



**UNIVERSITÀ  
DEGLI STUDI  
DI BERGAMO**

# Documentazione Progetto di Progettazione, Algoritmi e Computabilità

Pellegrinelli Nico	1065869
Pellegrinelli Sean	1065868

Anno Accademico 2022/2023

## Sommario

Iterazione 0.....	3
Introduzione al sistema .....	3
Requisiti funzionali .....	4
Topologia del sistema.....	7
Tool chain .....	8
Iterazione 1.....	9
Casi d'uso.....	9
Early Architecture .....	12
Analisi dinamica: JUnit.....	15
Analisi statica: STAN4J .....	17
API esposte .....	19
Iterazione 2.....	21
Casi d'uso.....	21
Evoluzione dell'architettura .....	23
Design dell'algoritmo.....	25
Complessità temporale.....	28
Analisi dinamica: JUnit.....	30
Analisi statica: STAN4J .....	32
API esposte .....	35

## Iterazione 0

### Introduzione al sistema

Il sistema realizzato ha lo scopo di gestire una comunità online per lo scambio di libri usati. L'idea è che se un utente ha un libro che non desidera più, può offrirlo ad altri utenti in cambio di un certo numero di **token**. Un utente intenzionato ad ottenere un libro usato dovrà "pagarlo" in token, il prezzo dipende dalla tipologia del libro, dalla sua condizione e dalla distanza che gli altri utenti devono percorrere per consegnare il libro. Infatti, la consegna dei libri viene effettuata dagli utenti stessi: ciascuno di essi indica quanti km è disposto a percorrere per trasportare i libri, e quando un libro viene acquistato il sistema identifica quali utenti (al massimo 3 incluso il "venditore") dovranno consegnarlo e li notifica. Il "venditore", quindi, non ottiene tutti i token versati dal "compratore" perché una porzione del prezzo viene elargita agli utenti selezionati dal sistema per la consegna (il numero di token che questi ottengono è proporzionale alla distanza che devono percorrere).

Gli utenti usufruiranno del servizio di scambio dei libri attraverso un'applicazione mobile che permetterà loro di:

- Registrarsi e gestire l'account.
- Mettere a disposizione dei libri usati.
- Comprare libri usati usando i suoi token.
- Modificare le condizioni di un libro.
- Acquistare token con valuta reale.
- Indicare i propri "libri desiderati" in modo che il sistema notifichi quando uno di tali libri è disponibile all'acquisto.

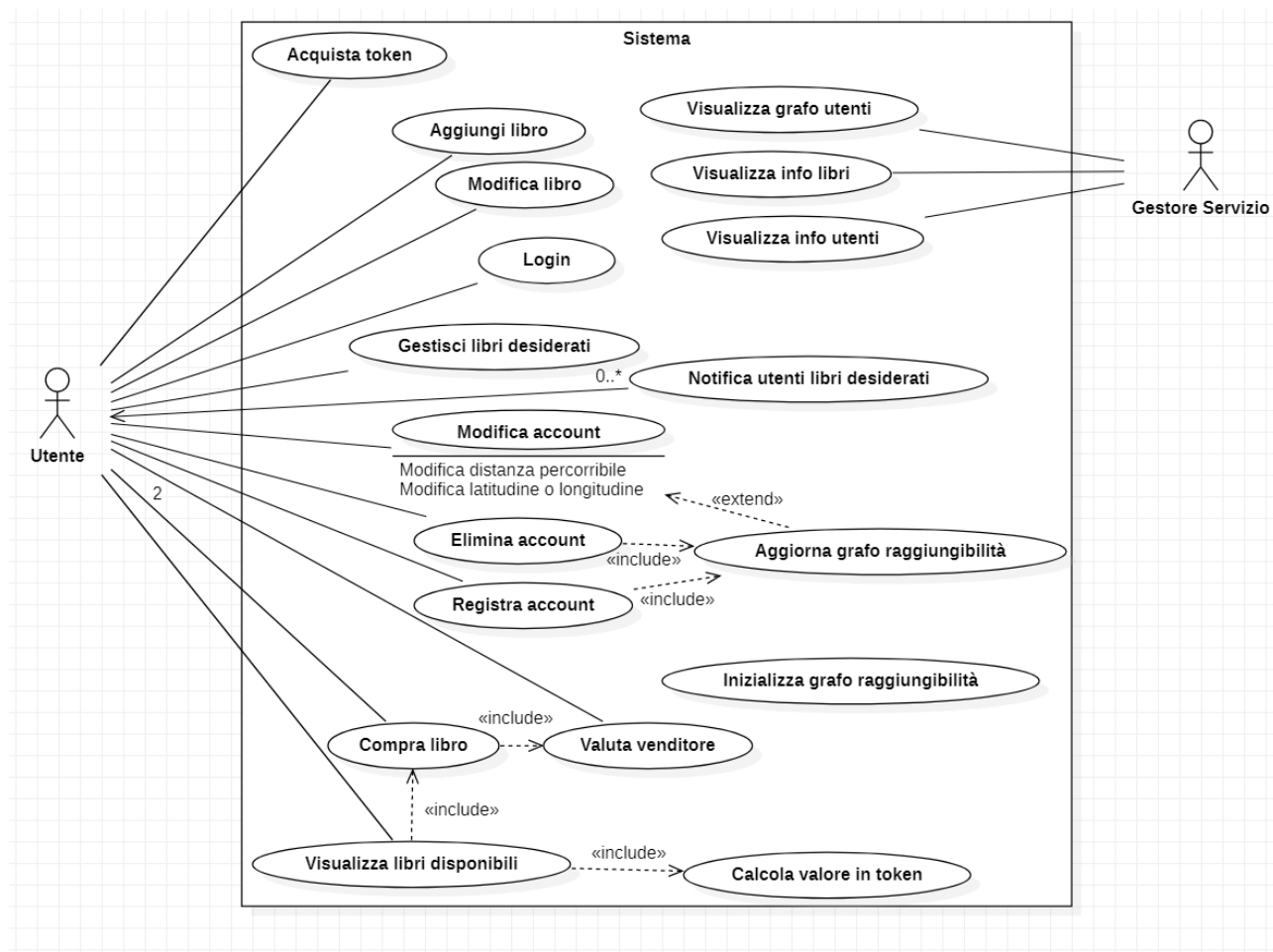
Il gestore di questo servizio si interfaccia al sistema con un'applicazione desktop dedicata che gli permette di:

- Visualizzare le statistiche sugli utenti (come chi ha più token, chi ha acquistato più libri etc.).
- Visualizzare le statistiche sui libri (quali sono i più disponibili, i più desiderati, i più scambiati etc.).
- Visualizzare il **grafo di raggiungibilità degli utenti**: ogni vertice rappresenta un utente, un arco dall'utente x all'utente y indica che l'utente y è sufficientemente vicino a x per potergli consegnare un libro. Il grafo è diretto perché se x può raggiungere y non è detto che sia vero il contrario (visto che ogni utente imposta la distanza massima che è disposto a percorrere).

Lo scopo di questo progetto non è la realizzazione del sistema nella sua interezza, verranno infatti solo implementate le componenti fondamentali lato server seguendo la metodologia di sviluppo software AMDD (Agile Model Driven Development).

## Requisiti funzionali

I requisiti funzionali del sistema sono rappresentati nel seguente use case diagram:



In ogni iterazione del processo di sviluppo AMDD verranno scelti ed implementati alcuni di questi requisiti. Ovviamente è importante sviluppare prima i requisiti più importanti per il funzionamento del sistema, per questa ragione i requisiti sono stati divisi in cinque livelli priorità, rappresentati nella matrice che segue insieme ad altre importanti informazioni sui requisiti.

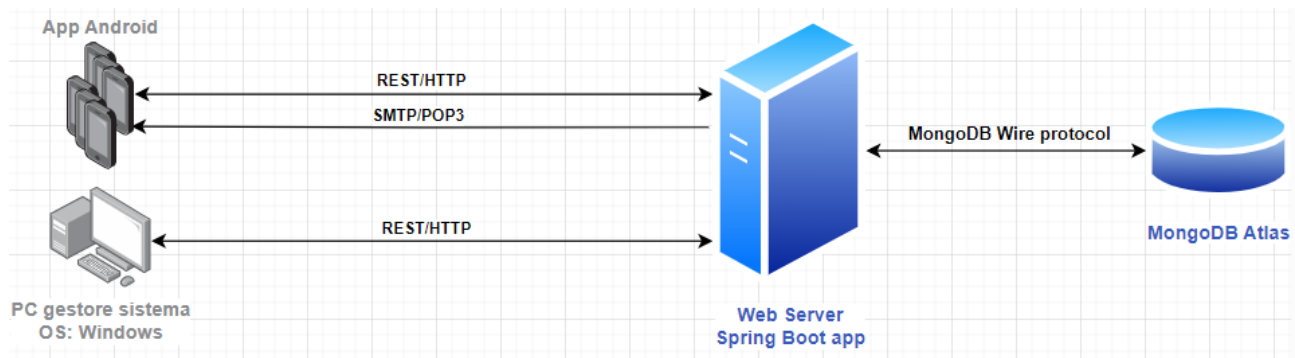
Nome – (id)	Priorità	Descrizione	Gruppo di casi d'uso	Fonte	Requisito Padre	Requisiti figli
<b>Registra account (1)</b>	Molto alta	Registrazione di un nuovo account con nome utente e password	Gestione account	AMDD iterazione 0 (25/01/2023)		2
<b>Aggiorna grafo raggiungibilità (2)</b>	Molto alta	Aggiunta di un nodo con relativi archi al grafo raggiungibilità utenti	Gestione grafo	AMDD iterazione 0 (25/01/2023)	1, 13, 14	
<b>Inizializza grafo raggiungibilità (18)</b>	Molto alta	Crea il grafo raggiungibilità utenti a partire dai dati presenti sul database all'avvio del server	Inizializzazione	Revisione iterazione 0 (29/01/2023)		

<b>Login (17)</b>	Molto alta	Accesso di un utente registrato	Gestione account	Revisione iterazione 0 (28/01/2023)		
<b>Aggiungi libro (3)</b>	alta	Aggiunta di un libro disponibile da parte di un utente	Gestione libri utente	AMDD iterazione 0 (25/01/2023)		
<b>Visualizza libri disponibili (4)</b>	alta	Visualizzazione dei libri che possono essere ottenuti da un utente (al più distanti tre passi nel grafo raggiungibilità utenti)	Acquisto libro	AMDD iterazione 0 (25/01/2023)		5, 6
<b>Calcola valore token (5)</b>	alta	Calcolo del valore in token di un libro in funzione di che utente lo desidera	Acquisto libro	AMDD iterazione 0 (25/01/2023)	4	
<b>Compra libro (6)</b>	alta	Acquisto da parte di un utente, tramite token, del libro di un altro utente	Acquisto libro	AMDD iterazione 0 (25/01/2023)	4	15
<b>Visualizza grafo raggiungibilità (7)</b>	media	Visualizzazione da parte del gestore del servizio del grafo raggiungibilità utenti	Visualizzazione informazioni di profiling	AMDD iterazione 0 (25/01/2023)		
<b>Visualizza info utenti (8)</b>	media	Visualizzazione da parte del gestore di informazioni riguardanti gli utenti	Visualizzazione informazioni di profiling	AMDD iterazione 0 (25/01/2023)		
<b>Visualizza info libri (9)</b>	media	Visualizzazione da parte del gestore di informazioni riguardanti i libri	Visualizzazione informazioni di profiling	AMDD iterazione 0 (25/01/2023)		
<b>Acquista token (10)</b>	media	Acquisto da parte di un utente di token (con denaro reale)	Gestione account	AMDD iterazione 0 (25/01/2023)		
<b>Modifica libro (11)</b>	bassa	Modifica da parte di un utente delle informazioni di un suo libro	Gestione libri utente	AMDD iterazione 0 (25/01/2023)		
<b>Gestisci libri desiderati (12)</b>	bassa	Aggiunta o modifica da parte di un utente dei libri desiderati, ovvero quelli che più desidera	Gestione libri utente	AMDD iterazione 0 (25/01/2023)		
<b>Elimina account (13)</b>	bassa	Eliminazione di un account	Gestione account	AMDD iterazione 0 (25/01/2023)		2

<b>Modifica account (14)</b>	bassa	Modifica delle informazioni di un utente, come posizione e massima distanza percorribile	Gestione account	AMDD iterazione 0 (25/01/2023)		2
<b>Valuta venditore (15)</b>	bassa	Valutazione di un venditore a seguito di un acquisto di libro	Acquisto libro	AMDD iterazione 0 (25/01/2023)	6	
<b>Notifica utenti libri desiderati (16)</b>	Molto bassa	Notifica tramite e-mail la disponibilità di un libro tra quelli indicati come desiderati	/	AMDD iterazione 0 (25/01/2023)		

## Topologia del sistema

La topologia del sistema è rappresentata dal seguente schema in notazione libera:



Come si può notare la struttura è quella tipica di un sistema **Client-Server**. I Client sono di due tipi:

- L'applicazione desktop sul PC del gestore del servizio, che contatta il server tramite REST/HTTP per ricevere informazioni sui libri ed utenti presenti nel sistema.
- L'applicazione Android utilizzabile dagli utenti per usufruire dei servizi offerti dal sistema. La comunicazione avviene tramite SMTP/POP3 per:
  - avvisare gli utenti che un libro nella lista dei "libri desiderati" è disponibile;
  - avvisare gli utenti scelti per la consegna di un libro.

