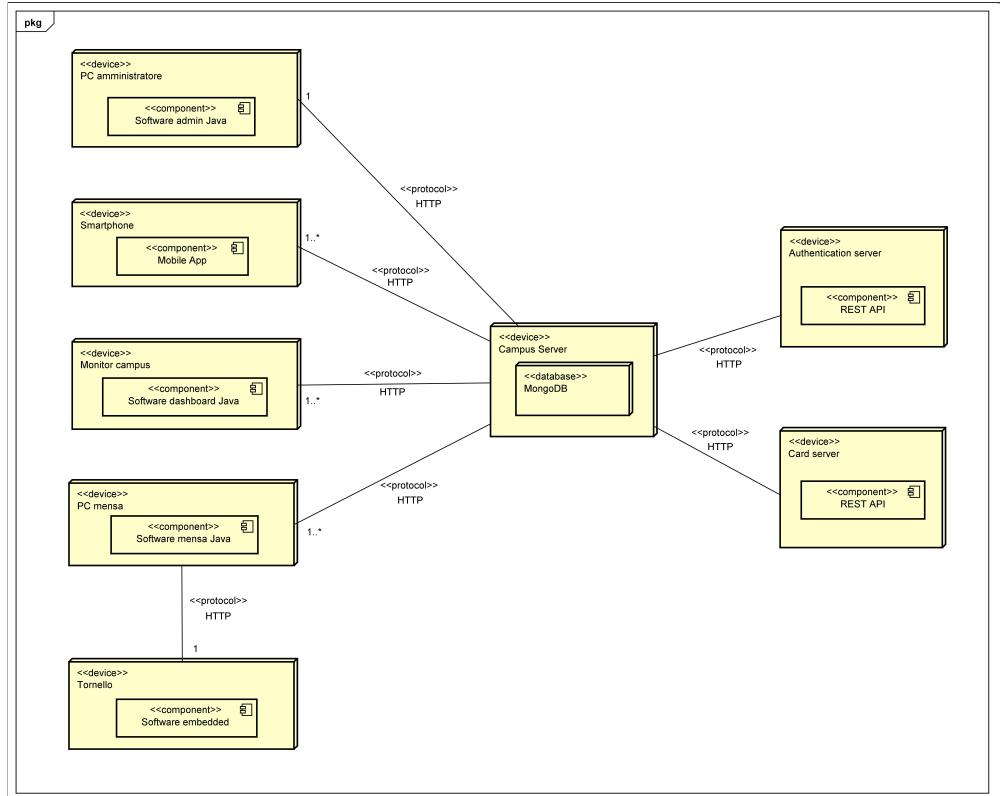


Fig. 17: Deployment diagram - UML

2 ITERAZIONE 1

Durante questa iterazione il team si è concentrato principalmente sullo sviluppo dell'intera architettura software.

Siamo partiti focalizzandoci sulla struttura interna del sistema sviluppando diverse rappresentazioni che ne hanno facilitato la comprensione dello stesso sotto diversi aspetti e, parallamente, abbiamo analizzato la conformazione del database e abbiamo sviluppato di conseguenza il data class diagram.

2.1 Component diagram - single component

Per la rappresentazioni delle componenti siamo partiti rappresentando il sistema come un unico macro-elemento ad alto livello come mostrato in Fig. 18 che espone interfacce tramite la tecnologia API REST.

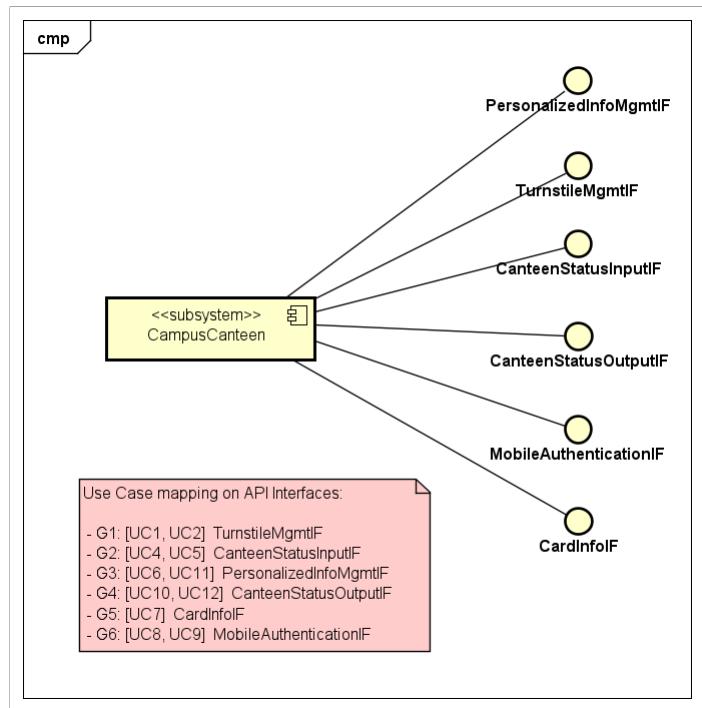


Fig. 18: Component diagram - single component

2.2 Component diagram delle interfacce

Il diagramma delle componenti presente in Fig. 19 rappresenta i principali elementi software del sistema e pone l'attenzione su come essi interagiscono tra di loro.

Come si può vedere, infatti, tramite la notazione *ball and socket* è stato messo in risalto quale componente espone una data interfaccia e quale componente invece ne usufruisce.

2.3 Interface definition

Il diagramma rappresentato in Fig. 20 definisce le interfacce di cui ogni componente del diagramma presentato nel paragrafo precedente si serve per effettuare tutte le operazioni necessarie al suo funzionamento.

Ogni interfaccia prevede, al suo interno, una serie di metodi che, allo stato attuale, sono rappresentati con una segnatura semplificata.

E' però importante specificare che la segnatura dei metodi non è definita nella sua totalità, in quanto essi verranno specificati nella loro implementazione definitiva solo durante le successive fasi di sviluppo del processo Agile.

Di conseguenza, anche i prototipi dei metodi presenti in questo diagramma verranno raffinati con precisione nelle prossime iterazioni, in cui verrà definita quindi anche la versione finale delle operazioni effettuabili tramite le interfacce stesse.

2.4 Data class diagram

Il diagramma dei dati delle classi, come mostrato in Fig. 21, descrive la struttura della base di dati su cui si appoggia il sistema e le relazioni statiche che sussistono tra gli oggetti che lo compongono.