Il resto delle comunicazioni viene gestito tramite REST/HTTP.

Nelle comunicazioni REST/HTTP i dati vengono scambiati in formato JSON.

Il server esegue un'applicazione Java realizzata mediante il framework spring boot che, oltre a comunicare con i client per soddisfare le loro richieste, comunica con un cloud database (ovvero MongoDB Atlas) dove sono salvati in modo persistente i dati relativi agli utenti registrati.

Oltre allo stile architetturale Client-Server viene anche adottato l'approccio architetturale a **microservizi**; infatti, le funzionalità offerte dal web server non sono altro che un insieme di microservizi realizzati attraverso il framework Spring.

## Tool chain

Lo sviluppo del progetto è supportato dai seguenti tool:

- **Eclipse**: IDE per lo sviluppo software utilizzato per la scrittura di tutto il codice (**Java**) del sistema. Alcuni degli altri tool utilizzati sono integrati con Eclipse.
- **Spring boot**: framework java per lo sviluppo di applicazioni web basate sui microservizi.
- **MongoDB Atlas**: cloud database non relazionale (i dati sono salvati come documenti JSON). Il cluster utilizzato si appoggia ad un server **AWS**.
- **JAutoDoc**: plugin di Eclipse per la generazione dei **Javadoc** che permettono di generare la documentazione del codice java a partire dai commenti del codice.
- **JUnit 4**: framework per i test di unità in Java.
- **Eclemma**: plugin di Eclipse per la verifica della copertura del codice.
- **JGraphT**: libreria Java per la modellazione di grafi.
- **STAN4J**: software per l'analisi statica di progetti Java.
- **GitHub**: Piattaforma per il versionamento basata su Git. È stato utilizzato anche GitHub Desktop che permette di interagire dal proprio PC con GitHub utilizzando una semplice GUI.
- **StarUML**: software per la creazione di modelli UML.
- **diagrams.net**: software online per la creazione di modelli in notazione libera.



## Iterazione 1

### Casi d'uso

Nell'iterazione 1 sono stati scelti e implementati i primi casi d'uso dallo stack delle priorità:

- Registra account (Id: 1)
- Aggiorna grafo raggiungibilità (Id: 2)
- Inizializza grafo raggiungibilità (Id: 18)
- Login (Id: 17)
- Aggiungi libro (Id: 3)

In relazione al component diagram realizzato nell'early architecture design (si veda la sezione successiva) si opererà sui subsystem **Gestore Grafo** e **Gestore Account** e sul component **Inizializzatore**. Di seguito vengono descritti in dettaglio partendo da una vista complessiva del sistema, comprendente quindi anche i client, anche se il progetto si concentrerà inizialmente sul solo sviluppo del codice lato server.

<b>Nome: Registra account</b>	
<b>Id:</b>	1
<b>Priorità:</b>	Molto alta
<b>Descrizione:</b>	L'utente inserisce nel client le informazioni richieste, tra le quali e-mail e password, e viene registrato un nuovo account nel sistema
<b>Pre-condizioni:</b>	Nessuna
<b>Post-condizioni:</b>	Viene registrato nel sistema il nuovo utente
<b>Situazioni di errore:</b>	L'utente è già registrato nel sistema, ovvero è già presente nel sistema un utente con la e-mail indicata
<b>Stato del sistema in caso di errore:</b>	L'utente viene avvisato con un messaggio di errore
<b>Attori:</b>	Utente, Sistema
<b>Trigger:</b>	L'utente seleziona l'opzione di registrazione dal client
<b>Processo standard:</b>	(1) L'utente seleziona l'opzione di registrazione dal client (2) L'utente inserisce le seguenti informazioni: <ul style="list-style-type: none"><li>• E-mail</li><li>• Password</li><li>• Latitudine e Longitudine</li><li>• Massima distanza percorribile</li><li>• I libri desiderati</li></ul> (3) L'utente viene registrato nel sistema (4) Aggiorna il grafo di raggiungibilità (use case 2)

**Nome: Aggiorna grafo raggiungibilità**

<b>Id:</b>	2
<b>Priorità:</b>	Molto alta
<b>Descrizione:</b>	Viene aggiornato il grafo di raggiungibilità, aggiungendo nuovi nodi e/o archi
<b>Pre-condizioni:</b>	Grafo raggiungibilità utenti non consistente rispetto alle informazioni degli utenti
<b>Post-condizioni:</b>	Grafo raggiungibilità utenti aggiornato e consistente
<b>Situazioni di errore:</b>	Nessuna
<b>Attori:</b>	Sistema
<b>Trigger:</b>	Aggiunta o eliminazione di un utente oppure modifica della massima distanza percorribile da un utente o della sua posizione (latitudine e/o longitudine)
<b>Processo standard</b> <b>AGGIUNTA UTENTE:</b>	(1) Viene aggiunto un vertice corrispondente al nuovo utente (2) Per ogni utente Y nel sistema viene calcolata la distanza rispetto all'utente aggiunto X e: <ul style="list-style-type: none"><li>• se è minore o uguale alla massima distanza percorribile da Y viene aggiunto un arco diretto da X a Y</li><li>• se è minore o uguale alla massima distanza percorribile da X viene aggiunto un arco diretto da Y a X</li></ul>
<b>Processo standard</b> <b>ELIMINAZIONE</b> <b>UTENTE:</b>	(1) Vengono eliminati tutti gli archi entranti o uscenti dal vertice il cui contenuto è pari all'id dell'utente eliminato (2) Viene eliminato tale vertice
<b>Processo standard</b> <b>MODIFICA UTENTE:</b>	(1) Per ogni utente Y nel sistema viene ricalcolata la distanza rispetto all'utente modificato X e: <ul style="list-style-type: none"><li>• se è minore o uguale alla massima distanza percorribile da Y viene aggiunto un arco diretto da X a Y</li><li>• se è minore o uguale alla massima distanza percorribile da X viene aggiunto un arco diretto da Y a X</li></ul>

**Nome: Inizializza grafo raggiungibilità**

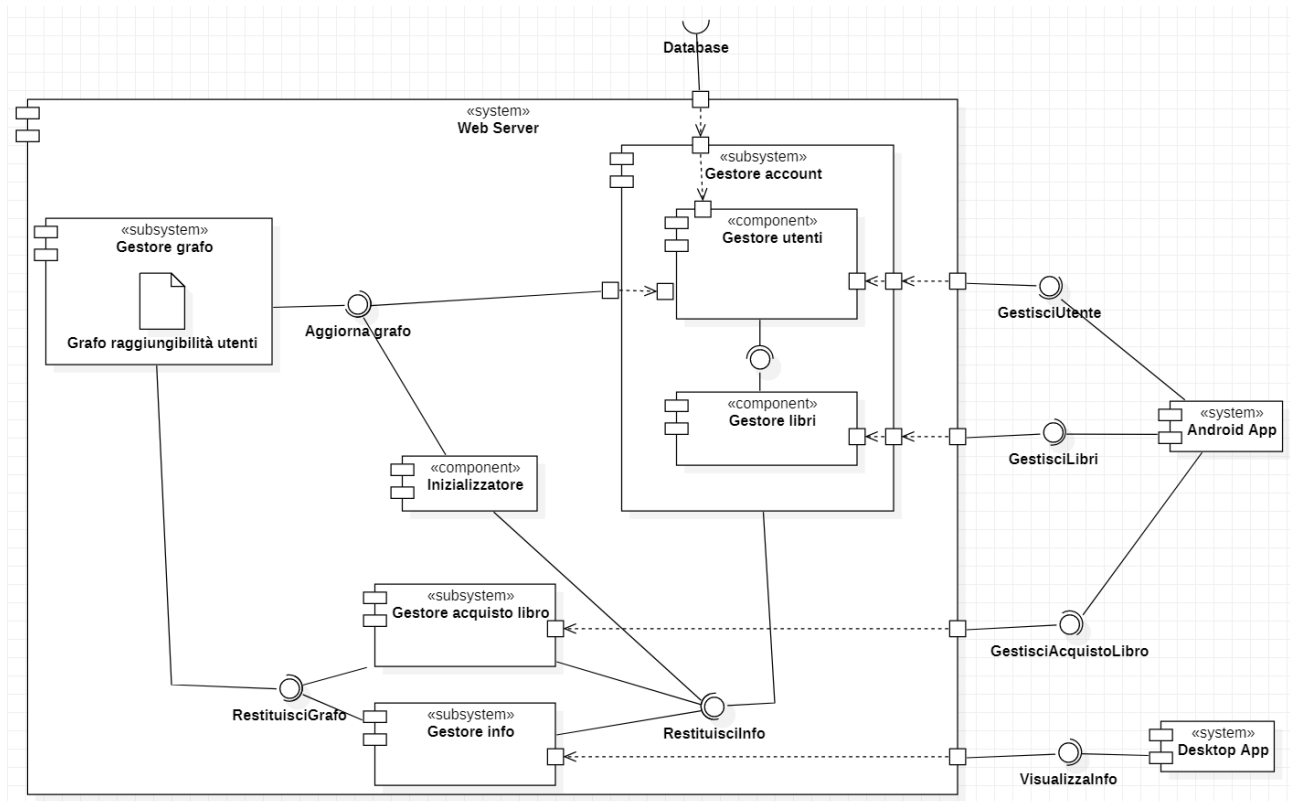
<b>Id:</b>	18
<b>Priorità:</b>	Molto alta
<b>Descrizione:</b>	All'avvio del server i dati sugli utenti registrati (salvati sul database remoto) vengono utilizzati per costruire il grafo raggiungibilità utenti
<b>Pre-condizioni:</b>	L'applicazione server non è stata avviata
<b>Post-condizioni:</b>	Il server è avviato e contiene il grafo raggiungibilità utenti
<b>Situazioni di errore:</b>	Nessuna
<b>Attori:</b>	Sistema
<b>Trigger:</b>	L'applicazione server viene avviata
<b>Processo standard:</b>	(1) Il sistema ottiene dal database gli utenti registrati (2) Per ogni utente il sistema aggiunge un vertice al grafo raggiungibilità utenti come descritto nel caso d'uso "Aggiorna grafo raggiungibilità" (Id: 2)

<b>Nome:</b>	<b>Login</b>
<b>Id:</b>	17
<b>Priorità:</b>	Molto alta
<b>Descrizione:</b>	L'utente inserisce nel client e-mail e password ed accede all'applicazione
<b>Pre-condizioni:</b>	L'utente ha già effettuato la registrazione
<b>Post-condizioni:</b>	L'utente accede all'applicazione
<b>Situazioni di errore:</b>	L'utente non è registrato nel sistema, ovvero non è presente nel sistema un utente con la e-mail indicata, oppure la password è errata
<b>Stato del sistema in caso di errore:</b>	L'utente viene avvisato con un messaggio di errore
<b>Attori:</b>	Utente, Sistema
<b>Trigger:</b>	L'utente seleziona l'opzione di login dal client
<b>Processo standard:</b>	(1) L'utente seleziona l'opzione di login dal client (2) L'utente inserisce le seguenti informazioni: <ul style="list-style-type: none"> <li>• E-mail</li> <li>• Password</li> </ul> (3) Il login avviene correttamente e l'utente può utilizzare l'applicazione

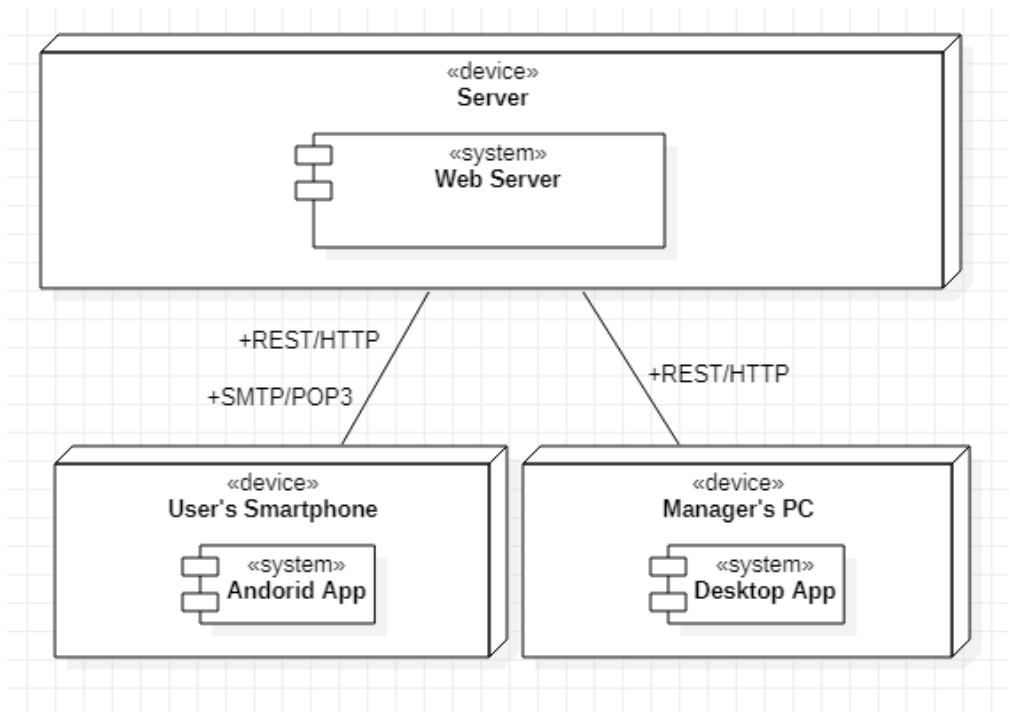
<b>Nome:</b>	<b>Aggiungi libro</b>
<b>Id:</b>	3
<b>Priorità:</b>	Alta
<b>Descrizione:</b>	L'utente inserisce nell'applicazione le informazioni riguardanti il libro che vuole aggiungere nella sua lista di libri
<b>Pre-condizioni:</b>	L'utente è registrato e ha effettuato l'accesso
<b>Post-condizioni:</b>	Viene aggiunto un libro alla sua lista di libri
<b>Situazioni di errore:</b>	Nessuna
<b>Attori:</b>	Utente, Sistema
<b>Trigger:</b>	L'utente seleziona l'opzione di aggiunta libro dal client
<b>Processo standard:</b>	(1) L'utente seleziona l'opzione di aggiunta libro dal client (2) L'utente inserisce le seguenti informazioni: <ul style="list-style-type: none"> <li>• Titolo</li> <li>• Numero pagine</li> <li>• Data di pubblicazione</li> <li>• Condizioni del libro</li> <li>• Se presenta o meno illustrazioni</li> </ul> (3) Il libro viene aggiunto alla lista di libri dell'utente

## Early Architecture

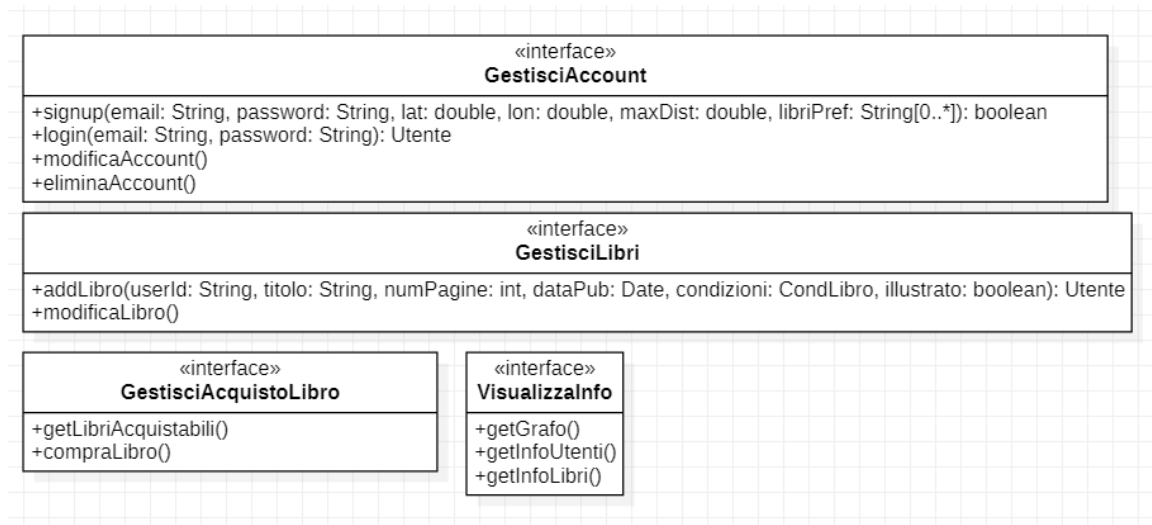
Di seguito viene presentata un'iniziale architettura del sistema ad alto livello (early architecture design). La decomposizione del sistema in sottosistemi e componenti è rappresentata nel seguente component diagram:



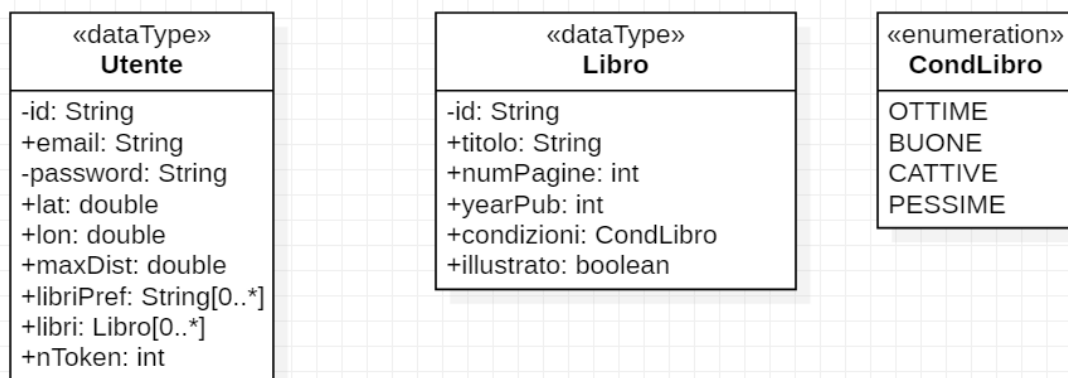
Come si può notare, il focus iniziale è sulle funzionalità del web server, mentre i sistemi che rappresentano i client sono trascurati. I componenti sono mappati sui device fisici secondo il seguente deployment diagram:



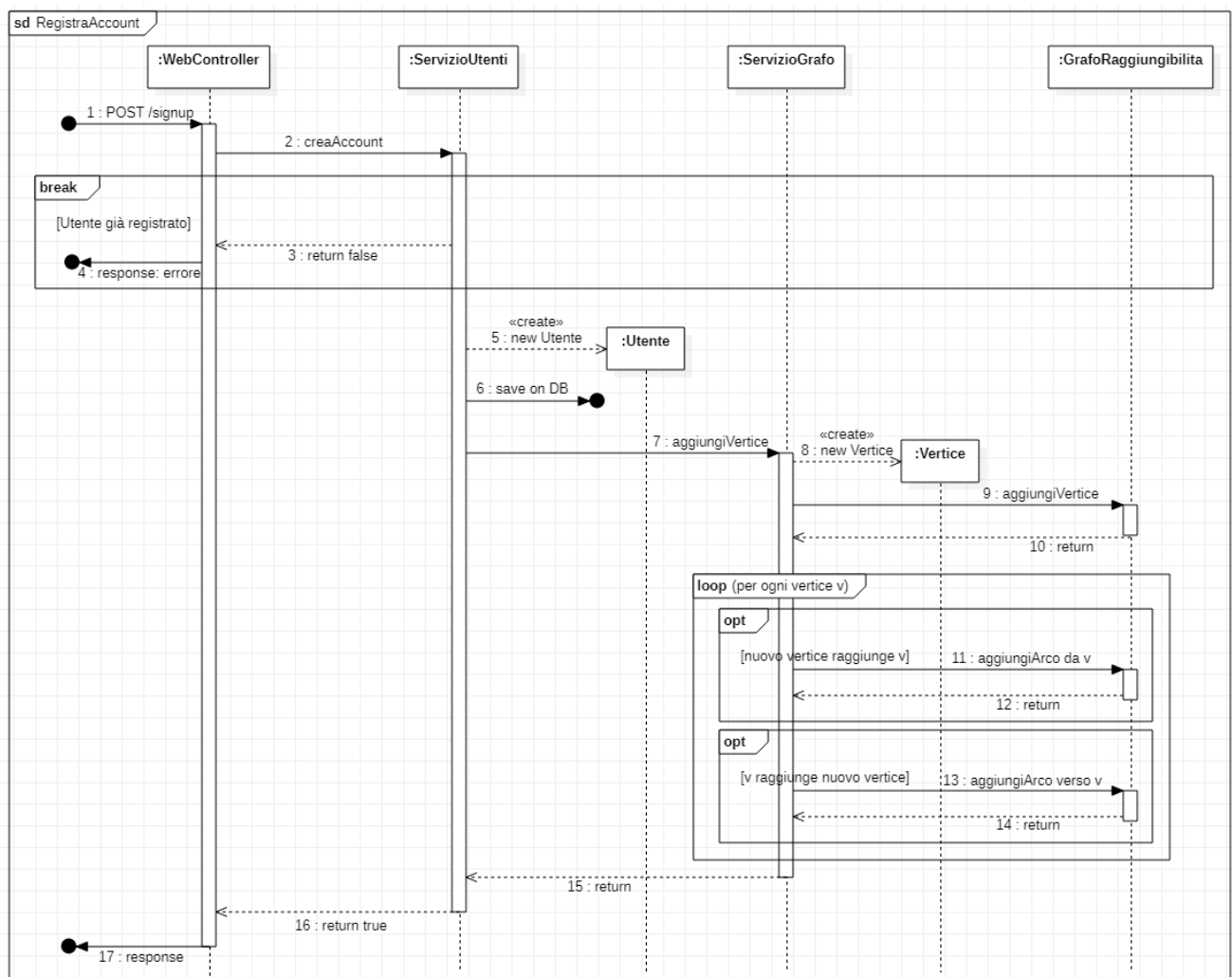
Proseguendo con l'early architecture design, il seguente class diagram mostra una prima definizione delle interfacce esposte dal Web Server:



Il seguente class diagram, invece, mostra i data types del sistema:



Per quanto riguarda le interfacce, sono state definite in maniera precisa solo quelle relative ai casi d'uso che sono stati implementati in questa iterazione. Le altre sono generiche e verranno specificate nelle successive iterazioni. Infine, è stata modellata una vista dinamica relativa ai casi d'uso **Registra Account** e **Aggiorna grafo raggiungibilità**.

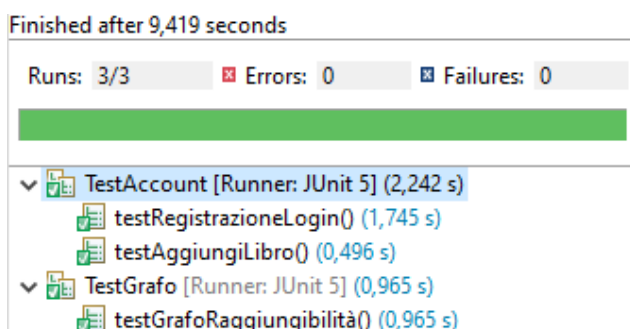


## Analisi dinamica: JUnit

Per testare la corretta implementazione dei casi d'uso scelti per l'iterazione 1 sono state realizzate due classi di test: TestAccount (con i metodi testRegistrazioneLogin e testAggiungiLibro) e testGrafo (con il metodo testGrafoRaggiungibilità). Segue la descrizione dei metodi di test, realizzati con SpringBootTest e JUnit 4.






