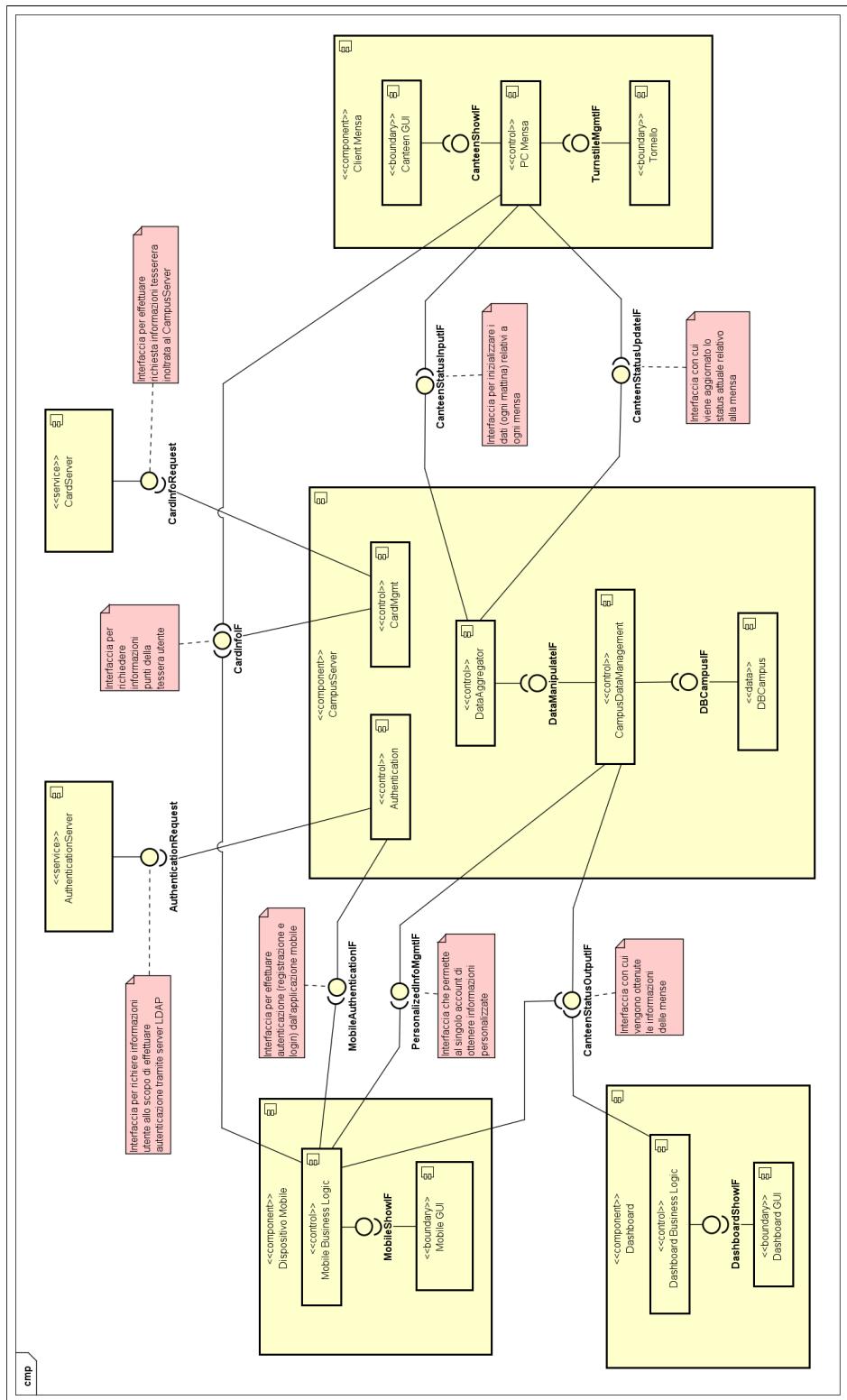


Fig. 19: Component diagram delle interfacce

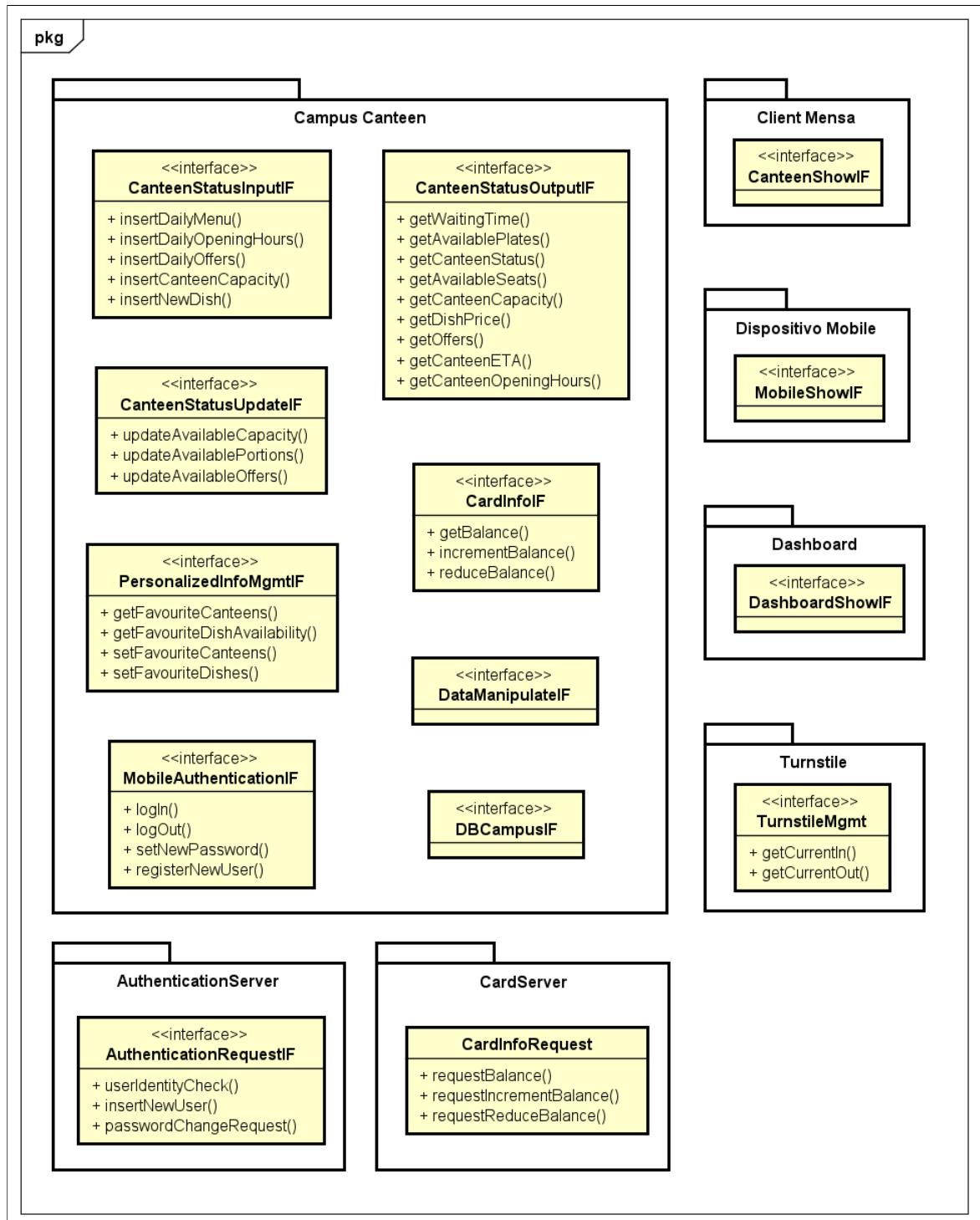


Fig. 20: Interface definition diagram

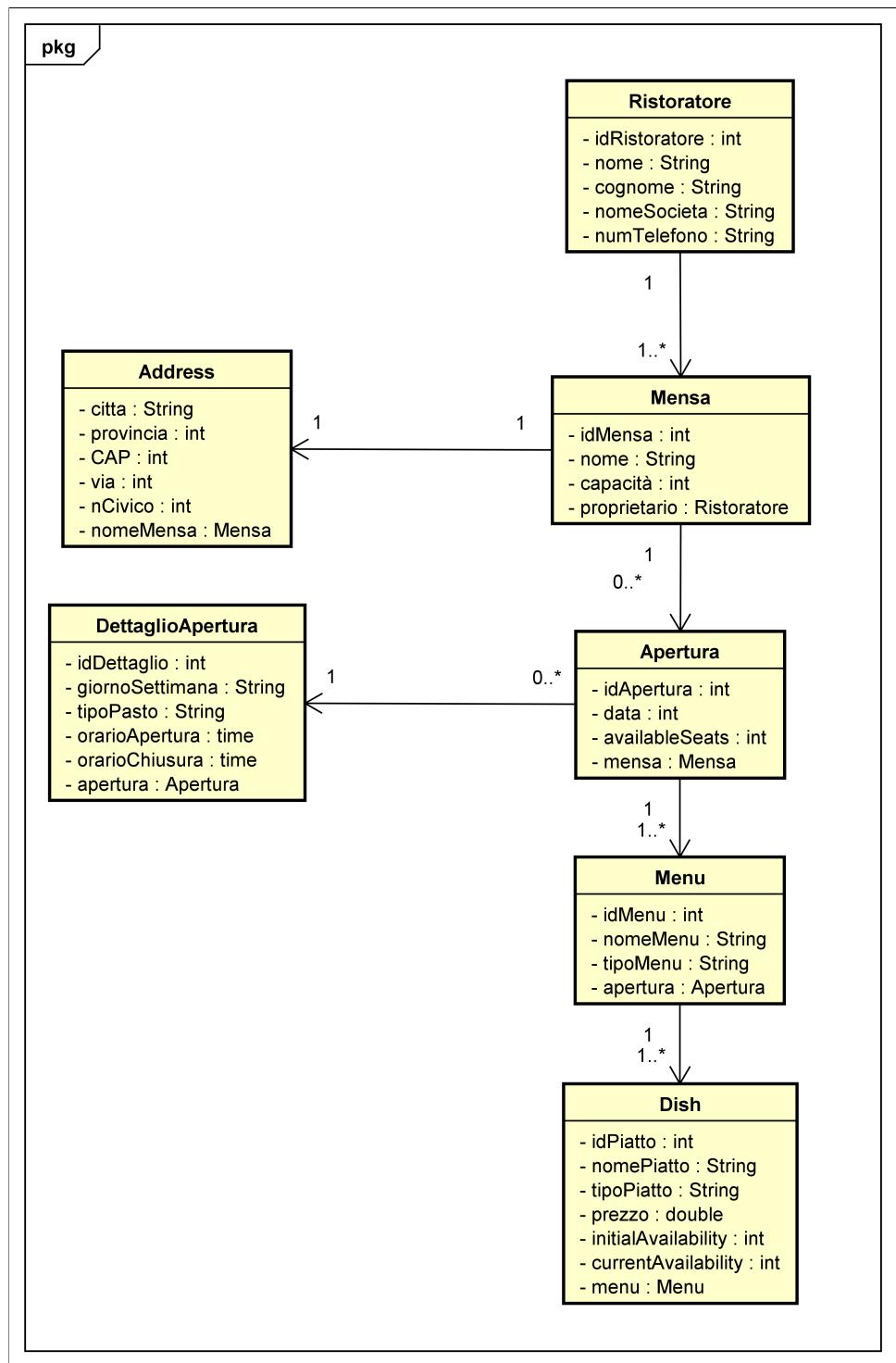


Fig. 21: Data class diagram

3 ITERAZIONE 2

3.1 Implementazione

Lo scopo di questo progetto è quello di creare un'applicazione software che gestisca in tempo reale tutte le informazioni che arrivano da più mense collocate nelle vicinanze di un campus universitario. L'applicazione permette agli utenti (principalmente studenti e personale universitario) di avere una visione in tempo reale di dati utili di tutte quelle mense e ristoranti che partecipano all'iniziativa.

In questa fase il team si è riunito per decidere quale funzionalità implementare e quali tecnologie utilizzare.

3.2 Selezione funzioni implementate

Si è scelto di sviluppare in questa fase le funzionalità legate alla visualizzazione dei dati sulle dashboard statiche mostrate sui monitor sparsi per il campus. In particolare si è deciso di trattare lo Use Case **Visualizzazione dashboard campus** dello Use Case diagram della fase 0 riportato in Fig. 22.

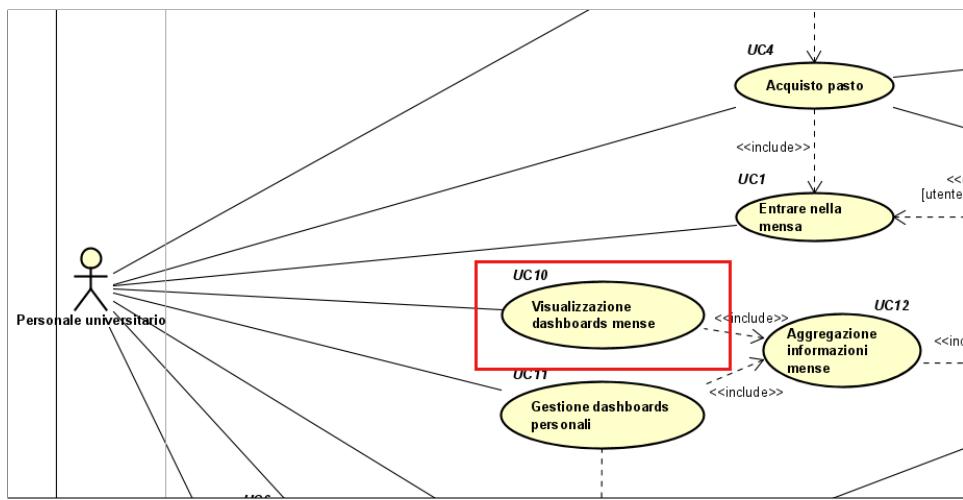


Fig. 22: Use Case selezionato

Si è implementata l'interfaccia di `CanteenStatusOutputIF` del diagramma delle

interfacce(Fig. 20) e i relativi metodi di cui nella fase precedente si è definito solo il nome e non la segnatura completa.

3.3 Component diagram White-Box

Si è deciso di implementare il caso d’uso *visualizzazione dashboard mense*, pertanto la componente che implementa questa funzionalità è descritta all’interno del component diagram black box della fase 1 ed in esso compare con il nome **CampusDataManagement**. Questa componente espone un’interfaccia per la visualizzazione delle dashboard statiche che può essere utilizzata sia dai client connessi ai monitor all’interno del campus che dall’app mobile scaricabile dagli utenti del campus. Tale interfaccia ha nome **CanteenStatusOutputIF** e definisce i metodi descritti nel class diagram in Fig. 25.

Tale componente interagisce all’interno del server con due altre componenti: la prima è quella che descrive il database non relazionale (**DBCampus**) e la seconda è quella che effettua l’aggregazione dei dati provenienti dai client presenti in ciascuna mensa (**DataAggregator**). In particolare, la componente **CampusDataManagement** interagisce con il database effettuando su di esso delle query per ottenere i dati da esporre tramite API ai client. La comunicazione tra i due avviene grazie alla dependecie definita nel framework Spring. L’interazione con la componente di aggregazione dati dà la possibilità ai dati provenienti dalle diverse mense di essere elaborati prima di essere memorizzati sul database.

In Fig. 23 si riporta il component diagram white box che mette in evidenza le interazioni tra questi componenti dal punto di vista delle funzionalità per l’ottenimento dati mense.