- **testRegistrazioneLogin:**
  - Viene creato un oggetto di tipo DtoUtente utilizzato come contenuto di una successiva richiesta http (url: `http://localhost:8080/signup`) eseguita da un oggetto di tipo TestRestTemplate (dello spring framework). Viene verificato che la risposta del server sia "200 OK".
  - Viene eseguita un'altra chiamata con gli stessi dati, il server dovrebbe rifiutare una tale richiesta perché è già stato registrato un utente con la e-mail specificata. Viene verificato quindi che la risposta sia "400 BAD\_REQUEST".
  - Viene eseguita una richiesta http (url: `http://localhost:8080/login`) il cui corpo sono l'e-mail e la password dell'utente precedentemente registrato, si verifica che l'oggetto Utente ritornato dal server abbia gli stessi dati di quello registrato.
  - Vengono eseguite due chiamate http al medesimo url ma con e-mail e password errati, si verifica che la risposta sia "400 BAD\_REQUEST".
  - Viene cancellato l'utente dal database (per non "inquinare" il database con i dati inseriti in fase di test)
- **testAggiungiLibro:**
  - Viene effettuata la registrazione di un utente ed il relativo login come descritto per il metodo precedente.
  - Viene creato un oggetto di tipo DtoLibro utilizzato come contenuto di una successiva richiesta http (url: `http://localhost:8080/utenti/{userId}` dove userId è l'id dell'utente ottenuto mediante login). La chiamata restituisce l'utente a cui si è aggiunto un libro, il quale avrà ora nella lista di libri il libro appena aggiunto. Si verifica che le informazioni in tale libro siano le stesse di quelle dell'oggetto di tipo DtoLibro.
  - Viene cancellato l'utente dal database.
- **testGrafoRaggiungibilità:**
  - Vengono registrati 3 nuovi utenti come descritto in precedenza
  - Si ottiene l'id dei tre utenti mediante il login (chiamando direttamente il servizio del server, senza passare per le richieste http).
  - Tramite questi id si ottengono i vertici del grafo relativi ai tre utenti.
  - Si verifica che nel grafo ci siano gli archi tra i tre vertici come atteso dalle posizioni geografiche degli utenti e dal campo maxDist. Per gli archi esistenti si verifica che il peso sia pari alla distanza geografica in chilometri tra gli utenti.
  - I tre utenti vengono cancellati dal database.

Il risultato dei test è il seguente:



Nota: come si può notare viene indicato che i test sono eseguiti mediante JUnit5, in realtà grazie alla dependency junit-vintage-engine viene comunque utilizzato JUnit4.

La copertura del codice di questi test (ottenuta con Eclemma) è la seguente:

Element	Coverage	Covered Instructio...	Missed Instructions	Total Instructions
▼ WebServer	 92,5 %	1.421	116	1.537
▼ src/main/java	 85,9 %	709	116	825
▼ com.ourbooks.code	 37,5 %	3	5	8
> WebServerApplication.java	 37,5 %	3	5	8
▼ com.ourbooks.code.domain.account	 87,3 %	384	56	440
> CondLibro.java	 100,0 %	44	0	44
> DtoLibro.java	 100,0 %	38	0	38
> DtoUtente.java	 100,0 %	45	0	45
> Libro.java	 58,0 %	40	29	69
> ServizioLibri.java	 100,0 %	26	0	26
> ServizioUtenti.java	 100,0 %	63	0	63
> Utente.java	 82,6 %	128	27	155
▼ com.ourbooks.code.domain.grafo	 80,4 %	226	55	281
> GrafoRaggiungibilita.java	 92,7 %	76	6	82
> ServizioGrafo.java	 75,6 %	102	33	135
> VerticeUtente.java	 75,0 %	48	16	64
▼ com.ourbooks.code.init	 100,0 %	30	0	30
> InitGrafo.java	 100,0 %	30	0	30
▼ com.ourbooks.code.web	 100,0 %	66	0	66
> WebController.java	 100,0 %	66	0	66
> src/test/java	 100,0 %	712	0	712

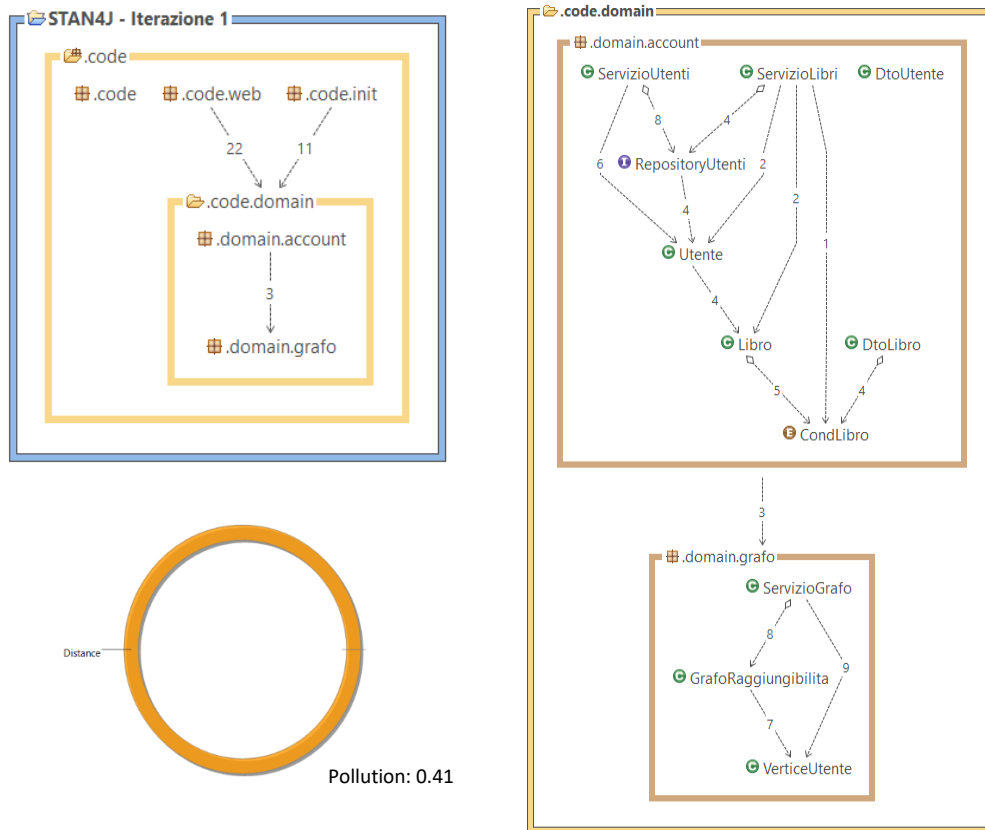
Il valore totale, ovvero 85,9% (test esclusi), è buono. Meno soddisfacente è la copertura delle classi:

- WebServerApplicazione (37,5%): il metodo main che avvia il server (ovvero l'unico contenuto nella classe) considerato da Eclemma come non eseguito, anche se evidentemente non è così.
- Libro e Utente (58% e 82,6%): il metodo equals, che costituisce buona parte del codice di ciascuna classe, viene implementato ma mai utilizzato.
- ServizioGrafo (75,6%): contiene due metodi (eliminaNodo e modificaNodo) implementati nell'iterazione 1 ma che non vengono chiamati per l'esecuzione dei casi d'uso d'alto livello scelti per quest'iterazione.
- VerticeUtente (75%): alcuni dei metodi set implementati non vengono mai chiamati, il metodo equals viene chiamato ma non vengono eseguiti tutti i branch dei costrutti if.

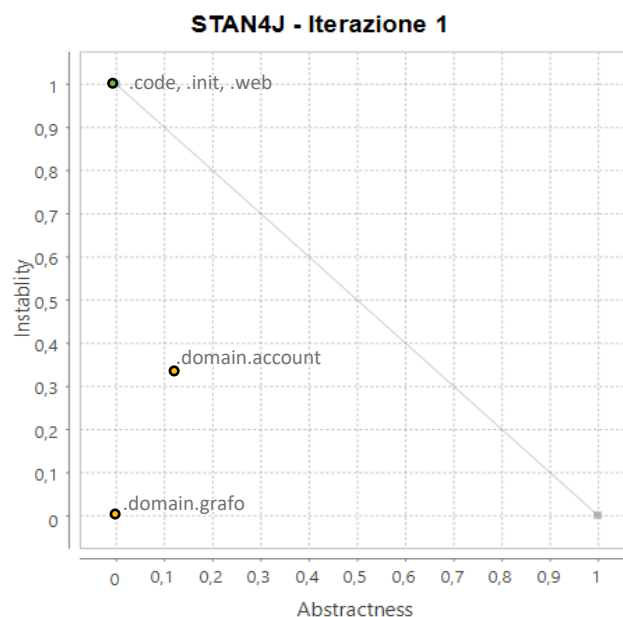


## Analisi statica: STAN4J

Di seguito vengono riportate le principali metriche fornite dal tool per eclipse STAN4J per l'analisi statica del codice. La seguente immagine mostra la composition view del progetto spring boot e il grafico della pollution. Come si può notare non sono presenti dipendenze circolari né tra i package né tra le classi al loro interno. E l'unica metrica violata è la distanza, che porta il valore di pollution a 0,41.

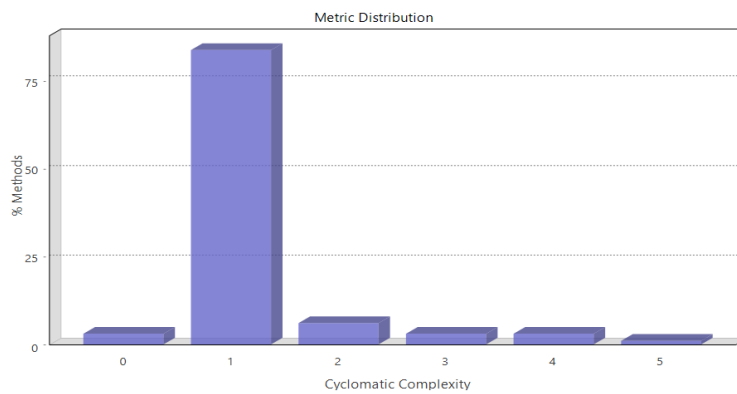
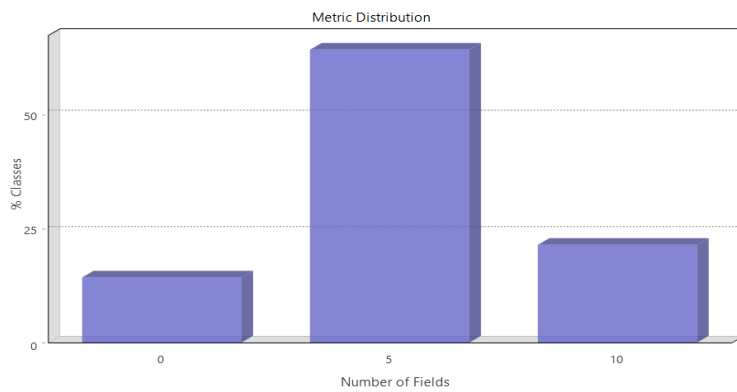
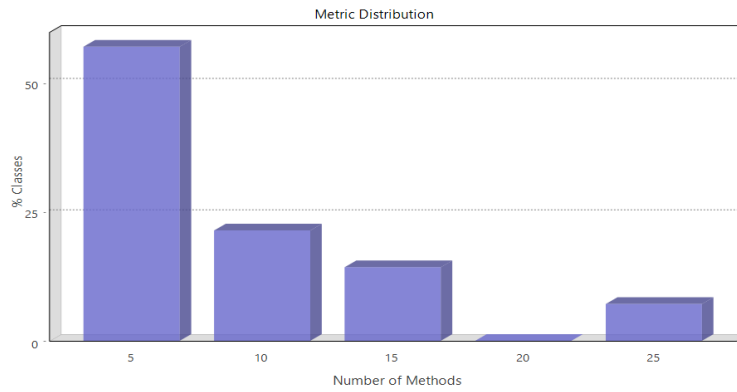


La distanza è negativa e dovuta principalmente alla mancanza di interfacce o classi astratte nei package del domain, come si può vedere nell'immagine a pagina successiva. La distanza del package `.domain.grafo` è -1 e del package `.domain.account` è invece -0,54.



Di seguito vengono riportate dei grafi riguardanti altre metriche che non vengono violate.

Metrica	Valore
Methods/Class	7.07
Fields/Class	3.21
Cyclomatic Complexity CC	1.22
Fat	2



## API esposte

Di seguito vengono mostrate, attraverso degli esempi di richieste corrette (ovvero che non generano errori) e relative risposte, le API esposte dal web server al termine della prima iterazione.

<b>API: signup</b>	
<b>Metodo HTTP:</b>	POST
<b>URL:</b>	http://localhost:8080/signup
<b>Parametri:</b>	nessuno
<b>Body:</b>	<pre>{   "email": "email@gmail.com",   "password": "passwordprova",   "lat": 35.5,   "lon": 4.6,   "maxDist": 10,   "libriDesiderati": ["Lo Hobbit", "V per Vendetta"] }</pre>
<b>Risposta:</b>	"Registrazione avvenuta con successo"

<b>API: login</b>	
<b>Metodo HTTP:</b>	POST
<b>URL:</b>	http://localhost:8080/login
<b>Parametri:</b>	nessuno
<b>Body:</b>	<pre>{   "email": "email@gmail.com",   "password": "passwordprova" }</pre>
<b>Risposta:</b>	<pre>{   "id": "fec9edfc-d615-4f9e-9594-11660f6830ed",   "email": "email@gmail.com",   "password": "passwordprova",   "lat": 35.5,   "lon": 4.6,   "maxDist": 10.0,   "libriDesiderati": [     "Lo Hobbit",     "V per Vendetta",     null   ],   "libri": [],   "nToken": 1000 }</pre>

**API: addLibro****Metodo HTTP:** PUT**URL:** http://localhost:8080/utenti/{userId}**Parametri:** nessuno**Body:**

```
{
  "titolo": "Cthulhu. I racconti del mito",
  "numPagine": 613,
  "yearPub": 2016,
  "condizioni": "OTTIME",
  "illustrato": true
}
```

**Risposta:**

```
{
  "id": "c07770c4-6d74-4760-b7ef-673f52549fca",
  "email": "email@gmail.com",
  "password": "passwordprova",
  "lat": 35.5,
  "lon": 4.6,
  "maxDist": 10.0,
  "libriDesiderati": [
    "Lo Hobbit",
    "V per Vendetta",
    null
  ],
  "libri": [
    {
      "id": "330988a0-4496-4b74-89f7-c2b11f017126",
      "titolo": "Cthulhu. I racconti del mito",
      "numPagine": 613,
      "yearPub": 2016,
      "condizioni": "OTTIME",
      "illustrato": true
    }
  ],
  "nToken": 1000
}
```

## Iterazione 2

### Casi d'uso

I casi d'uso implementati nell'iterazione 2 sono i seguenti:

- Visualizza libri disponibili (Id: 4)
- Calcola valore token (Id: 5)
- Compra libro (Id: 6)

In relazione al component diagram realizzato nell'early architecture design, quindi, si opererà sui subsystem **Gestore Acquisto libro** e **Gestore grafo**. Per i casi d'uso 4 e 5 si è implementato un algoritmo del quale design e analisi di complessità verranno descritti nelle sezioni successive. Di seguito vengono descritti in dettaglio partendo da una vista complessiva del sistema, comprendente quindi anche i client, anche se il progetto si concentrerà inizialmente sul solo sviluppo del codice lato server. Per i casi d'uso 4 e 5 si darà una descrizione unica visto che il caso d'uso 4 ingloba il 5, dal momento che il valore in token di un libro viene calcolato solamente quando si "scoprono" i libri disponibili per un utente.

<b>Nome: Visualizza libri disponibili e Calcola valore token</b>	
<b>Id:</b>	4 e 5
<b>Priorità:</b>	Alta
<b>Descrizione:</b>	L'utente seleziona l'opzione di visualizzazione dei libri acquistabili e visualizza una lista di libri a cui è associato un prezzo in token e l'opzione di acquisto.
<b>Pre-condizioni:</b>	L'utente è registrato e ha fatto login
<b>Post-condizioni:</b>	Viene visualizzata sul client la lista di libri disponibili all'utente con relativo prezzo in token
<b>Situazioni di errore:</b>	Nessuna
<b>Attori:</b>	Utente, Sistema
<b>Trigger:</b>	L'utente seleziona l'opzione di visualizzazione dei libri disponibili per l'acquisto
<b>Processo standard:</b>	(1) L'utente seleziona l'opzione di visualizzazione dei libri disponibili per l'acquisto (2) Il sistema calcola, attraverso un algoritmo*, i libri disponibili, il loro costo in token e a che utenti e in che quantità il costo verrebbe ridistribuito in caso di acquisto (3) L'utente visualizza la lista di libri disponibili e relativo prezzo in token

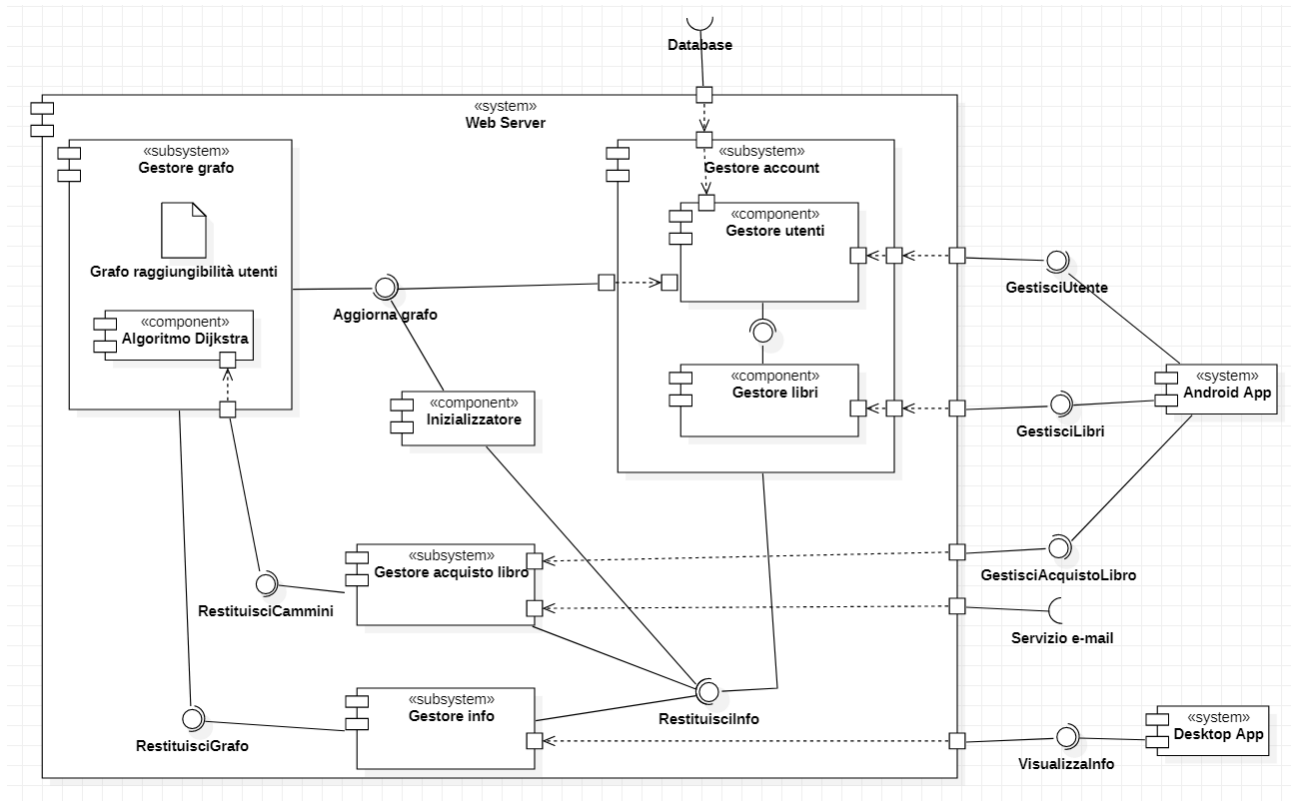
\*si vedano le successive sezioni

**Nome: Compra libro**

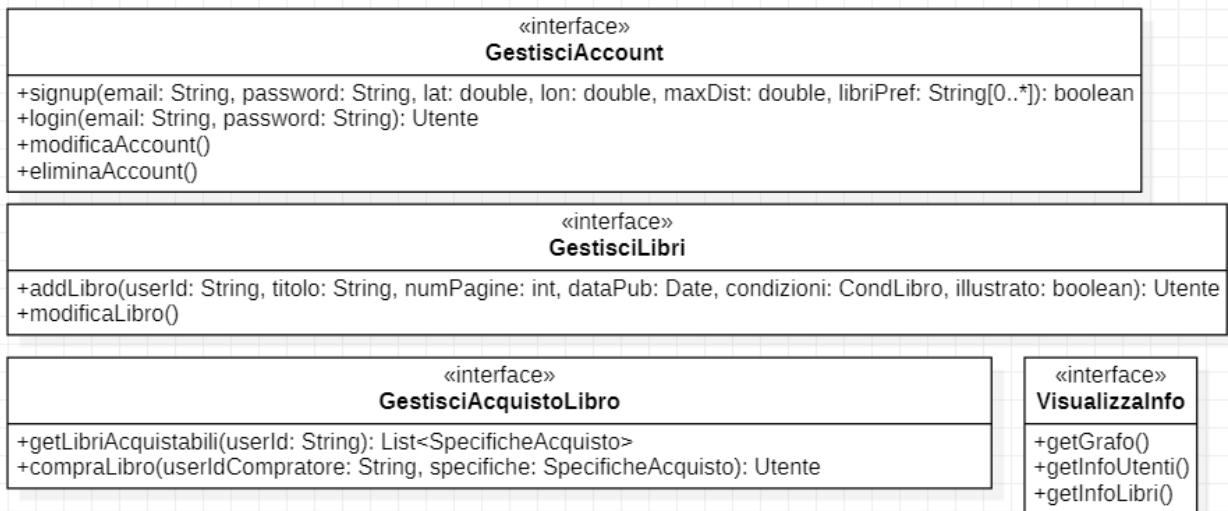
<b>Id:</b>	6
<b>Priorità:</b>	Alta
<b>Descrizione:</b>	L'utente seleziona il libro che vuole comprare, vengono avvisati il venditore e gli utenti che dovranno trasportare il libro (se presenti), vengono ripartiti i token dal compratore al venditore e agli altri utenti, viene eliminato il libro dalla lista del venditore
<b>Pre-condizioni:</b>	L'utente è registrato, ha fatto login e premuto sull'opzione di visualizzazione dei libri disponibili per l'acquisto
<b>Post-condizioni:</b>	Le informazioni degli utenti coinvolti nell'acquisto sono aggiornate correttamente: sono tolti i token necessari all'acquisto all'utente che ha acquistato il libro, viene tolto il libro dalla lista di quelli disponibili dell'utente che ha venduto il libro e aggiunti i token relativi al valore del libro, vengono aggiunti i token relativi agli spostamenti del venditore e degli eventuali ulteriori utenti
<b>Situazioni di errore:</b>	L'utente non possiede abbastanza token per acquistare il libro
<b>Stato del sistema in caso di errore:</b>	L'utente viene avvisato con un messaggio di errore
<b>Attori:</b>	Utente, Sistema
<b>Trigger:</b>	L'utente seleziona l'opzione di acquisto di un libro tra quelli visualizzati e disponibili (come descritto nel caso d'uso 4)
<b>Processo standard:</b>	<ol style="list-style-type: none"><li>(1) L'utente (compratore) seleziona il libro che vuole acquistare dalla lista di libri disponibili sul client</li><li>(2) L'utente conferma la volontà di acquistare il libro</li><li>(3) Il sistema produce le seguenti modifiche<ul style="list-style-type: none"><li>• Rimozione token all'utente compratore</li><li>• Rimozione del libro dalla lista di libri disponibili dell'utente venditore</li><li>• Aggiunta token relativi al valore del libro e dello spostamento del venditore</li><li>• Eventuale aggiunta token relativi allo spostamento a tutt'al più due utenti (che non sono né compratore né venditore)</li></ul></li><li>(4) L'utente viene avvisato del successo dell'operazione</li><li>(5) Vengono avvisati via e-mail il venditore e gli eventuali utenti che dovranno trasportare il libro</li></ol>

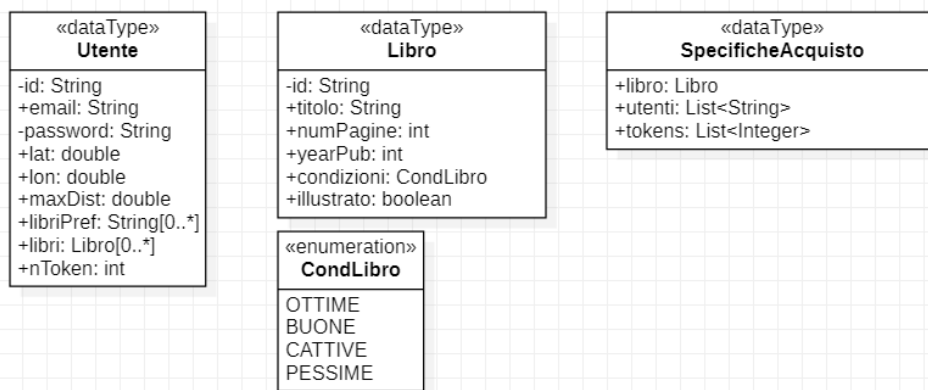
## Evoluzione dell'architettura

Il component diagram è stato raffinato per rappresentare meglio l'architettura del sistema dopo l'analisi dei casi d'uso implementati in questa iterazione. Il deployment diagram non è stato modificato in questa iterazione, rimane valido, quindi, quello descritto nell'iterazione 0.