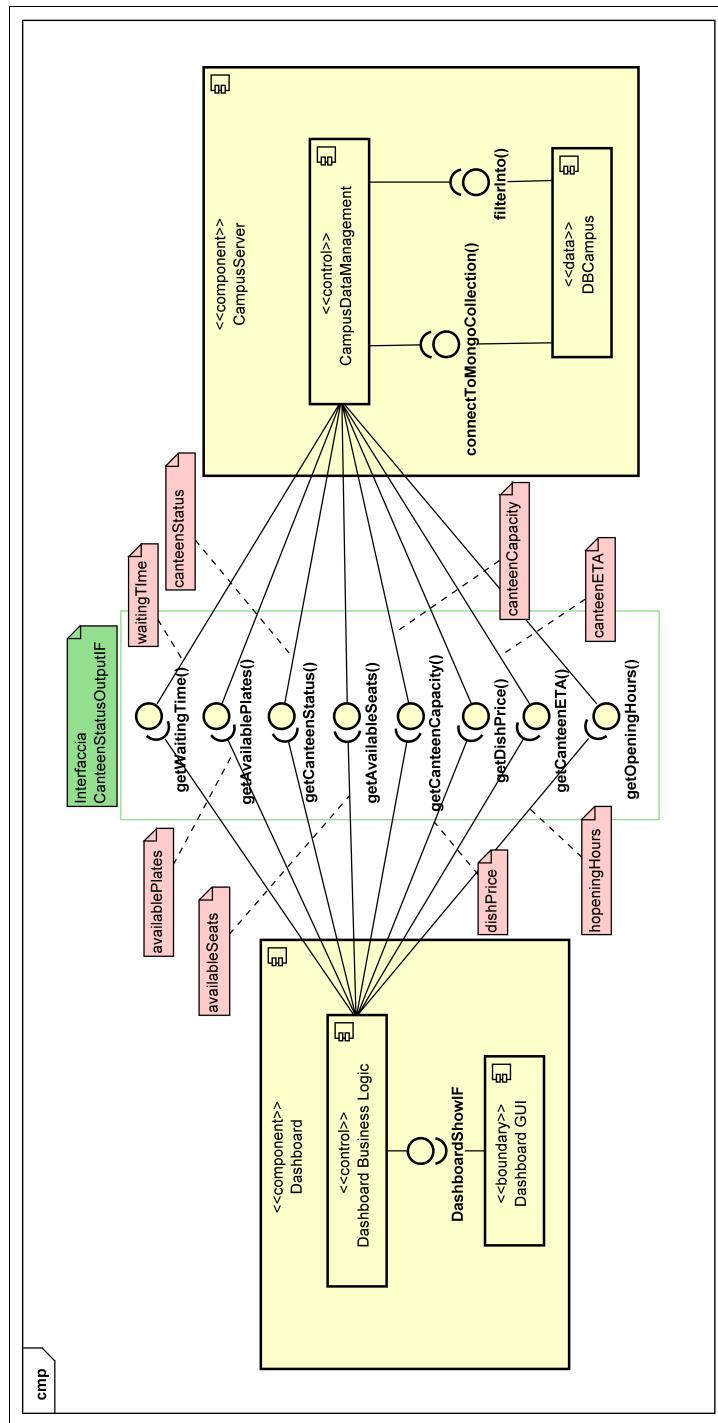


Fig. 23: Component Diagram white-box

3.4 Sequence diagram per visualizzazione

Il sequence diagram sviluppato in questa fase mette in evidenza l'interazione tra il client (sia app mobile che monitor campus) e la componente di visualizzazione offerta dal server. In particolare distinguiamo due diversi comportamenti per ciascuno dei due client rispetto alla componente di visualizzazione: le dashboard dei monitor richiedono periodicamente i dati contenuti nel database forniti tramite API (la quale sfrutta i metodi contenuti nell'interfaccia `CanteenStatusOutput` implementata in questa fase), mentre le app mobile effettuano una richiesta esplicita dei dati in funzione della volontà dell'utente.

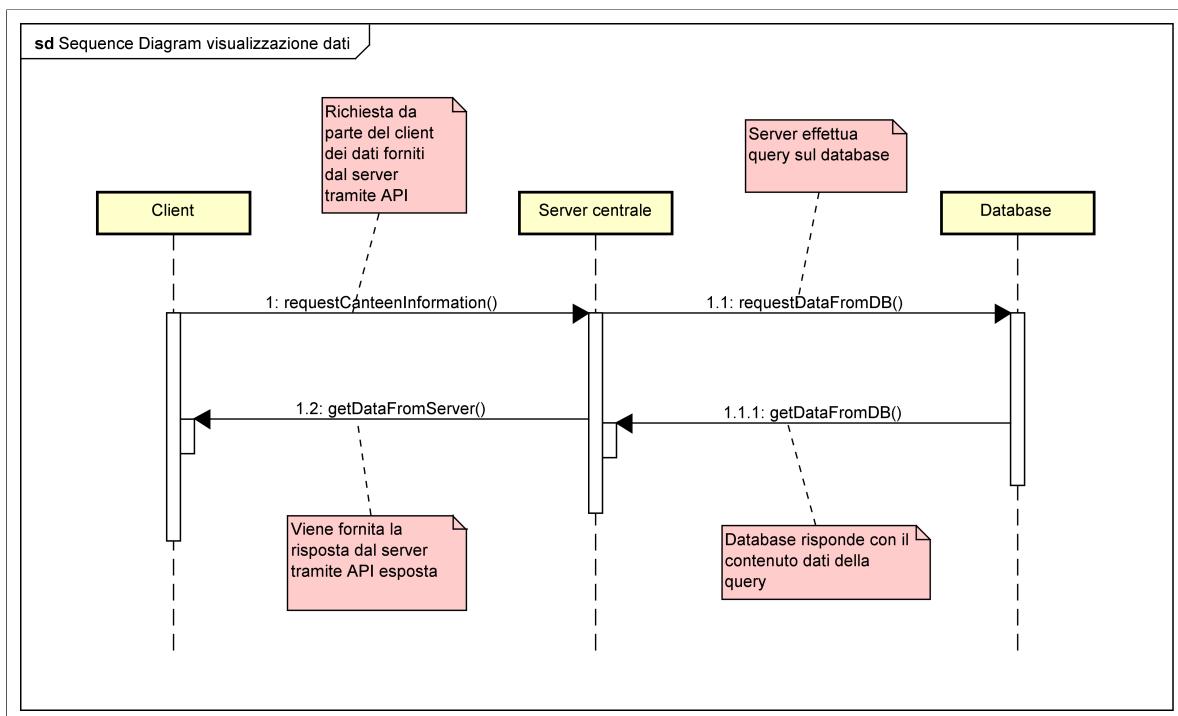


Fig. 24: Sequence diagram per visualizzazione

Dal sequence diagram in Fig. 24 si può notare come la richiesta tramite API del client venga implementata sul server tramite la chiamata dei metodi implementati nell'interfaccia `CanteenStatusOutputIF`. La chiamata di un metodo dell'interfaccia sul

server comporta l'esecuzione di query sul database, il quale restituisce un opportuno oggetto JSON di cui viene effettuato il parsing per estrarre le informazioni ricercate.

3.5 Class diagram dei metodi per visualizzazione

Il class diagram (Fig. 25) riferito all'interfaccia implementata in questa fase mette in evidenza la segnatura specifica dei metodi e i valori da essi ritornati i metodi dell'interfaccia risiedono nella componente CanteenDataManagement.

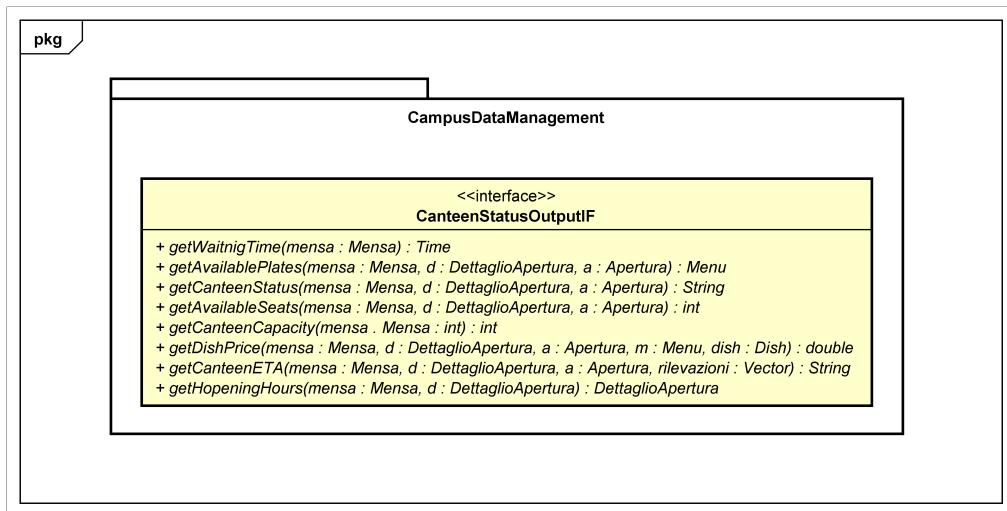


Fig. 25: Class diagram dell'interfaccia di visualizzazione

3.6 Tecnologie utilizzate

Questa fase iniziale ha comportato anche la scelta della tecnologia da utilizzare per la memorizzazione dei dati su database e la tecnologia da utilizzare per l'implementazione del software applicativo.

3.6.1 MongoDB Atlas

Abbiamo deciso di utilizzare **MongoDB Atlas** come data lake: un DBMS non relazionale che ci ha permesso di semplificare l'interazione con l'applicativo sfruttando il formato Json per l'interscambio dei dati. Inoltre, l'utilizzo di questa tecnologia ci ha evitato di hostare localmente il database utilizzando quindi i server di **MongoDB Atlas**.

3.6.2 Robo 3T

Durante l'implementazione del database abbiamo utilizzato anche il software gratuito **Robo 3T** allo scopo di visualizzare il contenuto del database ed inserire alcuni dati di prova utili per effettuare i test delle funzionalità implementate.

3.6.3 Java

Come linguaggio di programmazione si è scelto **Java** nella sua versione SE 15 (JRE 8). L'utilizzo di java con il database non relazionale MongoDB Atlas ha permesso di interagire facilmente con il data lake grazie al supporto nativo del formato di interscambio dei dati Json. Per l'implementazione delle API REST abbiamo scelto di utilizzare la tecnologia **Java Maven** per la sua semplicità di integrazione di diversi framework: nel nostro caso per l'implementazione delle API REST abbiamo deciso di utilizzare il framework **Spring**.

3.7 MongoDB: API di connessione

Per fare interagire l'applicativo Java con il database non relazione sviluppato, il framework **Spring** mette a disposizione nel file `pom.xml` la possibilità di specificare delle dependencies allo scopo di rendere possibile la connessione tra i due senza l'utilizzo di librerie esterne. Di seguito un estratto del file `pom.xml` che mette in evidenza la specifica della dependency che garantisce l'interazione descritta.

```
<dependency>
    <groupId>org.mongodb</groupId>
    <artifactId>mongo-java-driver</artifactId>
    <version>3.4.0</version>
</dependency>
```

3.8 Pseudocodice metodo: GetCanteenETA()

Il metodo `GetCanteenETA()` (Fig. 26) restituisce una previsione di quando la mensa selezionata avrà nuovamente posti disponibili. Nello pseudocodice si può vedere l'utilizzo del vettore `rilevazioni[]`, il quale contiene i dati dei posti occupati (della relativa mensa) in ordine temporale. In questo modo è possibile, attraverso un modello lineare, stimare i posti occupati alla prossima rilevazione.

Considerando che questo metodo viene chiamato ogni 10 minuti, allora `GetCanteenETA()` aggiungerà ogni volta nel vettore una nuova rilevazione, ma calcolerà la stima solo quando vede tutti i posti sono occupati. Ovviamente, non avendo a disposizione il dato relativo all'ingresso e all'uscita della singola persona all'interno della mensa, questa stima non è perfettamente accurata ed è stata sviluppata solo per dimostrare le capacità di questo metodo.

```

1 Alg getCanteenETA (Mensa m, dettaglioApertura d, Apertura a, vettore rilevazioni[], int t) {
2     //Presupponiamo che la funzione getCanteenETA venga chiamata ogni 10 minuti
3     //t varrabile che esprime il tempo in minuti
4
5     tempo = chrono.now()
6
7     //persone contate dal tornello posizionato all'entrata di m
8     int p_entrate = numero di persone entrate nella mensa
9     //persone contate dal tornello posizionato all'uscita di m
10    int p_uscite = numero di persone uscite dalla mensa
11    int postiOccupati = p_entrate - p_uscite
12    rilevazioni.pushback((postiOccupati,tempo))
13
14    if ( postiOccupati > a.AvailableSeats )
15        throws Exception
16    if ( postiOccupati == a.AvailableSeats && tempo >= 3){
17        Double stima = 0.60*rilevazioni[tempo-t] +
18                      0.25*rilevazione[tempo-2*t] +
19                      0.15*rilevazione[tempo-3*t]
20
21        if ( stima > rilevazioni[tempo-t].postiOccupati )
22            return "Posti disponibili tra più di 10 minuti"
23            // Ci sono posti liberi, ma pochi:
24            // la mia stima è > del dato all'istante tempo-t
25        else
26            return "Posti disponibili entro 10 minuti"
27            // Ci sono posti liberi, ma pochi:
28            // la mia stima è < del dato al all'istante tempo-t
29    }
30 }

```

Fig. 26: Pseudocodice algoritmo getCanteenETA()

3.8.1 Analisi di complessità metodo: GetCanteenETA()

Ora analizziamo la complessità dell'algoritmo in Fig. 26 ponendoci nel caso peggiore.

Supponiamo di costo costante qualsiasi operazione.

Analizzando il codice ci accorgiamo che non ci sono cicli. Il metodo effettua un semplice calcolo matematico sui dati presenti in un vettore passato come parametro. Di conseguenza la complessità totale dello pseudocodice dell'algoritmo **GetCanteenETA()** è **O(1)**.

3.9 Testing API di visualizzazione

Abbiamo verificato la corretta implementazione delle API esposte dal componente **CampusDataManagement** che permettono di effettuare la visualizzazione dei dati attraverso i metodi definiti nell’interfaccia **CanteenStatusOutputIF** tramite l’utilizzo del software **Postman**. Quest’ultimo permette di analizzare il comportamento di richieste REST tramite il verbo HTTP ”GET” con le quali è possibile effettuare il corretto funzionamento delle API da noi esposte. In Fig. 27, Fig. 28, Fig. 29 sono riportati gli screenshot dei test effettuati sulle API riguardanti le funzionalità di visualizzazione effettivamente implementate in questa fase.

La API in Fig. 27 permette di ottenere la capacità della mensa specificata come parametro alla richiesta; per l’ottenimento di questo dato viene invocato il metodo `getCanteenCapacity()` che effettua la query sul database e restituisce il valore ricercato.

Un esempio di richiesta GET è il seguente:

```
http://localhost:8080/getCanteenCapacity?nomeMensa=I saporì della terra
```

| KEY | VALUE | DESCRIPTION | ... | Bulk Edit |
|---|----------------------|-------------|-----|-----------|
| <input checked="" type="checkbox"/> nomeMensa | I saporì della terra | | | |
| Key | Value | Description | | |

Body Cookies Headers (5) Test Results 200 OK 236 ms 227 B Save Response ▾

Pretty Raw Preview Visualize JSON ▾

```

1  {
2    "nomeMensa": "I saporì della terra",
3    "canteenCapacity": 200
4  }
  
```

Fig. 27: API per il metodo `getCanteenCapacity()`

La API in Fig. 28 permette di ottenere il numero di posti disponibili in una determinata apertura di una certa mensa specificata come parametro alla richiesta; per l'ottenimento di questo dato viene invocato il metodo `getAvailableSeats()` che effettua la query sul database e restituisce il valore ricercato. In questo caso è necessario specificare il nome della mensa, il giorno della settimana di interesse, il tipo di pasto (quindi se colazione, pranzo o cena) e la data dell'apertura ricercata.

Un esempio di richiesta GET è il seguente:

```
http://localhost:8080/getAvailableSeats?nomeMensa=I saperi della terra&giornoSettimana=Lunedì&tipoPasto=Pranzo&data=04-01-2021
```

| KEY | VALUE | DESCRIPTION | ... | Bulk Edit |
|---|----------------------|-------------|-----|-----------|
| <input checked="" type="checkbox"/> nomeMensa | I saperi della terra | | | |
| <input checked="" type="checkbox"/> giornoSettimana | Lunedì | | | |
| <input checked="" type="checkbox"/> tipoPasto | Pranzo | | | |
| <input checked="" type="checkbox"/> data | 04-01-2021 | | | |
| Key | Value | Description | | |

Body Cookies Headers (5) Test Results 200 OK 244 ms 314 B Save Response ▾

Pretty Raw Preview Visualize JSON ⚡

```

1  [
2    "data": "04-01-2021",
3    "giornoSettimana": "Lunedì",
4    "availableSeats": 150,
5    "nomeMensa": "I saperi della terra",
6    "tipoPasto": "Pranzo"
7  ]
  
```

Fig. 28: API per il metodo `getAvailableSeats()`

La API in Fig. 29 permette di ottenere il prezzo di un determinato piatto all'interno di una certa mensa in un dato giorno (si presuppone infatti che il ristoratore abbia la possibilità di specificare, ad ogni apertura, il prezzo e la quantità disponibili

di ogni piatto nel menù); per l'ottenimento di questo dato viene invocato il metodo `getDishPrice()` che effettua la query sul database e restituisce il valore ricercato. In questo caso è necessario specificare il nome della mensa, il giorno della settimana di interesse, il tipo di pasto (quindi se colazione, pranzo o cena), la data, il nome del menù, il tipo di menù ed il nome di piatto.

Un esempio di richiesta GET è il seguente:

```
http://localhost:8080/getDishPrice?nomeMensa=I saperi della terra
&giornoSettimana=Lunedì&tipoPasto=Cena&data=04-01-2021&nomeMenu=cenaLunedì
&tipoMenu=Mediterraneo&nomePiatto=Pasta al salmone
```

| KEY | VALUE | DESCRIPTION | ... | Bulk Edit |
|---|----------------------|-------------|-----|-----------|
| <input checked="" type="checkbox"/> nomeMensa | I saperi della terra | | | |
| <input checked="" type="checkbox"/> giornoSettimana | Lunedì | | | |
| <input checked="" type="checkbox"/> tipoPasto | Cena | | | |
| <input checked="" type="checkbox"/> data | 04-01-2021 | | | |
| <input checked="" type="checkbox"/> nomeMenu | cenaLunedì | | | |
| <input checked="" type="checkbox"/> tipoMenu | Mediterraneo | | | |
| <input checked="" type="checkbox"/> nomePiatto | Pasta al salmone | | | |
| Key | Value | Description | | |

Body Cookies Headers (5) Test Results 200 OK 1306 ms 406 B Save Response ▾

Pretty Raw Preview Visualize JSON ↴

```

1  [
2    "nomeMenu": "cenaLunedì",
3    "data": "04-01-2021",
4    "tipoMenu": "Mediterraneo",
5    "giornoSettimana": "Lunedì",
6    "dishPrice": 2,
7    "nomePiatto": "Pasta al salmone",
8    "nomeMensa": "I saperi della terra",
9    "tipoPasto": "Cena"
10 ]
  
```

Fig. 29: API per il metodo `getDishPrice()`

3.10 Casi di test

Durante questa iterazione sono stati realizzati tre metodi:

- **getAllCanteens**, che restituisce tutti gli oggetti mensa presenti nel database;
- **getAllCanteensNames**, che restituisce la lista dei nomi di tutte le mense presenti nel database;
- **getCanteenCapacity**, che prendendo in input un oggetto Mensa restituisce la capacità della stessa dichiarata dal ristoratore;
- **getAvailableSeats**, che prendendo in input un oggetto Mensa, un oggetto DettaglioApertura e un oggetto Apertura restituisce i posti attualmente disponibili;
- **getAvailablePlates**, che prendendo in input un oggetto Mensa, un oggetto DettaglioApertura, un oggetto Apertura e un oggetto Menu restituisce la lista dei piatti ricercati;
- **getDishPrice**, che prendendo in input un oggetto Mensa, un oggetto DettaglioApertura, un oggetto Apertura, un oggetto Menu e un oggetto Dish restituisce il prezzo del piatto ricercato;

Per quanto riguarda i casi di test sono stati implementati in totale undici metodi:

- Sei metodi, uno per ogni funzione implementata, per testare il loro corretto funzionamento;
- Quattro metodi, uno per ogni funzione implementata ad eccezione delle prime due, per testare il comportamento nel caso in cui venga inserito un filtro in input errato (i metodi verificano che venga lanciata l'eccezione prevista);

- Un metodo per la copertura delle altre funzioni non implementate, che verifica se essi ritornano correttamente il valore null.

Come si può vedere in Fig. 37, tutti i test previsti risultano superati e coprono interamente la classe di implementazione delle funzioni realizzate.

The screenshot shows a JUnit test run interface. At the top, it says "Finished after 25,472 seconds". Below that, it shows "Runs: 11/11", "Errors: 0", and "Failures: 0", all in green. A large green bar indicates success. On the left, there's a tree view of test classes and methods. Under "CanteenStatusOutputTest [Runner: JUnit 5] (25,381 s)", there are 11 methods listed with their execution times: getAvailablePlatesExceptionTest (3,791 s), getAvailablePlatesTest (2,462 s), getDishPriceExceptionTest (2,334 s), getAvailableSeatsTest (2,420 s), getAvailableSeatsExceptionTest (2,220 s), getAllCanteensTest (2,380 s), getDishPriceTest (2,424 s), getAllCanteensNamesTest (2,470 s), getCapacityExceptionTest (2,282 s), unimplementedMethodTest (0,004 s), and getCapacityTest (2,594 s). To the right of the tree view is the corresponding Java code:

```

1 package Test;
2
3 import static org.junit.Assert.assertEquals;
4
5 public class CanteenStatusOutputTest {
6
7     CanteenStatusOutputIMPL obj = new CanteenStatusOutputIMPL();
8
9     @Test
10    public void getAllCanteensTest() {
11        assertNotNull("", obj.getAllCanteens());
12    }
13
14    @Test
15    public void getAllCanteensNamesTest() {
16        assertNotNull("", obj.getAllCanteensNames());
17    }
18}

```

Fig. 30: risultato del test sulle funzioni di visualizzazione implementate

3.11 Analisi statica del codice (STAN4J)

A iterazione conclusa è stata effettuata un'analisi statica del codice mediante lo strumento *STAN4J*. Questa tecnologia permette di effettuare un'analisi visiva del progetto nella sua interezza, e di esplorarne i vari moduli e pacchetti presenti per vedere di cosa essi sono composti e quali e quante dipendenze possiedono.

Osservando quanto riportato dal tool, abbiamo reingegnerizzato il codice per poter soddisfare nel modo migliore diversi parametri. Di seguito vengono presentate le principali caratteristiche codice a fine Iterazione 2.

3.11.1 Legame tra Abstractness e Instability

Identifica quanto una categoria è lontana dal caso ideale. Minore è la distanza del software dalla linea maggiore è la sua qualità. Come possiamo osservare nella figura

sottostante (Fig. 31), la componente sviluppata in questa iterazione si trova molto vicino alla retta ideale. Questo è un ottimo segno e indica un buon equilibrio tra Abstractness e Instability.

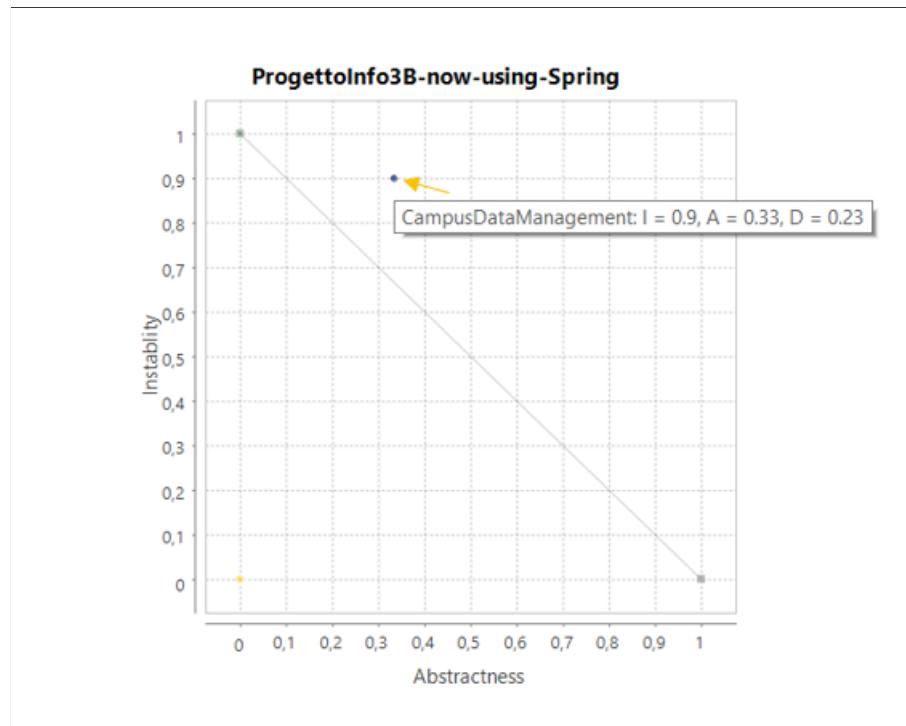


Fig. 31: Legame tra Abstractness e Instability

3.11.2 Map View

Questa visualizzazione ci permette di individuare le dipendenze attraverso un approccio visivo legato ai colori. Il codice rappresentato da uno dei blocchi, dipende dal codice rappresentato dal blocco sottostante e ha una dipendenza bidirezionale dai blocchi collocati sul suo stesso livello. La Fig. 32 ci consente di vedere le dipendenze presenti nella nostra applicazione software e possiamo dire che sono coerenti con quello che ci aspettavamo.

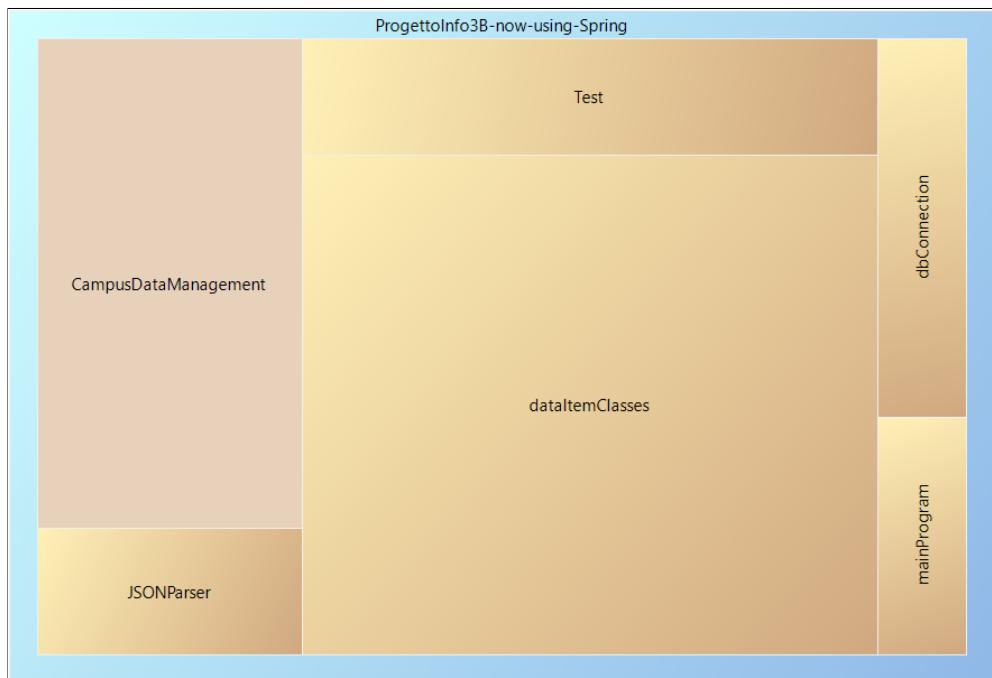


Fig. 32: Map view

3.11.3 Composition View

Questa view ci mostra i package che compongono il software sotto analisi. Come si può vedere dalla Fig. 36 il package mostrato dalla analisi strutturale coincide con quello progettato nelle fasi precedenti. Possiamo dire quindi di aver rispettato le specifiche software decise nelle iterazioni passate.

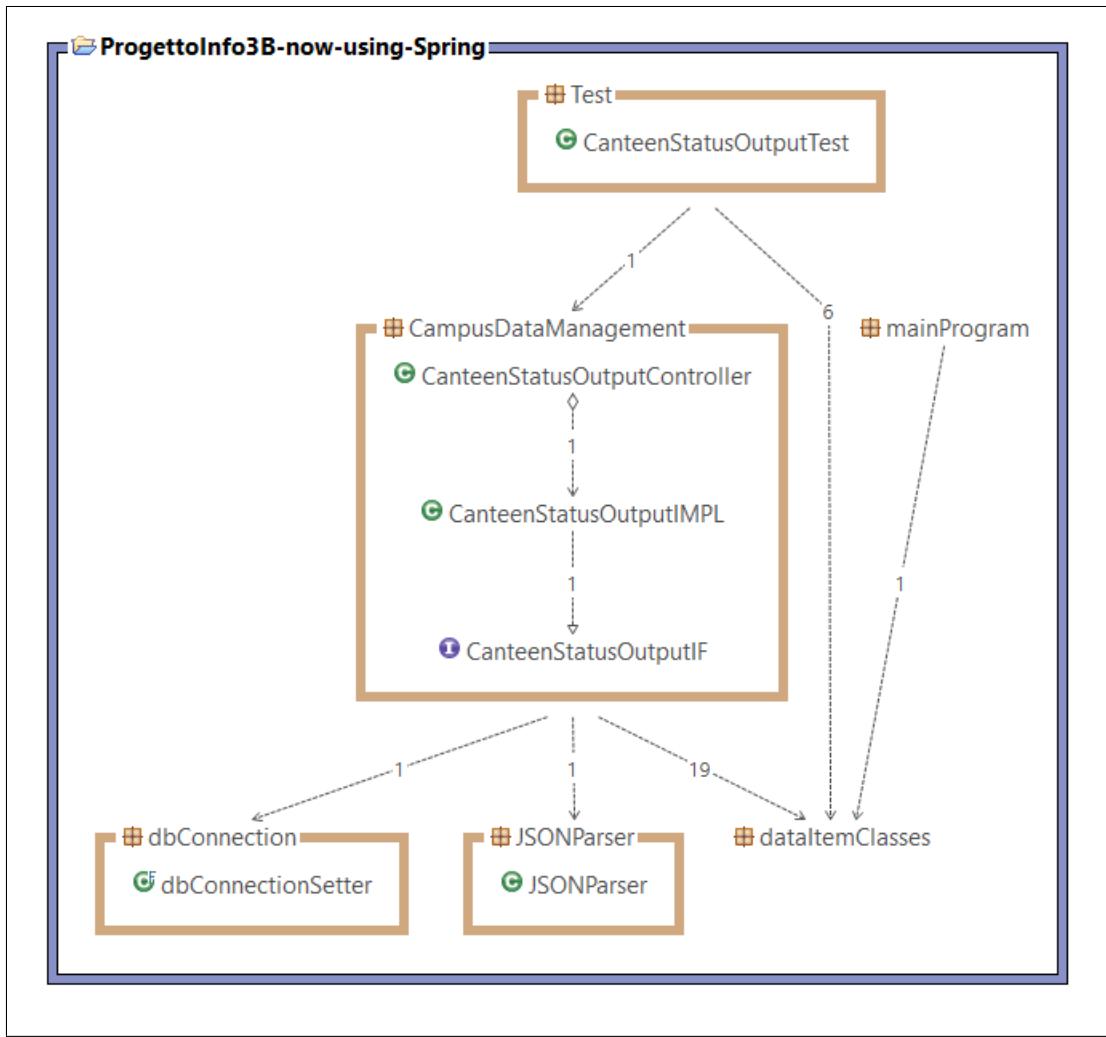


Fig. 33: Composition View

3.12 Conclusioni iterazione 2

L’iterazione 2 si conclude con la corretta e funzionante implementazione delle API esposte dalla componente *CampusDataManagement*. Lo use case ”*UC10: Visualizzazione dashboards mense*” verrà ripreso e approfondito ulteriormente nelle prossime iterazioni. Di seguito è riportata una tabella (Tab. 2) che racchiude i metodi implementati a fine iterazione 2 e correttamente esposti dalla API del nostro sistema.

In questa fase abbiamo deciso di aggiungere l’ultimo metodo `getAllCanteen()`

| Codice | Metodi (API) | Implementato |
|--------|----------------------|--------------|
| R4 | getWaitingTime() | NO |
| R4 | getAvailablePlates() | NO |
| R4 | getCanteenStatus() | NO |
| R4 | getAvailableSeats() | SI |
| R4 | getCanteenCapacity() | SI |
| R4 | getDishPrice() | SI |
| R4 | getCanteenETA() | PSEUDO |
| R4 | getOpeningHours() | NO |
| R4 | getAllCanteen() | SI |

Tab. 2: Metodi implementati nell'Iterazione 2

perchè ci sembrava utile avere un metodo che ci ritornasse il nome di tutte le mense presenti nel database.

4 ITERAZIONE 3

4.1 Selezione funzioni implementate

In questa fase abbiamo deciso di implementare tutte quelle funzionalità legate all'inserimento e all'aggiornamento di informazioni presenti nel database da parte di un ristoratore.

In particolare si è deciso di trattare lo Use Case **Inserimento dati mensa** dello Use Case Diagram della fase 0 riportato in Fig. 22.

4.2 Class diagram dei metodi per inserimento

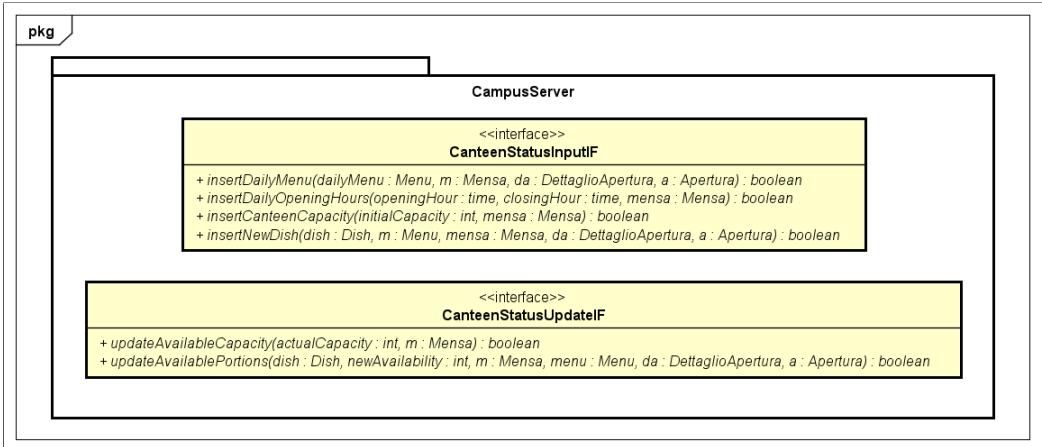


Fig. 34: Class diagram

4.3 Component diagram White-Box

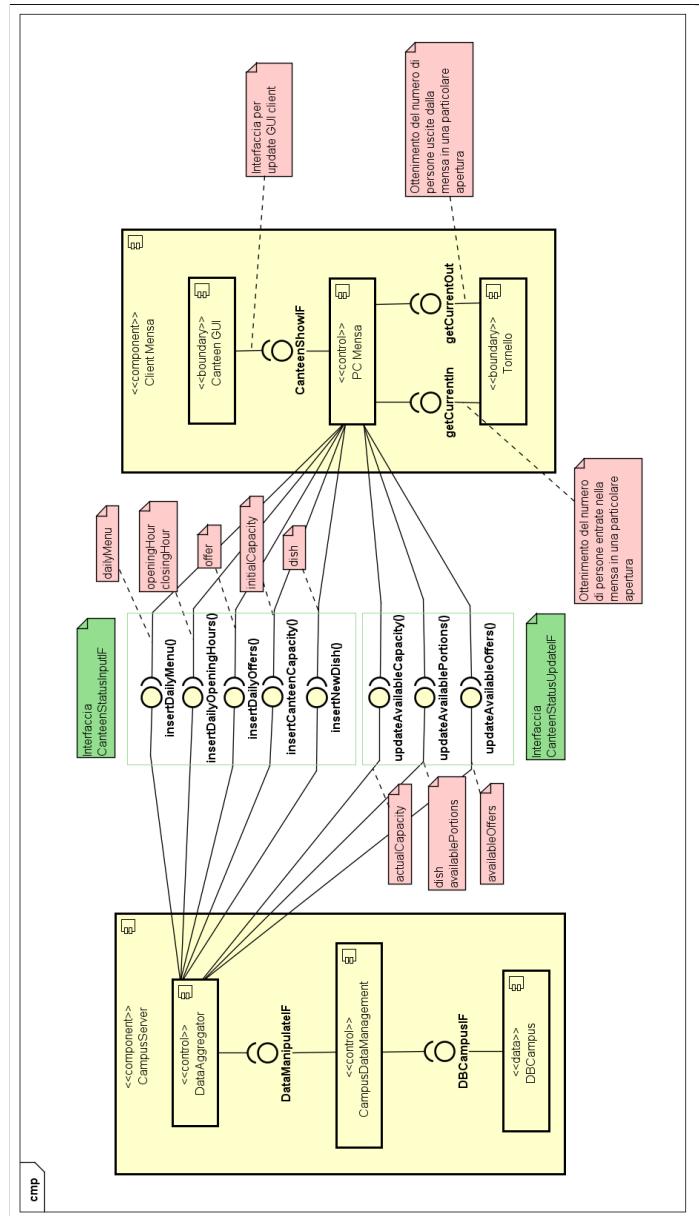


Fig. 35: Component diagram white-box

Il component diagram per la fase 3 mette in evidenza i metodi esposti tramite API dal componente **DataAggregator** sul server al client mensa, con il quale il ristoratore può effettuare l'inserimento dei dati dalla propria mensa.

4.4 Sequence diagram per inserimento

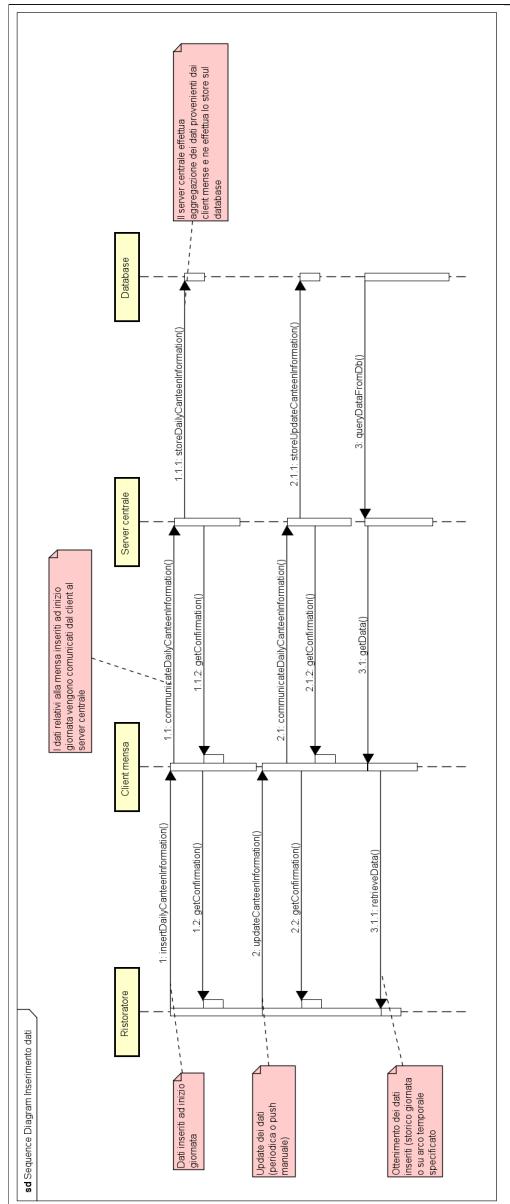


Fig. 36: Composition View

Il sequence diagram mette in evidenza il comportamento del server che interagisce con il ristoratore al momento dell'inserimento dei dati. In particolare, egli interagisce con il server tramite il client sul quale è installato il software per l'inserimento dei dati.

4.5 Casi di test

Durante questa iterazione sono stati realizzati per l'interfaccia di inserimento tre metodi:

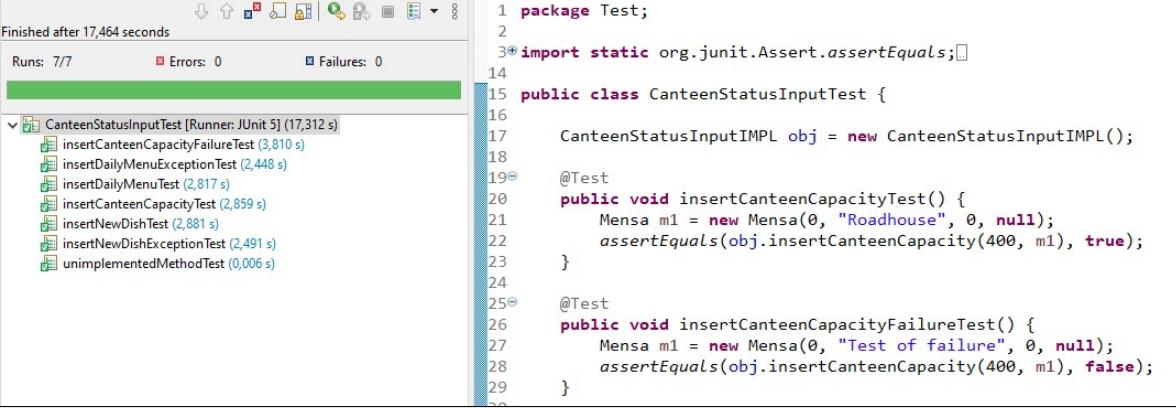
- **insertCanteenCapacity**, che inserisce la capacità di una certa mensa specificata come parametro del metodo;
- **insertDailyMenu**, che inserisce un nuovo menù specificando come parametro del metodo la mensa, il dettaglio apertura e l'apertura interessata dall'operazione;
- **insertNewDish**, che inserisce un nuovo piatto specificando come parametro del metodo la mensa, il dettaglio apertura, l'apertura e il menù interessato dall'operazione;

Per quanto riguarda i casi di test sono stati implementati in totale sette metodi:

- tre metodi, uno per ogni funzione implementata, per testare il loro corretto funzionamento;
- tre metodi, uno per ogni funzione implementata, per testare il comportamento nel caso in cui venga inserito un filtro in input errato (i metodi verificano che venga lanciata l'eccezione prevista);
- Un metodo per la copertura delle altre funzioni non implementate, che verifica se essi ritornano correttamente il valore false.

Come si può vedere in Fig. 37, tutti i test previsti risultano superati e coprono interamente la classe di implementazione delle funzioni realizzate.

In merito invece l'interfaccia di update, sono stati realizzati due metodi:



The screenshot shows the JUnit test results for the `CanteenStatusInputTest`. The summary at the top indicates "Runs: 7/7", "Errors: 0", and "Failures: 0". Below the summary, a tree view shows the test class `CanteenStatusInputTest` and its methods: `insertCanteenCapacityFailureTest`, `insertDailyMenuExceptionTest`, `insertDailyMenuTest`, `insertCanteenCapacityTest`, `insertNewDishTest`, `insertNewDishExceptionTest`, and `unimplementedMethodTest`. Each method is accompanied by its execution time in parentheses.

```

1 package Test;
2
3 import static org.junit.Assert.assertEquals;
4
5 public class CanteenStatusInputTest {
6
7     CanteenStatusInputIMPL obj = new CanteenStatusInputIMPL();
8
9     @Test
10    public void insertCanteenCapacityTest() {
11        Mensa m1 = new Mensa(0, "Roadhouse", 0, null);
12        assertEquals(obj.insertCanteenCapacity(400, m1), true);
13    }
14
15    @Test
16    public void insertCanteenCapacityFailureTest() {
17        Mensa m1 = new Mensa(0, "Test of failure", 0, null);
18        assertEquals(obj.insertCanteenCapacity(400, m1), false);
19    }
20
21}
22
23}
24
25}
26
27}
28
29}

```

Fig. 37: risultato del test sulle funzioni di inserimento implementate

- **updateAvailableCapacity**, che aggiorna la capacità di una certa mensa specificata come parametro del metodo;
- **updateAvailablePortions**, che aggiorna il numero di porzioni disponibili specificando come parametro del metodo la mensa, il dettaglio apertura, l'apertura, il menù e il piatto interessato dall'operazione;

Per quanto riguarda i casi di test sono stati implementati in totale quattro metodi:

- due metodi, uno per ogni funzione implementata, per testare il loro corretto funzionamento;
- due metodi, uno per ogni funzione implementata, per testare il comportamento nel caso in cui venga inserito un filtro in input errato (i metodi verificano che venga lanciata l'eccezione prevista);

Come si può vedere in Fig. 38, tutti i test previsti risultano superati e coprono interamente la classe di implementazione delle funzioni realizzate.

```

1 package Test;
2
3* import static org.junit.Assert.assertEquals;
4
5 public class CanteenStatusUpdateTest {
6
7     CanteenStatusUpdateIMPL obj = new CanteenStatusUpdateIMPL();
8
9     @Test
10    public void updateAvailableCapacityTest() {
11        Mensa m1 = new Mensa(0, "Roadhouse", 0, null);
12        assertEquals(obj.updateAvailableCapacity(400, m1), true);
13    }
14}

```

Fig. 38: risultato del test sulle funzioni di update implementate

4.6 Conclusioni iterazione 3

L’iterazione 3 si conclude con la corretta e funzionante implementazione delle API esposte dalla componente *DataAggregator*. Di seguito è riportata una tabella (Tab. 3) che racchiude i metodi implementati a fine iterazione 2 e correttamente esposti dalla API del nostro sistema.

| Codice | Metodi (API) | Implementato |
|--------|--|--------------|
| R1 | <code>insertDailyOpeningHours()</code> | NO |
| R1 | <code>insertCanteenCapacity()</code> | SI |
| R1 | <code>insertDailyMenu()</code> | SI |
| R1 | <code>insertNewDish()</code> | SI |
| R2 | <code>updateAvailableCapacity()</code> | SI |
| R2 | <code>updateAvailablePortions()</code> | SI |

Tab. 3: Metodi implementati nell’Iterazione 3

5 ITERAZIONE FINALE

5.1 Interfacce grafiche

In questa sezione sono state riportate le schermate delle interfacce grafiche sviluppate appositamente per testare il funzionamento delle API esposte dal server. In particolare, abbiamo deciso di implementare le interfacce grafiche tramite l'utilizzo di Java Swing (si ricorda che queste interfacce rappresentano solamente una prima implementazione e non sono da considerarsi definitive).

In particolare, l'interfaccia grafica dashboard mense potrebbe presentare un campo *stato* ad oggi non mappato, ma che in una futura implementazione potrebbe rappresentare con un flag colorato lo stato di apertura della mensa, ovviamente sfruttando sempre le API esposte dal server. Infatti, le informazioni mostrate sulla dashboard sono ottenute tramite chiamata REST con verbo HTTP GET delle API, in particolare quelle esposte dal componente **CanteenDataMgmt**.

| Nome mensa | Capacità | Posti disponibili |
|----------------------|----------|-------------------|
| I sapori della terra | 100 | 200 |
| Green Canteen | 150 | 130 |
| CampusBBQ | 250 | 200 |
| Roadhouse | 300 | 170 |

Fig. 39: Interfaccia grafica dashboard monitor

In Fig. 39 viene simulata una possibile implementazione del client dashboard, che verrà posizionato all'interno del campus universitario: come già precisato prima, è solo un'interfaccia semplificata che ci permette di verificare e testare il corretto funzionamento delle chiamate API. Infatti, come si può notare, è una semplice pagina statica con la lista delle varie mense disponibili con le relative capacità ed i posti liberi.

Inoltre, per rendere un minimo dinamica l'interfaccia, abbiamo implementato un thread che ogni 10s aggiorna i dati che sta mostrando, di modo da predisporre l'ambiente per il refresh automatico delle informazioni rese disponibili al pubblico. Un possibile sviluppo futuro potrebbe essere, ad esempio, quello di arricchire il contenuto mostrato sfruttando le altre funzioni già sviluppate, come ad esempio quella che permette di visualizzare la lista dei piatti previsti per un certo menù preso come input.

In Fig. 40 invece, viene mostrata l'interfaccia grafica di un possibile client utilizzato dal ristoratore di una certa mensa. Come si può vedere, abbiamo momentaneamente scelto di dare la possibilità di inserire solamente la capacità di una mensa, un nuovo menù per una certa apertura e un nuovo piatto per un certo menù, ma ci sono altri metodi già disponibili che danno la possibilità di effettuare ulteriori operazioni (vedi Iterazione 3).

E' importante sottolineare che, all'inserimento di un nuovo elemento nel database, l'interfaccia restituisce all'utente un feedback sull'esito dell'operazione richiesta: nel caso in cui tutto sia andato nel verso giusto viene mostrata la correttezza dell'inserimento, mentre in caso contrario viene riportato un messaggio d'errore.

Questa funzionalità e questo modo di agire sono da subito risultati fondamentali per tutti i membri del gruppo, in quanto si è dimostrato vitale essere sempre al corrente della situazione attuale e della presenza di eventuali orrori o anomalie.

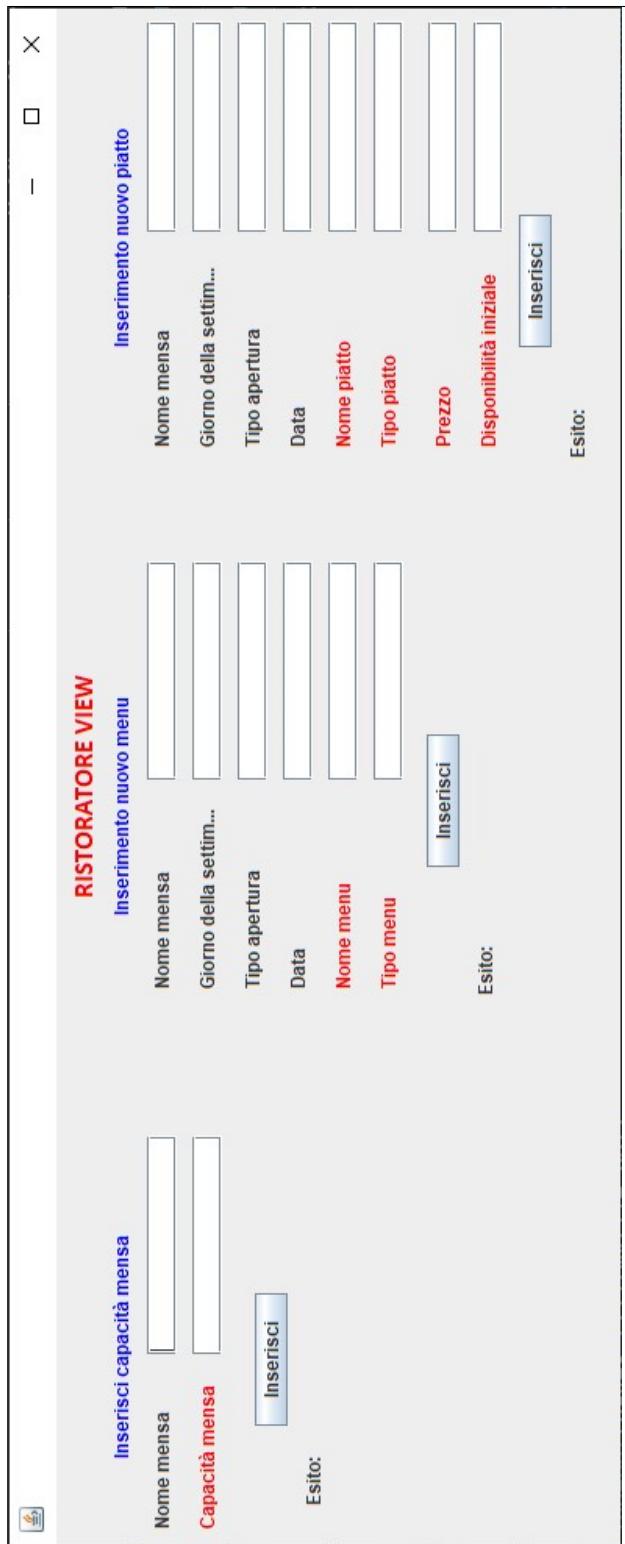


Fig. 40: Interfaccia grafica client ristoratore

5.2 Conclusioni finali

Alla conclusione dell'iterazione finale, il progetto risulta essere composto dalle funzionalità di visualizzazione dei dati sulle dashboard statiche (monitor) e di inserimento ed aggiornamento da parte dei ristoratori tramite appositi client e software dedicato. Tutte queste funzioni sono state implementate tenendo sempre in considerazione il *processo di sviluppo agile*, suddividendo le fasi di implementazione in funzione del particolare componente da realizzare.

Tutto ciò ha permesso al team di sviluppo di operare in modalità cooperativa durante la fase 0, ovvero quella della stesura delle idee di progetto e successivi brainstorming, e di proseguire in maniera agile in tutte le fasi successive. In particolare, ogni iterazione ha previsto un momento iniziale di ideazione e progettazione, seguito da una fase di definizione delle interfacce e successiva implementazione delle stesse, eventualmente rivedendo le decisioni prese nelle precedenti iterazioni, raffinando quindi iterativamente il prodotto.

Prima di concludere ogni fase, sono stati effettuati dei test, sia statici che dinamici, allo scopo di garantire il corretto funzionamento dei metodi realizzati e delle API esposte.

5.2.1 API esposte

Lo scopo di questo progetto è stato anche quello di fornire una versione alternativa al caso di sviluppo CoCoME che potesse trarre vantaggio dall'utilizzo di tecnologie di comunicazione tra sistemi informativi che non fossero quelle del message-passing fornite da Java (ovvero JMS, Java Messaging System) bensì le più moderne soluzioni basate sul paradigma REST ed implementate tramite l'esposizione di API.

La progettazione e l'implementazione delle API è stato infatti un punto chiave nel

processo di sviluppo Agile su cui si è basato questo caso di studio, in quanto per tutti i metodi è stata prevista una API con cui esporre le funzionalità messe a disposizione dal metodo stesso. Ciò rende la fruizione dei metodi molto flessibile anche tenendo conto di futuri sviluppi ed aggiunte, separando quindi le funzionalità in componenti sul server e gestendoli indipendentemente l'uno dagli altri.

5.2.2 Repository GitHub

Tutto il codice relativo al progetto esposto è disponibile sotto licenza MIT alla seguente *repository GitHub*, nella quale è possibile accedere a tutto il codice, sia lato server che lato client. Inoltre, è possibile sfruttare le funzionalità di "Wiki" ed "Issues" per implementare i successivi sviluppi futuri di questo progetto.

5.2.3 Sviluppi futuri

Lo stato attuale di implementazione del progetto prevede solamente che alcune delle funzionalità siano correttamente funzionanti, segnatamente le funzionalità di visualizzazione delle dashboard e di inserimento da parte del ristoratore delle informazioni relative alla propria mensa.

Gli sviluppi futuri dovranno certamente riguardare la progettazione e l'implementazione delle componenti di autenticazione al server universitario ed accesso al server di gestione delle carte prepagate per il pagamento dei pasti da parte degli utenti.

Altro potenziale punto di partenza potrebbe essere lo sviluppo di un applicativo mobile per Android attraverso cui gli utenti del campus possono visualizzare sia le informazioni presenti sulle dashboard statiche che informazioni personalizzate, anche legate a proprie preferenze personali (piatti e mense preferite, fasce orarie prioritarie, ecc.)

5.3 Manuale utente

Di seguito riportiamo il manuale utente che servirà per utilizzare il prodotto software su qualunque sistema operativo (Windows o MacOS).

Abbiamo innanzitutto esportato dall'IDE Eclipse il runnable jar di modo che esso potesse essere utilizzabile mediante cmd locale. Una volta accertati che il tutto funzionasse correttamente, abbiamo generato l'eseguibile utilizzando Launch4j creando tre file diversi:

- `CampusServer.exe`, che avvia il server;
- `DashboardGUI.exe`, che avvia la GUI statica che rimane in esecuzione sulle dashboard del campus;
- `ClientRistoratoreGUI.exe`, che avvia la GUI dinamica che viene utilizzata dal ristoratore

E' necessario avviare dapprima il file `CampusServer.exe` per mandare in esecuzione il server e, in seguito, sarà possibile utilizzare l'eseguibile `DashboardGUI.exe` e l'eseguibile `ClientRistoratoreGUI.exe` per usufruire delle funzionalità dell'applicazione progettata.

6 STAN4J

Quality Report



Creation Date 2021-01-21

Level of Detail Class

ProgettoInfo3B-now-using-Spring

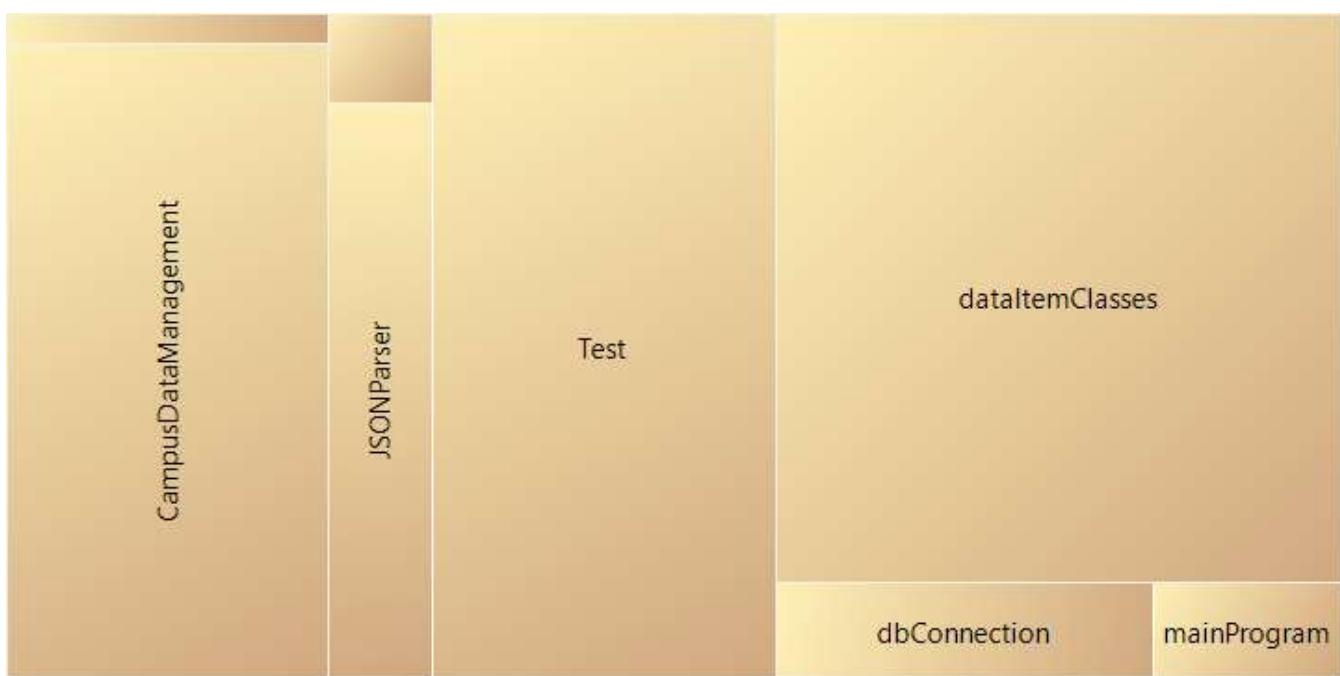
Library Dependency Graph

📁/ProgettoInfo3B-now-using-Spring/src/main/java

📁/ProgettoInfo3B-now-using-Spring/src/main/resources

📁/ProgettoInfo3B-now-using-Spring/src/test/java

Treemap Overview



Metrics Summary

| Metric | Value |
|---|-------|
| Number of Libraries | 3 |
| Number of Packages | 8 |
| Number of Top Level Classes | 20 |
| Average Number of Top Level Classes per Package | 2.50 |
| Average Number of Member Classes per Class | 0 |
| Average Number of Methods per Class | 8.10 |
| Average Number of Fields per Class | 2.35 |
| Estimated Lines of Code | 1025 |
| Estimated Lines of Code per Top Level Class | 51.25 |

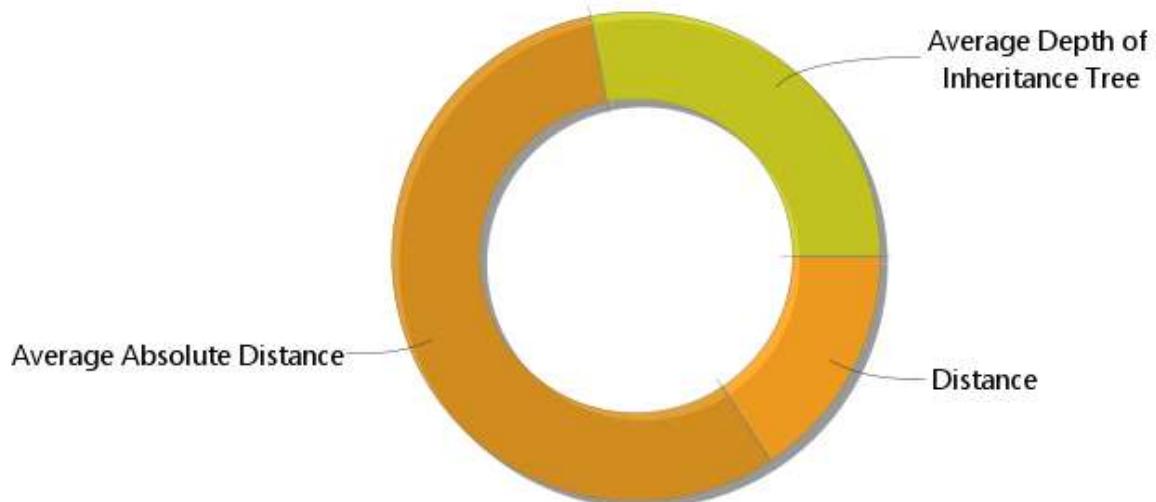
| | |
|--|--------|
| Average Cyclomatic Complexity | 1.07 |
| Fat for Library Dependencies | 0 |
| Fat for Flat Package Dependencies | 5 |
| Fat for Top Level Class Dependencies | 52 |
| Tangled for Library Dependencies | 0% |
| Average Component Dependency between Libraries | 0% |
| Average Component Dependency between Packages | 12.50% |
| Average Component Dependency between Units | 21.32% |
| Average Distance | -0.10 |
| Average Absolute Distance | 0.65 |
| Average Weighted Methods per Class | 8.65 |
| Average Depth of Inheritance Tree | 0.85 |
| Average Number of Children | 0 |
| Average Coupling between Objects | n/a |
| Average Response for a Class | n/a |
| Average Lack of Cohesion in Methods | n/a |

Top Violations (7 of 7)

| Artifact | Metric | Value |
|---------------------------------|--------|-------|
| ProgettoInfo3B-now-using-Spring | D | 0.65 |
| ProgettoInfo3B-now-using-Spring | DIT | 0.85 |
| dataItemClasses | D | -1 |
| JSONParser | D | -1 |
| dbConnection | D | -1 |
| Authentication | D | 1 |
| CardMgmt | D | 1 |

Pollution Chart

Pollution 1.37



Violations by Metric

Distance

| Artifact | Value |
|-----------------|-------|
| dataItemClasses | -1 |
| JSONParser | -1 |
| dbConnection | -1 |
| Authentication | 1 |
| CardMgmt | 1 |

Average Absolute Distance

| Artifact | Value |
|---------------------------------|-------|
| ProgettoInfo3B-now-using-Spring | 0.65 |

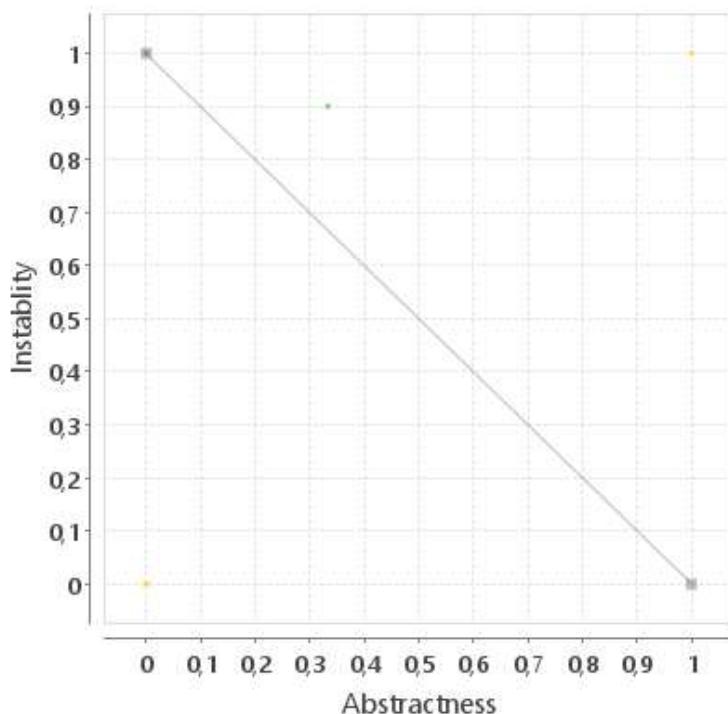
Average Depth of Inheritance Tree

| Artifact | Value |
|---------------------------------|-------|
| ProgettoInfo3B-now-using-Spring | 0.85 |

Design Tangles

There are no design tangles.

Package Distance Chart



Metric Ratings

Count Metrics

| Metric | Rating | Linear |
|-----------------------------|--|--------|
| Number of Top Level Classes | <div><div style="width: 100%;">20 40 60 80</div></div> | ✓ |
| Number of Methods | <div><div style="width: 100%;">25 50 100 200</div></div> | ✓ |



Complexity Metrics

| Metric | Rating | Linear |
|--|--------|--------|
| Cyclomatic Complexity | | ✓ |
| Fat | | ✓ |
| Fat | | ✓ |
| Fat | | ✓ |
| Tangled | | ✓ |
| Tangled for Library Dependencies | | ✓ |
| Average Component Dependency between Libraries | | ✓ |
| Average Component Dependency between Packages | | ✓ |

Robert C. Martin Metrics

| Metric | Rating | Linear |
|---------------------------|--------|--------|
| Distance | | ✓ |
| Average Absolute Distance | | ✓ |

Chidamber & Kemerer Metrics

| Metric | Rating | Linear |
|-----------------------------------|--------|--------|
| Weighted Methods per Class | | ✓ |
| Depth of Inheritance Tree | | ✓ |
| Average Depth of Inheritance Tree | | ✓ |
| Coupling between Objects | | ✓ |
| Response for a Class | | ✓ |