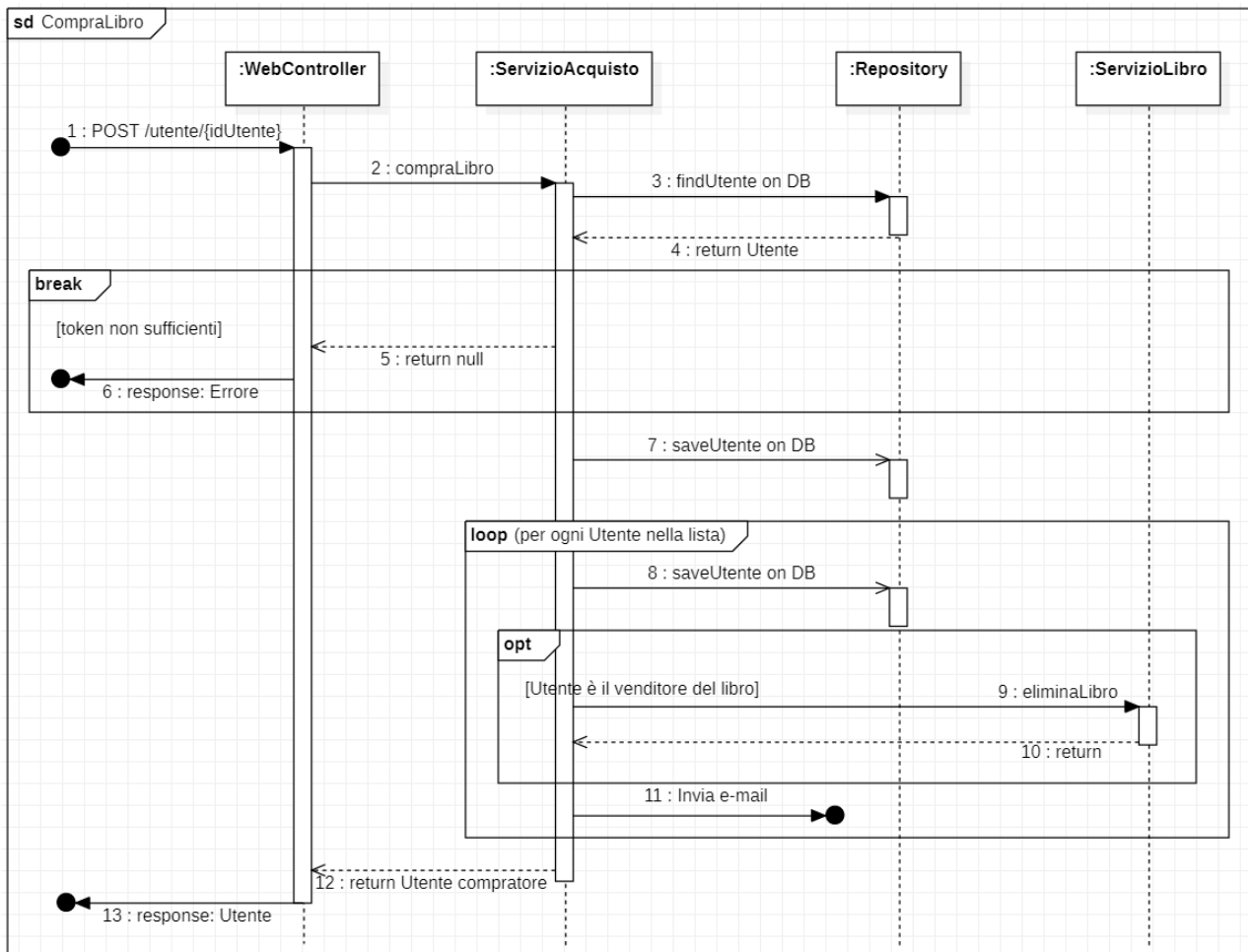


Sono stati anche ampliati le interfacce ed i data types. I relativi class diagram sono mostrati di seguito:





Le aggiunte riguardano l'interfaccia GestisciAcquistoLibro e il data type SpecificheAcquisto. Infine, è stata modellata solamente la vista dinamica relativa al caso d'uso **Compra Libro**, visto che per i casi d'uso **Visualizza libri disponibili** e **Calcola valore token** viene fatto un design in piccolo a parte.





## Design dell'algoritmo

L'algoritmo descritto di seguito implementa la logica dei casi d'uso **Visualizza libri disponibili** e **Calcola valore token**. L'algoritmo sarà distribuito sui subsystem **Gestore Acquisto libro** e **Gestore grafo**. Un utente può comprare solamente libri che possono essere consegnati a lui da al più tre persone, ossia il venditore stesso e al più due ulteriori utenti che trasportano semplicemente il libro. La valuta utilizzata per acquistare un libro è il token e, come già indicato, l'utente compratore dovrà versare al venditore un ammontare in token relativo al valore del libro (calcolato dall'algoritmo) e alla distanza che il venditore deve percorrere per consegnare il libro, inoltre, se la consegna del libro comprende anche ulteriori utenti, dovrà versare parte dei suoi token anche a tali utenti in funzione di che distanza essi devono percorrere. Se ci sono più possibilità di consegna di un libro, ovvero ci sono più possibili percorsi (in termini di utenti intermedi) per portare il libro dal venditore al compratore, viene scelta quello che minimizza il totale di chilometri percorsi e quindi il costo totale del libro. Di conseguenza verranno mostrati i libri appartenenti ad utenti per cui il cammino minimo dal compratore è di tre passi o meno. Di seguito viene indicato come queste quantità di token vengono calcolate:

- **Calcolo del valore di un libro:**

(Numero di pagine + illustrato) \* moltiplicatore condizioni \* moltiplicatore anno

- Illustrato = 0 token se non illustrato, 500 token se illustrato
- Moltiplicatore condizioni = 0.5 se PESSIME, 0.8 se CATTIVE, 1 se BUONE, 1.5 se OTTIME
- Moltiplicatore anni =  $1 + [50 - (\text{anno attuale} - \text{anno pubblicazione})] / 100$

Il moltiplicatore anni è compreso tra 1 (libro di 50 anni) e 1.5 (libro pubblicato nell'anno corrente). Inoltre, se il libro è più vecchio di 50 anni si imposta il moltiplicatore a 1.

- **Calcolo del valore di uno spostamento:**  $100 + (\text{Chilometri percorsi} * 20)$

Ad esempio, per un libro di 575 pagine, illustrato, in cattive condizioni e pubblicato nel 2013, si ha un valore in token di:

$$Valore_{libro} = (575 + 500) * 0.8 * \left(1 + \frac{50 - (2023 - 2013)}{100}\right) = 1075 * 0.8 * 1.4 = 1204 \text{ token}$$

Se il libro deve passare dal venditore ad un utente intermedio per una distanza di 20km e da questo utente al compratore per una distanza di 15km. Allora il valore degli spostamenti in token è:

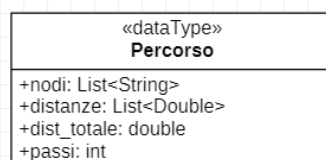
$$Valore_{spostamento1} = 100 + (20 * 20) = 500 \text{ token}$$

$$Valore_{spostamento2} = 100 + (15 * 20) = 400 \text{ token}$$

Se uno di questi valori non è un intero viene approssimato. Il valore totale che l'utente compratore dovrà quindi spendere è:

$$Costo_{totale} = 1204 + 500 + 400 = 2104 \text{ token}$$

L'input dell'algoritmo è l'id dell'utente (String) che ha richiesto il servizio di visualizzazione libri, l'output è una lista di specifiche di acquisto (classe SpecificheAcquisto), una per ogni libro disponibile all'acquisto. L'algoritmo lavora sul grafo raggiungibilità utenti, in cui ogni vertice presenta un campo di tipo Percorso definito dal seguente class diagram:



Di seguito vengono descritti in prosa i passi seguiti dall'algoritmo:

1. Viene chiamata una modifica all'algoritmo di Dijkstra sul grafo raggiungibilità utenti:
  - 1.1. Si ottiene il vertice con id pari al parametro in input, ovvero il vertice da cui inizia l'algoritmo.
  - 1.2. Per ogni vertice del grafo se ne inizializza il percorso, impostando la distanza a infinito, il numero di passi a zero, la lista di nodi precedenti ad una lista vuota e la lista delle distanze ad una lista vuota. Si imposta la distanza del vertice iniziale a zero.
  - 1.3. Si creano una lista di percorsi (che sarà l'output dell'algoritmo di Dijkstra) e una di vertici, chiamata Q. Si inizializza Q aggiungendo tutti i vertici del grafo.
  - 1.4. Finché Q non è vuota, si estrae da Q il vertice con minor valore di distanza del percorso. Se questo ha distanza finita, si aggiunge il suo percorso alla lista dei percorsi e se il numero dei passi del suo percorso è minore di tre si rilassano gli archi in uscita al vertice, altrimenti si estrae un nuovo vertice da Q. Se tutti i vertici in Q hanno distanza infinita si esce dal ciclo. Il rilassamento di un arco consiste nel far passare il percorso relativo al vertice a cui esso punta per il vertice da cui esso parte se tale nuovo percorso migliora la distanza totale.
  - 1.5. Si restituisce la lista dei percorsi così ottenuti.
2. Si crea una lista di specifiche acquisto inizialmente vuota, che è l'output dell'algoritmo.
3. Per ogni percorso ottenuto al punto precedente, si calcola la suddivisione dei token per gli utenti che compaiono nel percorso in funzione della distanza che devono percorrere. Inoltre, per ogni libro nella lista dei libri dell'ultimo utente del percorso (ovvero il venditore) se ne calcola il valore in token, si crea una specifica acquisto e questa viene aggiunta nella lista di specifiche acquisto.
4. Si restituisce la lista delle specifiche acquisto così ottenuta.

Lo pseudocodice corrispondente è:

```

algoritmo getLibriAcquistabili (Grafo G = (V, E), String idAcquirente)
    → List di SpecificheAcquisto
O(n2) { List di Percorso percorsi := getPercorsiMinimi(G, idAcquirente)
    List di SpecificheAcquisto result := ∅
    Utente u := null
    List di String utenti_percorso := null
    List di int token_percorso := null
    foreach Percorso p in percorsi do
O(n log n) { u := Utente con id pari alla prima stringa in p.nodi
    utenti_percorso := p.nodi
    token_percorso := ∅
    foreach double d in p.distanze do
        calcola il valore in token relativo alla distanza e
        aggiungilo a token_percorso
    endfor
    foreach Libro l in u.libri do
O(n) { crea un nuovo SpecificheAcquisto con libro := l,
    utenti := utenti_percorso, tokens := token_percorso
    O(k) { calcola il valore in token del libro e sommalo al primo
    In totale elemento di tokens di SpecificheAcquisto
    aggiungi l'oggetto così creato a result
    endfor
    endfor
    return result;
  
```

```

algoritmo getPercorsiMinimi (Grafo G = (V, E), String idAcquirente)
    → List di Percorsi
O(n) { Vertice acquirente := Vertice in V con id uguale a idAcquirente
    foreach vertice v in V do
O(n) {
        v.percorso.nodi :=  $\emptyset$ 
        v.percorso.distanze :=  $\emptyset$ 
        v.percorso.passi := 0
        v.percorso.dist_totale := Infinito
    endfor
    acquirente.percorso.dist_totale := 0
    List di Percorso percorsi :=  $\emptyset$ 
    Set di Utente Q :=  $\emptyset$ 
O(n) { Aggiungi a Q tutti i vertici del grafo G
    Vertice vu
    while (Q !=  $\emptyset$ ) do
O(n) { vu := extraxtMin(Q)
        if (vu == null) then
            break;
        endif
        aggiungi vu.percorso in coda a percorsi
        if (vu.percorso.passi >= 3) then
            continue;
        endif
O(n) { foreach vertice vua adiacente a vu do
        if (vua.percorso.dist_totale > vu.percorso.dist_totale
            + peso arco (vu,vua) then
            vua.percorso.nodi :=
                vu.percorso.nodi U {vua.id}

            vua.percorso.distanze :=
                vu.percorso.distanze U {peso arco (vu,vua)}

            vua.percorso.passi :=
                vu.percorso.passi + 1

            vua.percorso.dist_totale :=
                vu.percorso.dist_totale + peso arco (vu,vua)
        endif
    endfor
    endwhile
    rimuovi il primo elemento di percorsi
    return percorsi
}

```

O(m)  
In totale

```

algoritmo extractMin (Set Q di Vertici) → Vertice
    double min_dist := Infinito
    Vertice min_vu := null
O(n) { foreach Vertice v in Q do
        if (v.percorso.dist_totale < min_dist) then
            min_dist := v.percorso.dist_totale
            min_vu := v
        endif
    endfor
    if (min_vu != null) then
        rimuovi min_vu da Q
    endif
    return min_vu
}

```

## Complessità temporale

Nello pseudocodice sono indicate le complessità temporali di segmenti di codice nel caso peggiore. La complessità temporale è in funzione della taglia dell'input, che in questo caso è:

- **n:** numero di utenti nel sistema, ovvero numero di vertici del grafo raggiungibilità utenti
- **m:** numero di archi del grafo raggiungibilità utenti
- **k:** numero totale dei libri resi disponibili dagli utenti

Se accanto ad una istruzione o ad un ciclo non è riportata alcuna informazione sulla complessità significa che tale parte di codice ha complessità temporale  $O(1)$ . Di seguito viene descritta passo per passo la complessità degli algoritmi così da poter capire la complessità temporale dell'algoritmo `getLibriAcquistabili`, nel caso migliore e peggiore.

### CASO PEGGIORE

Il caso peggiore si verifica quando, nel grafo raggiungibilità utenti, il vertice da qui inizia l'algoritmo è in grado di raggiungere tutti gli altri utenti attraverso un percorso di costo minimo di al massimo 3 passi. Ossia l'utente che ha fatto la richiesta di visualizzazione dei libri acquistabili può comprare da tutti gli utenti nel sistema.

- algoritmo `extractMin`: essendo la ricerca di un minimo, l'algoritmo scansiona l'intero set di vertici, compiendo operazioni con complessità temporale costante ad ogni iterazione, quindi si ha  $O(n)$ .
- algoritmo `getPercorsiMinimi`: vengono eseguite inizialmente una serie di operazioni con complessità  $O(n)$ , ossia la ricerca di un vertice con un determinato id, l'inizializzazione di tutti i vertici con operazioni di complessità temporale costante, l'inizializzazione della lista `Q` di  $n$  elementi. Successivamente viene eseguito un ciclo `while` fintantoché non sono stati estratti da `Q` tutti gli elementi (caso peggiore), quindi viene eseguito  $O(n)$  volte. All'interno di tale ciclo viene chiamato `extractMin` (complessità  $O(n)$ ) e successivamente un ciclo `for` che viene eseguito in totale (ovvero considerando tutte le iterazioni del ciclo `while`)  $O(m)$  volte. Ogni iterazione di tale ciclo `for` ha complessità costante  $O(1)$ . In conclusione, la complessità è quindi  $O(n) + O(n^2) + O(m)$ , visto che  $m$  è, nel caso peggiore, pari a  $n^2 - n$ , si ha  $O(n^2)$ .
- algoritmo `getLibriAcquistabili`: inizialmente viene chiamato `getPercorsiMinimi`, con complessità  $O(n^2)$ , poi viene eseguito un ciclo `for` che itera sul numero di percorsi, nel caso peggiore ogni vertice è raggiungibile attraverso un cammino minimo di 3 passi dal vertice iniziale; quindi, il numero di percorsi è  $O(n)$ . Ad ogni iterazione viene ricercato nel database un utente in funzione del suo id, dal momento che nel database MongoDB l'id è indicizzato, la ricerca viene fatta su un B-albero e quindi ha complessità  $O(\log n)$ . Il ciclo `for` successivo viene eseguito in totale (ovvero considerando tutte le iterazioni del ciclo `for` più esterno)  $O(k)$  volte, ossia una volta per libro. La complessità è quindi  $O(n^2) + O(n \cdot \log n) + O(k)$ . Ragionevolmente  $k$  (ossia il numero totale di libri nel sistema) è  $O(n^2)$ , di conseguenza la complessità temporale è  $T(n) = O(n^2)$ .

### CASO MIGLIORE

Il caso migliore si verifica quando, nel grafo raggiungibilità utenti, il vertice da cui inizia l'algoritmo non ha vertici adiacenti. Ossia nessun utente è in grado di raggiungere l'utente che ha fatto la richiesta di visualizzazione dei libri disponibili.

- algoritmo `extractMin`: non dipende dalla "forma" dell'input e quindi vale il ragionamento fatto per il caso peggiore, la complessità è  $O(n)$ .
- algoritmo `getPercorsiMinimi`: La prima parte del codice (ovvero prima del ciclo `while`) non dipende dalla forma dell'input e quindi, come detto prima, ha complessità  $O(n)$ . Nel caso migliore il ciclo `while` viene eseguito due volte: la prima per estrarre il vertice da cui inizia l'algoritmo, quindi

con complessità  $O(n)$ , nel secondo invece `extractMin` restituisce null (visto che nessun vertice è raggiungibile dal primo) e il ciclo `while` termina. Quindi la complessità temporale è  $O(n) + O(n) = O(n)$ .

- algoritmo `getLibriAcquistabili`: nel caso migliore, quindi, `getPercorsiMinimi` (con complessità temporale  $O(n)$ ) restituisce una lista vuota e il ciclo `for` non viene eseguito nemmeno una volta. **La complessità temporale è quindi  $T(n) = O(n)$ .**

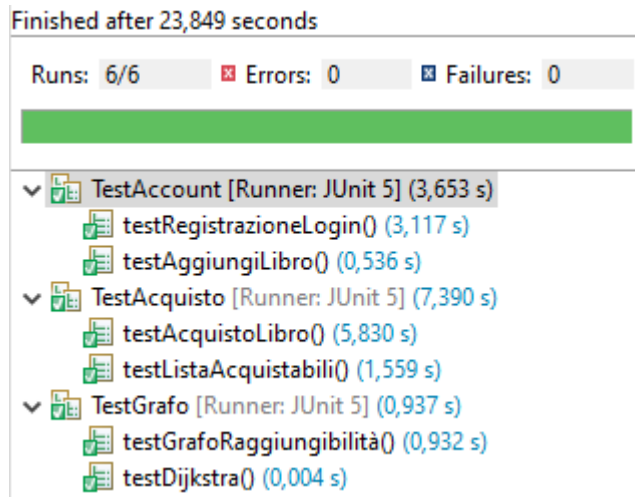
In generale, quindi, la complessità nel caso peggiore è  $O(n^2)$  e in quello migliore è  $O(n)$ , dove  $n$  è il numero di utenti presenti nel sistema

## Analisi dinamica: JUnit

Per testare la corretta implementazione dei casi d'uso scelti per l'iterazione 2 è stata realizzata la classe di test `TestAcquisto` (con i metodi `testListaAcquistabili` e `testAcquistoLibro`). Inoltre, è stato implementato un nuovo test (metodo `testDijkstra`) nella classe `TestGrafo`. Segue la descrizione dei metodi di test, realizzati con `SpringBootTest` e `JUnit 4`.

- **testDijkstra:**
  - Vengono cancellati tutti i nodi del grafo raggiungibilità, così che non interferiscano con il test.
  - Vengono aggiunti al grafo 6 nodi interconnessi tra loro (la creazione avviene direttamente attraverso un oggetto `ServizioGrafo` senza passare per le richieste Rest, già testate in precedenza). Per questo test non è necessario creare i nodi a partire dalla registrazione degli utenti.
  - Si chiama il metodo `getPercorsiMinimi` di `ServizioGrafo` (il metodo da testare) passando l'id del nodo 1 e si salva il risultato nella variabile "percorsi" di tipo `LinkedList<Percorso>`.
  - Si verifica che "percorsi" contenga i 4 percorsi minimi attesi e che non ne contenga altri.
- **testListaAcquistabili:**
  - Vengono cancellati tutti i nodi del grafo raggiungibilità, così che non interferiscano con il test.
  - Vengono creati (direttamente senza passare per le chiamate Rest) 3 utenti a cui vengono aggiunti alcuni libri.
  - Si esegue una chiamata Rest per ottenere la lista di libri acquistabili dall'utente 1 con i relativi utenti coinvolti e prezzi in token (`SpecificheAcquisto`).
  - Si verifica che questa lista contenga gli oggetti di tipo `SpecificheAcquisto` attesi e nient'altro.
  - Si eliminano dal database gli utenti registrati in questo test.
- **testAcquistoLibro:**
  - Vengono cancellati tutti i nodi del grafo raggiungibilità, così che non interferiscano con il test.
  - Vengono creati (direttamente senza passare per le chiamate Rest) 3 utenti a cui vengono aggiunti alcuni libri.
  - Si esegue una chiamata Rest per ottenere la lista di libri acquistabili dall'utente 1 (`SpecificheAcquisto`).
  - Si sceglie dalla lista un oggetto `SpecificheAcquisto` il cui prezzo in token sia minore di 1000 (quantità iniziale di token assegnati ai nuovi utenti).
  - Si esegue una chiamata Rest per l'acquisto da parte del primo utente del libro selezionato.
  - Si verifica che a seguito della chiamata i token siano stati redistribuiti correttamente tra gli utenti coinvolti e che il libro venduto non sia più presente tra la lista di libri dell'utente venditore.
  - Si sceglie dalla lista un altro oggetto `SpecificheAcquisto` il cui prezzo in token sia maggiore dei token rimanenti all'utente 1.
  - Si esegue una chiamata Rest per l'acquisto da parte del primo utente del libro selezionato.
  - Si verifica che la risposta sia un messaggio http con status "400 BAD\_REQUEST".
  - Si eliminano dal database gli utenti registrati in questo test.

Il risultato dei test è il seguente:



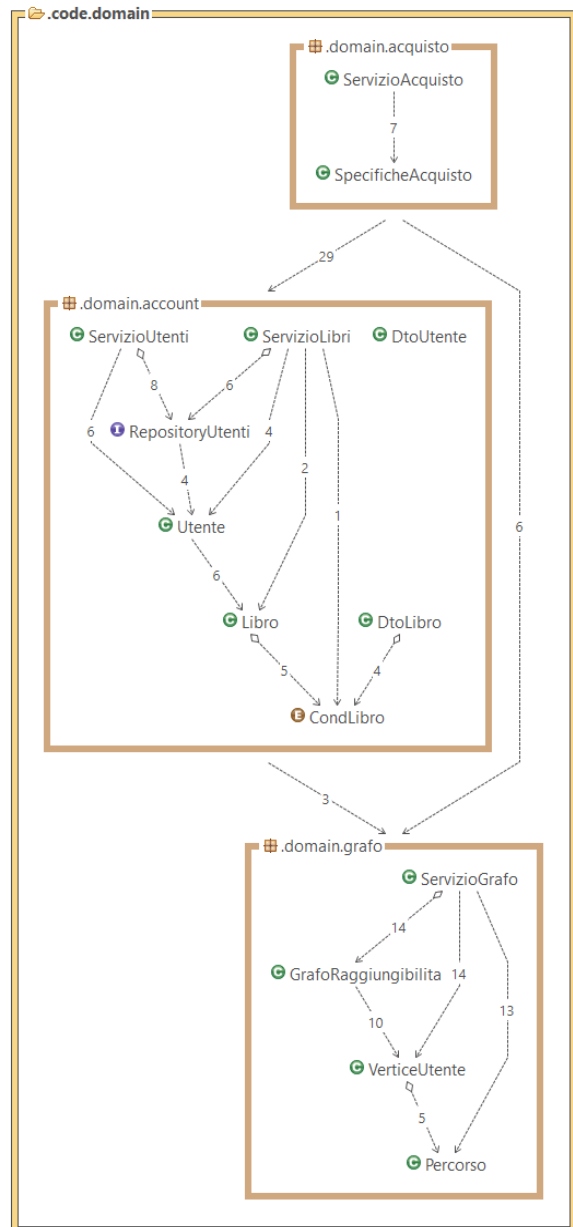
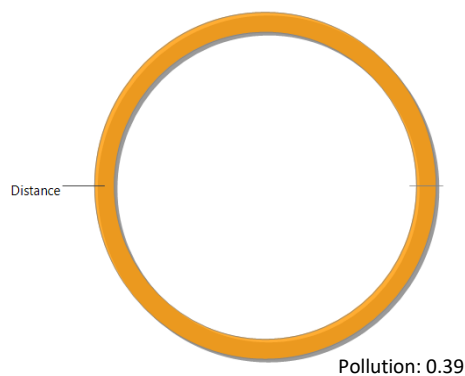
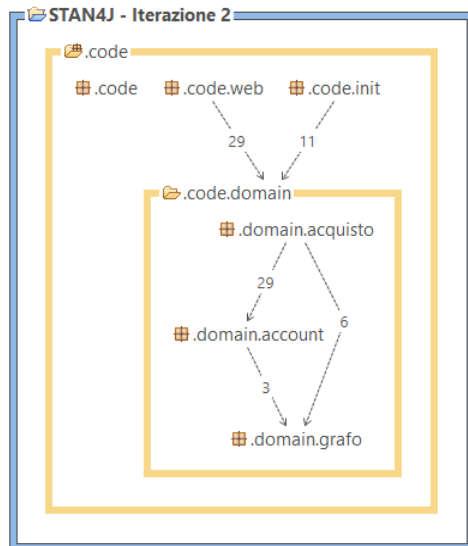
La copertura del codice di questi test (ottenuta con Eclemma) è la seguente:

Element	Coverage	Covered Instructio...	Missed Instructions	Total Instructions
WebServer	95,7 %	3.017	137	3.154
src/main/java	91,9 %	1.551	137	1.688
com.ourbooks.code	37,5 %	3	5	8
WebServerApplication.java	37,5 %	3	5	8
com.ourbooks.code.domain.account	86,5 %	391	61	452
CondLibro.java	100,0 %	27	0	27
DtoLibro.java	100,0 %	38	0	38
DtoUtente.java	100,0 %	45	0	45
Libro.java	85,5 %	59	10	69
ServizioLibri.java	100,0 %	40	0	40
ServizioUtenti.java	100,0 %	63	0	63
Utente.java	70,0 %	119	51	170
com.ourbooks.code.domain.acquisto	98,7 %	444	6	450
ServizioAcquisto.java	100,0 %	359	0	359
SpecificheAcquisto.java	93,4 %	85	6	91
com.ourbooks.code.domain.grafo	90,2 %	596	65	661
GrafoRaggiungibilita.java	94,7 %	108	6	114
Percorso.java	95,7 %	154	7	161
ServizioGrafo.java	89,4 %	278	33	311
VerticeUtente.java	74,7 %	56	19	75
com.ourbooks.code.init	100,0 %	30	0	30
InitGrafo.java	100,0 %	30	0	30
com.ourbooks.code.web	100,0 %	87	0	87
WebController.java	100,0 %	87	0	87
src/test/java	100,0 %	1.466	0	1.466

Il valore totale è 91,9% (esclusi i test). Le classi con una bassa copertura sono le stesse di quelle analizzate per l'iterazione 1, ovviamente le considerazioni sono le medesime.

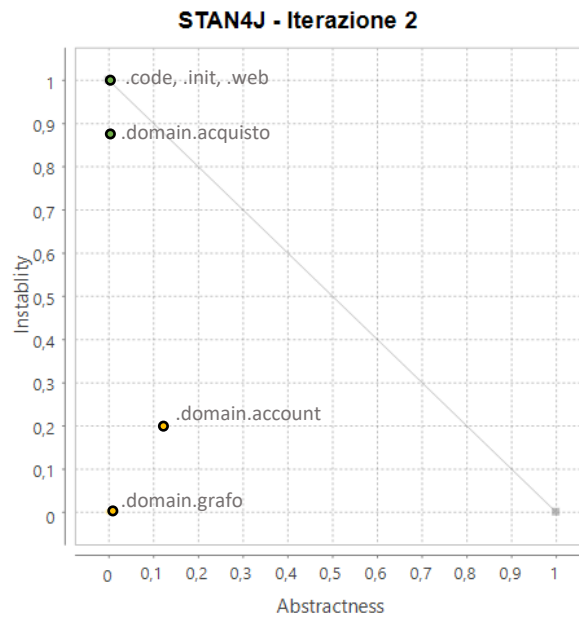
## Analisi statica: STAN4J

Di seguito vengono riportate le principali metriche fornite dal tool per eclipse STAN4J per l'analisi statica del codice al termine della seconda iterazione. La seguente immagine mostra la composition view del progetto spring boot e il grafo della pollution. Come si può notare non si sono create dipendenze circolari né tra i package né tra le classi al loro interno. L'unica metrica violata rimane la distanza, e la pollution diminuisce leggermente, diventando 0,39.



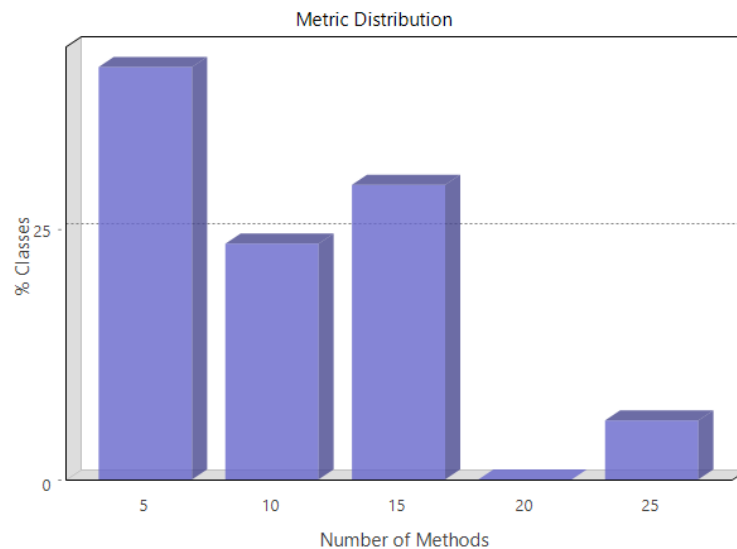


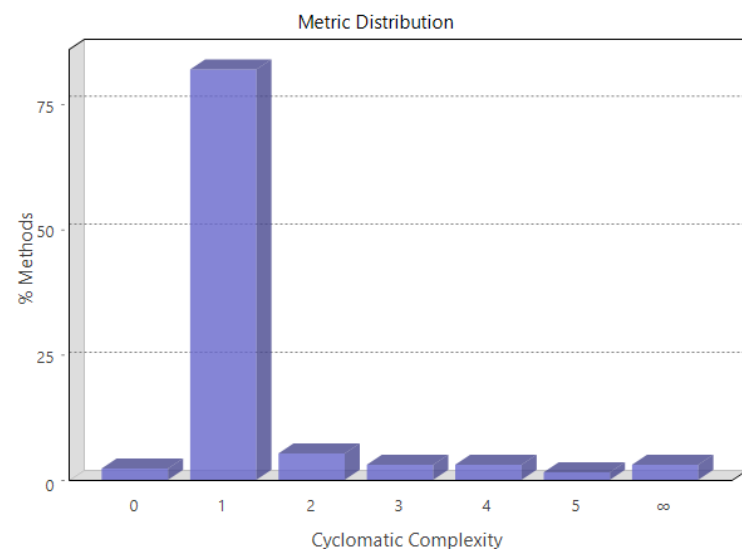
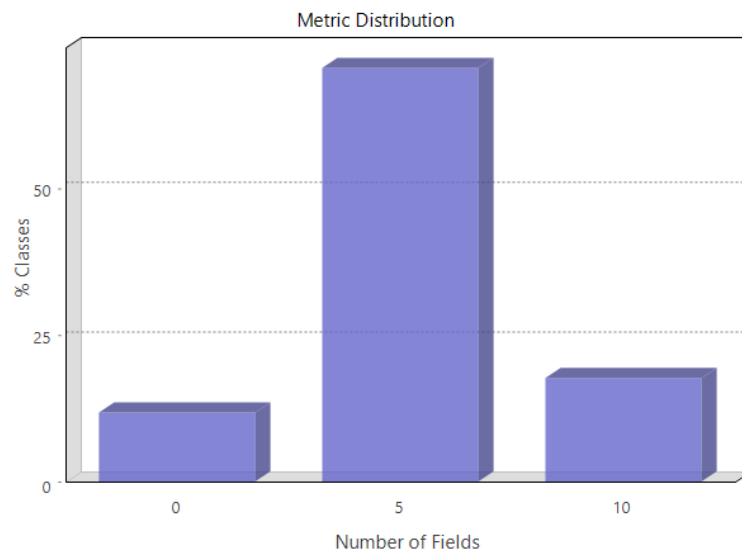
La violazione della distanza è da attestare al package `.domain.grafo`, per cui rimane -1, e al package `.domain.account`, per cui scende ulteriormente a -0,675.



Di seguito vengono riportate dei grafi riguardanti altre metriche che non vengono violate.

Metrica	Valore
Methods/Class	7.82
Fields/Class	3.41
Ciclomatic Complexity CC	1.47
Fat	2





## API esposte

Di seguito vengono mostrate, attraverso degli esempi di richieste e relative risposte, le nuove API esposte dal web server implementate nella seconda iterazione.

<b>API:</b>	<b>getLibriAcquistabili</b>
<b>Metodo HTTP:</b>	GET
<b>URL:</b>	http://localhost:8080/utenti/{idUtente}
<b>Parametri:</b>	nessuno
<b>Body:</b>	nessuno
<b>Risposta:</b>	<pre>[   {     "libro": {       "id": "ca07976c-6d05-4298-bd5d-d51ee7d89057",       "titolo": "It",       "numPagine": 1216,       "yearPub": 2019,       "condizioni": "CATTIVE",       "illustrato": false     },     "utenti": [       "b2fbcd3a-e088-4696-a1af-ae09464876a8"     ],     "tokens": [       1803     ]   },   {     "libro": {       "id": "1f886b62-ce79-4d7f-9042-23a20740c60a",       "titolo": "Cthulhu. I racconti del mito",       "numPagine": 613,       "yearPub": 2016,       "condizioni": "OTTIME",       "illustrato": true     },     "utenti": [       "d508f627-f791-4a31-b497-72efe472d3d3",       "b2fbcd3a-e088-4696-a1af-ae09464876a8"     ],     "tokens": [       2582,       383     ]   },   {     "libro": {       "id": "94a8f59b-9199-47b9-9381-c19dea628451",       "titolo": "Lo Hobbit",       "numPagine": 342,       "yearPub": 2012,       "condizioni": "CATTIVE",       "illustrato": false     }   } ]</pre>

```

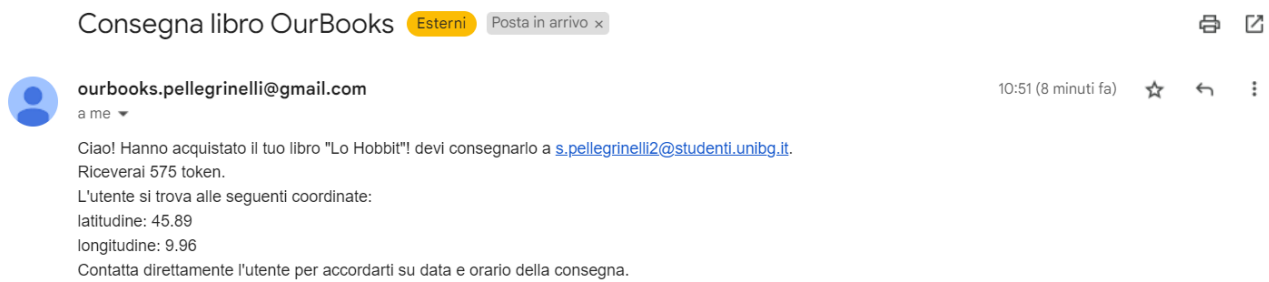
    },
    "utenti": [
        "d508f627-f791-4a31-b497-72efe472d3d3",
        "b2fbcd3a-e088-4696-a1af-ae09464876a8"
    ],
    "tokens": [
        575,
        383
    ]
}
]

```

### API: compraLibro

Metodo HTTP:	POST
URL:	http://localhost:8080/utenti/{idUtente}
Parametri:	nessuno
Body:	<pre> {   "libro": {     "id": "94a8f59b-9199-47b9-9381-c19dea628451",     "titolo": "Lo Hobbit",     "numPagine": 342,     "yearPub": 2012,     "condizioni": "CATTIVE",     "illustrato": false   },   "utenti": [     "d508f627-f791-4a31-b497-72efe472d3d3",     "b2fbcd3a-e088-4696-a1af-ae09464876a8"   ],   "tokens": [     575,     383   ] } </pre>
Risposta:	<pre> {   "id": "16497ef0-6817-4b22-929e-35d82ea63b19",   "email": "utente.compratore@gmail.com",   "password": "passwordutente",   "lat": 45.797,   "lon": 9.835,   "maxDist": 10.0,   "libriDesiderati": [     "Lo Hobbit",     "V per Vendetta",     null   ],   "libri": [],   "nToken": 42 } </pre>

In seguito ad una richiesta all'API **compraLibro**, inoltre, viene inviata una e-mail ad ogni utente che dovrà trasportare il libro. Per quanto riguarda l'esempio utilizzato qui sopra per mostrare tale API, sono state inviate due e-mail. La prima riguarda il venditore ed è la seguente:



La seconda invece riguarda un utente intermedio che dovrà portare il libro consegnatogli dal venditore all'utente compratore, la e-mail è la seguente:

