

Gestione di un sistema di book-crossing: *The WalkingBooks* *Documentazione progettuale*

Progetto del corso di Informatica IIIB
A.A. 2018/2019

PAGANESSI ANDREA
PIFFARI MICHELE
VILLA STEFANO

September 27, 2019

Contents

I Iterazione 0	1
1 Requisiti e specifiche	3
1.1 Requisiti utente	3
2 Use cases	5
2.1 Analisi testuale dei casi d'uso	5
2.2 Use Case Diagram	13
2.3 Funzionalità richieste	14
2.4 Stati del libro	14
2.5 Tool chain	15
3 Architettura	17
3.1 Deployment diagram	17
3.2 Architecture Envisioning	17
3.3 Database architecture	18
II Iterazione 1	21
4 Architettura	23
4.1 Architettura software	23
4.2 Logical view	24
5 Analisi dei componenti	27
5.1 Scelta delle funzionalità da implementare	27
5.2 Aderenza UML a Logical View	28
6 Observer-delegate pattern: la nostra implementazione	31
7 Implementazione comunicazione server	37
7.1 il framework <i>Netty</i>	37
7.2 Semantica dei messaggi	41
8 Creazione della base di dati	43
III Iterazione 2	45
9 Analisi dei componenti	47
10 Parte algoritmica: <i>reservation handler</i>	49

11 Test ed analisi dei componenti implementati	53
11.1 Lato Android	53
11.1.1 Espresso Junit	53
11.2 Lato Server	54
11.2.1 Analisi Statica	54

List of Figures

2.1	Use cases diagram	13
2.2	Stati del libro all'interno della community	15
3.1	Deployment Diagram	17
3.2	Architecture Envisioning	17
3.3	Modello del databse	18
4.1	Architettura Software	23
4.2	Logical view - GUI subsystem	24
4.3	Logical view - Request Manager subsystem client side	25
4.4	Logical view - Request Manager subsystem server side	25
4.5	Logical view - Server functionality	26
5.1	class diagram	28
5.2	Diagramma della classe ComputeRequest	29
6.1	Struttura generica pattern observer	32
6.2	Register function	32
6.3	Unregister function	33
6.4	Interfaccie per la ricezione degli eventi	33
6.5	Struttura del componente <i>DataDispatcher</i>	34
6.6	Struttura del componente <i>LoginFragment</i>	34
6.7	Struttura del componente <i>BookRegistrationFragment</i>	34
6.8	Struttura del componente <i>ResearchFragment</i>	35
6.9	Struttura del componente <i>ReservationFragment</i>	35
7.1	Struttura del package <i>requestManager.Communication</i>	39
7.2	Struttura del package <i>requestManager.Communication</i>	41
9.1	Diagramma della classe ComputeRequest	48
10.1	Distanza tra lettore e prenotante i-esimo	50
10.2	Zona d'incontro tra lettore e prenotante	50
10.3	Rete di utenti che potrebbero essere attivi nel prestito	52
11.1	Analisi statica del codice	54
11.2	analisi statica grafico	55

List of Tables

2.1 Panoramica requisiti funzionali progettuali	14
2.2 Descrizione requisiti non funzionali progettuali	14

Part I

Iterazione 0

1

Requisiti e specifiche

1.1 Requisiti utente

La startUp bergamasca Book Crossing UniBg desidera mettere a disposizione dei propri utenti un'applicazione Android per poter gestire la libera condivisione di libri all'interno di una vasta community di utenti.

I libri della rete di BookCrossing che l'azienda punta a gestire si possono trovare

- Casualmente (in stazione, su una panchina, in un locale o in un qualsiasi altro luogo): funzionalità "*on the go*"
- Nella zona di scambio ufficiale ("*OCZ UniBg*": Official Crossing Zone UniBg)

Per il momento, l'unica OCZ gestita direttamente dalla startUp si trova all'interno dell'aula studio del campus di Ingegneria di Dalmine, la quale coincide anche con la sistemazione del server centrale che andrà a gestire i vari interscambi tra gli users.

La startUp richiede che, per usufruire dell'applicativo mobile, i clienti debbano registrarsi fornendo i propri dati quali

- Nome
- Cognome
- Contatto di riferimento (opzionale)
 - Numero telefonico
 - Indirizzo mail
 - Facebook
 - ID Twitter
- Categorie di libri preferite
- Zona di residenza
- Raggio d'azione (inteso come l'area, in km, entro cui l'utente è disposto a spostarsi per incontrare altri utenti con cui scambiarsi libri)

Una volta registratosi, l'utente può partecipare al programma di Book Crossing.

Secondo la politica del book sharing, per rendere disponibile alla comunità uno o più libri che non sono ancora presenti nel network stesso, serve identificarli univocamente, per poterne così tracciare la storia, ovvero ciò che concerne il percorso seguito dal libro, le recensioni lasciate dagli utenti etc.

Prima di procedere con l'identificazione univoca del libro, l'utilizzatore deve inserire i dati del testo (o dei testi) che intende condividere con il resto della community: questo inserimento può avvenire

- In maniera "automatica" tramite scansione del codice ISBN
- In modalità "manuale", nel caso in cui, per esempio, non sia presente il barcode

fornendo i seguenti dati:

- Titolo
- Autore
- Anno di pubblicazione/Edizione
- Categoria

A questo punto il sistema genererà un BCID di 10 caratteri, ovvero un *Book Crossing ID* univoco, il quale dovrà essere riportato sul testo dall'utente.

La vera e propria condivisione avviene nel momento in cui il volume/i viene rilasciato (azione che può avvenire in un secondo momento rispetto alla fase di identificazione), il sistema dovrà acquisire i seguenti dati:

- Luogo di rilascio (con estensione future per un'acquisizione automatica della posizione tramite GPS)
- Ora e data di rilascio

L'app inoltre consiglierà all'utente un luogo di rilascio in cui sia già presente almeno un libro, facilitando così la creazione di cassette virtuali, ovvero di luoghi in cui sono presenti più libri: l'idea è quella quindi di permettere al sistema di creare, in maniera autonoma, dei punti "fissi" di consegna senza dover applicare interventi a livello infrastrutturale.

Successivamente il sistema dovrà notificare gli utenti, interessati al genere del libro rilasciato, della presenza di un nuovo testo, appena rilasciato, che potrebbe interessargli.

In qualsiasi momento è possibile effettuare le seguenti operazioni su ogni libro personalmente condiviso con la rete di sharing:

- Aggiunta di recensione
- Rating del libro

Quando viene trovato un libro (nel gergo definito come "*journal entry*"), il cliente che vuole prelevarlo, dopo aver effettuato il login nell'applicazione, deve inserire nell'apposito menù il BCID del libro che intende acquisire. Il sistema si occuperà poi di informare la community aggiornando lo status del libro raccolto, che diventerà "*underReading*".

Per quanto concerne invece l'area riservata, ogni utente ha la possibilità di visualizzare informazioni in merito ai libri che:

- ha messo a disposizione della community (**relased**)
- ha ottenuto dalla community (**chased**)
- attualmente possiede

L'utente può effettuare la prenotazione di libri già inseriti nella liste "chased" e "relased" del proprio profilo.

Il sistema deve prevedere anche la possibilità di ricercare un specifico testo e visualizzare i contatti dei lettori del libro al fine di potersi scambiare opinioni e/o pareri in merito al libro stesso. Tale funzionalità di ricerca permette anche la prenotazione del testo ricercato purché lo stesso sia nello stato "*under reading*". Per soddisfare questa richiesta il sistema provvederà a consigliare, al lettore corrente del libro prenotato, zone di rilascio specifiche al fine di avvicinare tale libro al richiedente, tenendo presente anche la necessità di creare cassette virtuali (come specificato in precedenza).

2

Use cases

2.1 Analisi testuale dei casi d'uso

- **UC1: Registration**

- **Descrizione:** registrazione alla rete di Book Crossing.
- **Attori coinvolti:** utente.
- **Preconditions:**
 - * smartphone dotato di connessione dati;
 - * l'utente non è ancora registrato al programma di Book Crossing.
- **Postconditions:** l'utente è registrato al programma di Book Crossing.
- **Processo:**
 1. l'utente seleziona “Registrati” nella schermata iniziale dell'applicazione;
 2. l'applicazione mostra un form in cui l'utente può inserire:
 - a. Nome
 - b. Cognome
 - c. Data di nascita
 - d. Username
 - e. Password
 - f. Contatto di riferimento
 - g. Categorie di libri preferite
 - h. Zona di residenza
 - i. Raggio d'azione rispetto alla zona di residenza
 3. l'utente inserisce i dati richiesti nel form presentato;
 4. l'utente attende la visualizzazione della conferma di avvenuta registrazione;
 5. l'utente viene reindirizzato alla pagina principale dell'applicazione.

- **Alternative**

- * **Dati non validi:** se l'utente inserisce dei dati non validi e/o mancanti, l'applicazione mostra un messaggio d'errore permettendo all'utente di modificare i dati non validi.

- **Estensioni**

- **UC2: Login**

- **Descrizione:** accesso alla rete di Book Crossing.
- **Attori coinvolti:** utente.
- **Preconditions:**
 - * smartphone dotato di connessione dati;

* l'utente è già registrato al programma di Book Crossing.

– **Postconditions:** l'utente è loggato nella rete di Book Crossing.

– **Processo:**

1. l'utente seleziona "Login" nella schermata iniziale dell'applicazione;
2. l'applicazione mostra una schermata in cui l'utente inserisce username e password;
3. l'utente inserisce i dati richiesti nella view presentata;
4. l'utente attende la verifica della correttezza dei dati inseriti;
5. l'utente viene reindirizzato alla pagina principale dell'applicazione.

– **Alternative**

* **Username e/o password non corretti:** se l'utente inserisce username e/o password non validi, l'applicazione mostra un messaggio d'errore permettendo all'utente di modificare i dati.

– **Estensioni**

- **UC3: Logout**

– **Descrizione:** disconnessione profilo personale.

– **Attori coinvolti:** utente.

– **Preconditions:**

- * smartphone dotato di connessione dati;
- * l'utente è già registrato al programma di Book Crossing;
- * l'utente è loggato.

– **Postconditions:** l'utente non è più loggato nella rete di Book Crossing.

– **Processo:**

1. l'utente seleziona "Profilo personale" nella schermata principale dell'applicazione;
2. l'applicazione mostra una schermata in cui l'utente può visualizzare tutte le proprie informazioni;
3. l'utente preme il bottone "Logout";
4. l'utente viene reindirizzato alla pagina di login.

- **UC4: Book pick-up**

– **Descrizione:** raccolta di un libro "on the go"¹ o in una OCZ.²

– **Attori coinvolti:** utente.

– **Preconditions:**

- * smartphone dotato di connessione dati;
- * l'utente ha effettuato l'accesso alla rete di Book Crossing;
- * il libro è stato siglato con il codice BCID.

– **Postconditions:** Il libro viene associato all'utente.

– **Processo:**

1. l'utente seleziona "Raccogli libro" nel menu principale dell'applicazione;
2. l'applicazione mostra un form in cui l'utente può inserire il BCID del libro;
3. l'utente inserisce il codice BCID riportato nel libro;
4. l'utente attende la visualizzazione di una scheda riepilogativa relativa al libro appena aggiunto;

¹In questo caso il libro condiviso dalla community viene raccolto dall'utente in una qualsiasi zona (come per esempio la stazione, il parco, sale di attesa etc...).

²*Official Crossing Zone*: zone riconosciute e fisse in cui la community può liberamente scambiarsi i libri.

5. l'utente verifica la corrispondenza delle informazioni mostrate;
6. l'utente conferma la raccolta.

– **Alternative**

- * **BCID inesistente:** se l'utente inserisce un BCID non esistente, l'applicazione mostra un messaggio d'errore permettendo all'utente di modificare il BCID.
- * **BCID associato ad un altro utente:** se l'utente inserisce un BCID già in possesso di un'altro utente, l'applicazione mostra un messaggio d'errore permettendo all'utente di modificare il BCID.
- * **BCID non corrispondente:** se l'utente, al punto (4), verifica che il libro reale non corrisponde alle informazioni mostrate dall'applicazione, può annullare l'operazione di raccolta.

– **Estensioni**

• **UC5: Book registration**

- **Descrizione:** registrazione di un libro alla rete di Book Crossing (*journal entry*).
- **Generalizzazione di:**
 - * aggiunta manuale dei dati del libro (UC8);
 - * scansione ISBN (UC9).
- **Include:** scrittura BCID (UC10).
- **Attori coinvolti:** utente.
- **Preconditions:**
 - * smartphone dotato di connessione dati;
 - * l'utente ha effettuato l'accesso alla rete di Book Crossing;
 - * il libro non è ancora stato siglato con il codice BCID.
- **Postconditions:** Il libro viene registrato alla rete di Book Crossing.
- **Processo:**
 1. l'utente seleziona "Registra un nuovo libro" nel menu principale dell'applicazione;
 2. l'applicazione tenta di iniziare la scansione ISBN mostrando all'utente la fotocamera;
 3. l'utente inquadra il codice ISBN del libro per il tempo sufficiente al riconoscimento del codice stesso;
 4. l'applicazione mostra all'utente la schermata contenente tutte le informazioni del libro;
 5. l'utente preme il pulsante "Conferma registrazione", dopo aver verificato rapidamente la coerenza dei dati.

– **Alternative**

- * **Aggiunta manuale dei dati:** se, dalla schermata di scansione, l'utente decide di inserire manualmente i dati del libro da registrare, l'applicazione mostra una schermata dove aggiungere manualmente i dati del libro.
- * **Scansione fallita:** se la scansione fallisce, l'applicazione riapre la fotocamera permettendo all'utente di ripetere l'operazione.

– **Estensioni**

• **UC6: Book research**

- **Descrizione:** ricerca di un libro all'interno della rete di Book Crossing.
- **Attori coinvolti:** utente.
- **Preconditions:**
 - * smartphone dotato di connessione dati;

- * l'utente ha effettuato l'accesso alla rete di Book Crossing;
- * il libro è presente nella rete di Book Crossing.
- **Postconditions:** il libro ricercato viene mostrato.
- **Processo:**
 1. l'utente seleziona "Ricerca libro" nel menu principale dell'applicazione;
 2. l'applicazione mostra all'utente un form da completare in cui inserire i parametri della ricerca;
 3. l'utente inserisce i parametri a cui è interessato;
 4. l'utente preme il pulsante "Cerca" ed attende il completamento;
 5. l'utente visualizza la lista di tutti i libri nella rete, che soddisfano la ricerca.
 6. il sistema verifica la presenza del libro cercato;
 7. in caso di esito positivo, l'applicazione mostra una scheda riassuntiva del libro;
 8. in caso di esito negativo, l'applicazione mostrerà un messaggio di errore.
- **Alternative**
 - * **Parametri non validi:** se l'utente inserisce dei parametri non validi, l'applicazione mostra un messaggio d'errore permettendo all'utente di modificarli.
 - * **Ricerca senza risultati:** se la ricerca non va a buon fine, l'applicazione mostra un messaggio all'utente, comunicando che nessun libro presente nella rete soddisfa i parametri di ricerca inseriti.
- **Estensioni**
 - * L'utente può selezionare uno dei libri mostrati dall'applicazione e visualizzare le sue informazioni.
- ***UC7: Info visualization***
 - **Descrizione:** Visualizzazione informazioni
 - **Generalizzazione:** visualizzazione informazioni libri chased (UC13), visualizzazione informazioni libri relase d(UC14) e visualizzazione informazioni libri in possesso (UC15).
 - **Attori coinvolti:** utente.
 - **Preconditions:**
 - * smartphone dotato di connessione dati;
 - * l'utente ha effettuato l'accesso alla rete di Book Crossing.
 - **Postconditions:** L'applicazione mostra le informazioni desiderate
 - **Processo:**
 1. l'utente seleziona la voce "I miei libri" nel menu principale dell'applicazione;
 2. l'applicazione mostra categorie di informazioni visualizzabili;
 3. l'utente seleziona la categoria che vuole visualizzare;
 4. l'applicazione mostra l'elenco dei libri della categoria selezionata.
 - **Alternative**
 - * **Nessun libro in elenco:** se nessun libro è presente nello storico, l'applicazione mostra un messaggio all'utente, comunicando che non è stata ancora effettuata nessuna operazione nella comunità.
 - **Estensioni**
- ***UC8: Personal area visualization***
 - **Descrizione:** Visualizzazione profilo personale utente
 - **Attori coinvolti:** utente.

– **Preconditions:**

- * smartphone dotato di connessione dati;
- * l'utente ha effettuato l'accesso alla rete di Book Crossing;

– **Postconditions:** l'applicazione mostra il profilo dell'utente.

– **Processo:**

1. l'utente seleziona la voce “Il mio profilo” nel menu principale dell'applicazione;
2. l'applicazione mostra l'anagrafica, i contatti e le attività svolte dall'utente;

– **Estensioni:**

• ***UC9: Manual addition of book's data***

– **Descrizione:** Inserimento manuale di un libro nella rete di Book Crossing

– **Attori coinvolti:** utente.

– **Preconditions:**

- * smartphone dotato di connessione dati;
- * l'utente ha effettuato l'accesso alla rete di Book Crossing;
- * il libro non è stato ancora siglato con il codice BCID.

– **Postcondition:** Viene generato il codice BCID e il libro viene aggiunto alla rete di Book Crossing

– **Processo:**

1. facendo riferimento al passo 1 e 2 del UC4, l'utente preme il pulsante “Aggiunta manuale”;
2. l'applicazione mostra un form da compilare con i dati del libro;
3. l'utente inserisce i dati del libro richiesti e conferma l'operazione;
4. l'applicazione mostra il codice BCID da trascrivere sul libro;
5. il sistema aggiunge il libro alla rete di Book Crossing.

– **Estensioni**

• ***UC10: ISBN scan***

– **Descrizione:** scansione del codice ISBN tramite fotocamera per ottenere le informazioni in merito al libro da registrare.

– **Attori coinvolti:** utente.

– **Preconditions:**

- * smartphone dotato di connessione dati;
- * l'utente ha effettuato l'accesso alla rete di Book Crossing;
- * il libro possiede il codice ISBN.

– **Postconditions:** il libro è in possesso dell'utente e non più condiviso con la community.

– **Processo:**

1. l'utente seleziona “Registra un nuovo libro” nel menu principale dell'applicazione;;
2. Viene aperta la fotocamera all'interno dell'applicazione;
3. l'utente inquadra il codice ISBN finché il sistema non rileva il barcode.

– **Alternative**

- * **ISBN non riconosciuto:** il sistema non è in grado di riconoscere l'ISBN inquadrato. Si chiuderà la fotocamera e l'utente verrà reindirizzato alla pagina di inserimento manuale del libro (UC9).

– **Estensioni**

- **UC11: Instruction to write BCID code³**

- **Descrizione:** scrittura del codice identificativo sul libro condiviso.
- **Attori coinvolti:** utente
- **Preconditions:**
 - * smartphone dotato di connessione dati;
 - * l'utente ha effettuato l'accesso alla rete di Book Crossing;
- **Postconditions:** il libro è univocamente riconosciuto del sistema tramite il BCID.
- **Processo:** l'utente copia il codice BCID sul libro, seguendo le istruzioni mostrate dall'applicazione.
- **Estensioni**

- **UC12: View of users contacts**

- **Descrizione:** l'utente ottiene i contatti che un altro utilizzatore ha deciso di condividere con la community.
- **Attori coinvolti:** utente.
- **Preconditions:**
 - * smartphone dotato di connessione dati;
 - * l'utente ha effettuato l'accesso alla rete di Book Crossing.
- **Postconditions:** l'applicazione visualizza i contatti a cui l'utente può e/o vuole essere contattato.
- **Processo:**
 1. l'utente seleziona "Ricerca" dal menu principale dell'applicazione;
 2. l'utente seleziona il libro a cui è interessato;
 3. il sistema mostra tutte le informazioni relative al libro (tra cui anche la lista di tutti i lettori che sono stati in possesso del libro in questione);
 4. l'utilizzatore seleziona l'utente che desidera contattare;
 5. il sistema mostra tutte le informazioni di contatto che il cliente terzo ha deciso di condividere con la community.
- **Alternative:**
 - * **Nessun libro trovato:** l'applicazione notifica l'utilizzatore del fatto che la ricerca non sia andata a buon fine.
 - * **Utente senza alcun contatto condiviso:** il sistema filtra la visualizzazione della lista degli utenti, visualizzando solo coloro che hanno inserito, durante la fase di registrazione, almeno un contatto o che desiderano essere contattati.
- **Estensioni**

- **UC13: Book reservation**

- **Descrizione:** l'utilizzatore prenota un determinato libro in possesso di un altro lettore.
- **Attori coinvolti:**
 - * utente richiedente (*claimant user*);
 - * utente attualmente in possesso del libro richiesto (*owner user*).
- **Preconditions:**
 - * smartphone dotato di connessione dati;
 - * l'utente ha effettuato l'accesso alla rete di Book Crossing;
 - * il libro che si vuole prenotare deve essere registrato alla rete;

³Book Crossing IDentifier

- * il libro richiesto deve essere già in possesso di un altro utente.

– **Postconditions:**

- * Se i due utenti hanno un punto di incontro in comune, si accordano sul luogo di scambio.
- * Se invece non hanno un punto di incontro comune, il libro passerà tra gli utenti che si trovano tra *claimant user* e *owner user*.

– **Processo:**

1. l'utente seleziona "Ricerca libro" nel menu principale dell'applicazione;
2. l'applicazione mostra all'utente un form da completare in cui inserire i parametri della ricerca;
3. l'utente inserisce i parametri a cui è interessato;
4. l'utente preme il pulsante "Cerca" ed attende il completamento;
5. l'utente visualizza la lista di tutti i libri nella rete, che soddisfano la ricerca.
6. il sistema verifica la presenza del libro cercato;
7. l'utente va a selezionare il libro all'interno della lista proposta dal sistema;
8. l'applicazione mostra un riepilogo sulle informazioni del libro, unitamente alla possibilità di prenotare;
9. l'utente preme il pulsante "Prenota";
10. l'applicazione mostra una pagina di conferma dell'avvenuta prenotazione.

– **Alternative**

- * **Libro non prenotabile:** il libro selezionato non è prenotabile poiché non in possesso di un altro utente.

– **Estensioni**

• ***UC14: Chased books informations***

– **Descrizione:** visualizzazione storico dei libri "raccolti" dall'utente.

– **Attori coinvolti:** utente.

– **Preconditions:**

- * smartphone dotato di connessione dati;
- * l'utente ha effettuato l'accesso alla rete di Book Crossing.

– **Postconditions:** mostrata sulla grafica la lista dei libri raccolti, con relativa data e luogo di "chasing".

– **Processo:**

1. l'utente seleziona "Il mio profilo" dal menù principale dell'applicazione;
2. l'applicazione mostra le informazioni dell'utente e i diversi stati in cui si possono trovare i suoi libri;
3. l'utente sceglie lo stato "Libri chased";
4. l'applicazione mostra un riepilogo di tutti i testi ottenuti dalla community di sharing.

– **Alternative:**

- * **Nessun libro chased:** l'applicazione mostra un messaggio all'utente, comunicando che nessun libro è stato ancora raccolto da lui dalla community.

– **Estensioni**

• ***UC15: Released books informations***

– **Descrizione:** visualizzazione storico dei propri libri condivisi con la rete.

– **Attori coinvolti:** utente.

– **Preconditions:**

- * smartphone dotato di connessione dati;
 - * l'utente ha effettuato l'accesso alla rete di Book Crossing.
 - **Postconditions:** mostrata sulla grafica la lista dei libri inseriti nel programma di sharing, con relativa data e luogo di "relase".
 - **Processo:**
 1. l'utente seleziona "Il mio profilo" dal menù principale dell'applicazione;
 2. l'applicazione mostra le informazioni dell'utente e i diversi stati in cui si possono trovare i suoi libri;
 3. l'utente sceglie lo stato "Libri released";
 4. l'applicazione mostra un riepilogo di tutti i libri rilasciati alla community di sharing.
 - **Alternative:**
 - * **Nessun libro released:** l'applicazione mostra un messaggio all'utente, comunicando che nessun libro è stato ancora rilasciato da lui nella rete di Book Crossing.
 - **Estensioni**
- **UC16: "Under reading" books informations**
 - **Descrizione:** visualizzazione dei propri libri attualmente "under reading".
 - **Attori coinvolti:** utente.
 - **Preconditions:**
 - * smartphone dotato di connessione dati;
 - * l'utente ha effettuato l'accesso alla rete di Book Crossing.
 - **Postconditions:** l'applicazione mostra la lista dei libri attualmente in possesso dell'utente
 - **Processo:**
 1. l'utente seleziona "Il mio profilo" dal menù principale dell'applicazione;
 2. l'applicazione mostra le informazioni dell'utente e i diversi stati in cui si possono trovare i suoi libri;
 3. l'utente sceglie lo stato "Libri in possesso";
 4. l'applicazione mostra un riepilogo di tutti i libri attualmente in possesso.
 - **Alternative:**
 - * **Nessun libro in lettura:** l'applicazione mostra un messaggio all'utente, comunicando che nessun libro è in suo possesso al momento.
 - **Estensioni**
 - **UC17: Book released**
 - **Descrizione:** l'utente libera un libro.
 - **Attori coinvolti:** utente.
 - **Preconditions:**
 - * smartphone dotato di connessione dati;
 - * l'utente ha effettuato l'accesso alla rete di Book Crossing;
 - * libro rilasciato già registrato al sistema di Book Crossing.
 - **Postconditions:** il libro passa dallo stato "under reading" a quello "Available".
 - **Processo:**
 1. l'utente seleziona "Il mio profilo" dal menù principale dell'applicazione;
 2. l'applicazione mostra le informazioni dell'utente e i diversi stati in cui si possono trovare i suoi libri;
 3. l'utente sceglie lo stato "Libri in possesso";

4. l'applicazione mostra un riepilogo di tutti i libri attualmente in possesso;
5. l'utente preme sul testo che intende rilasciare;
6. l'utente conferma il rilascio.
7. l'applicazione mostra una conferma di avvenuto rilascio del testo selezionato.

– **Alternative:**

- * **Nessun libro in lettura:** l'applicazione mostra un messaggio all'utente, comunicando che nessun libro è in suo possesso al momento.
- * **Segnale GPS non trovato:** l'applicazione avvisa l'utente di attivare il GPS del dispositivo.

– **Estensioni**

2.2 Use Case Diagram

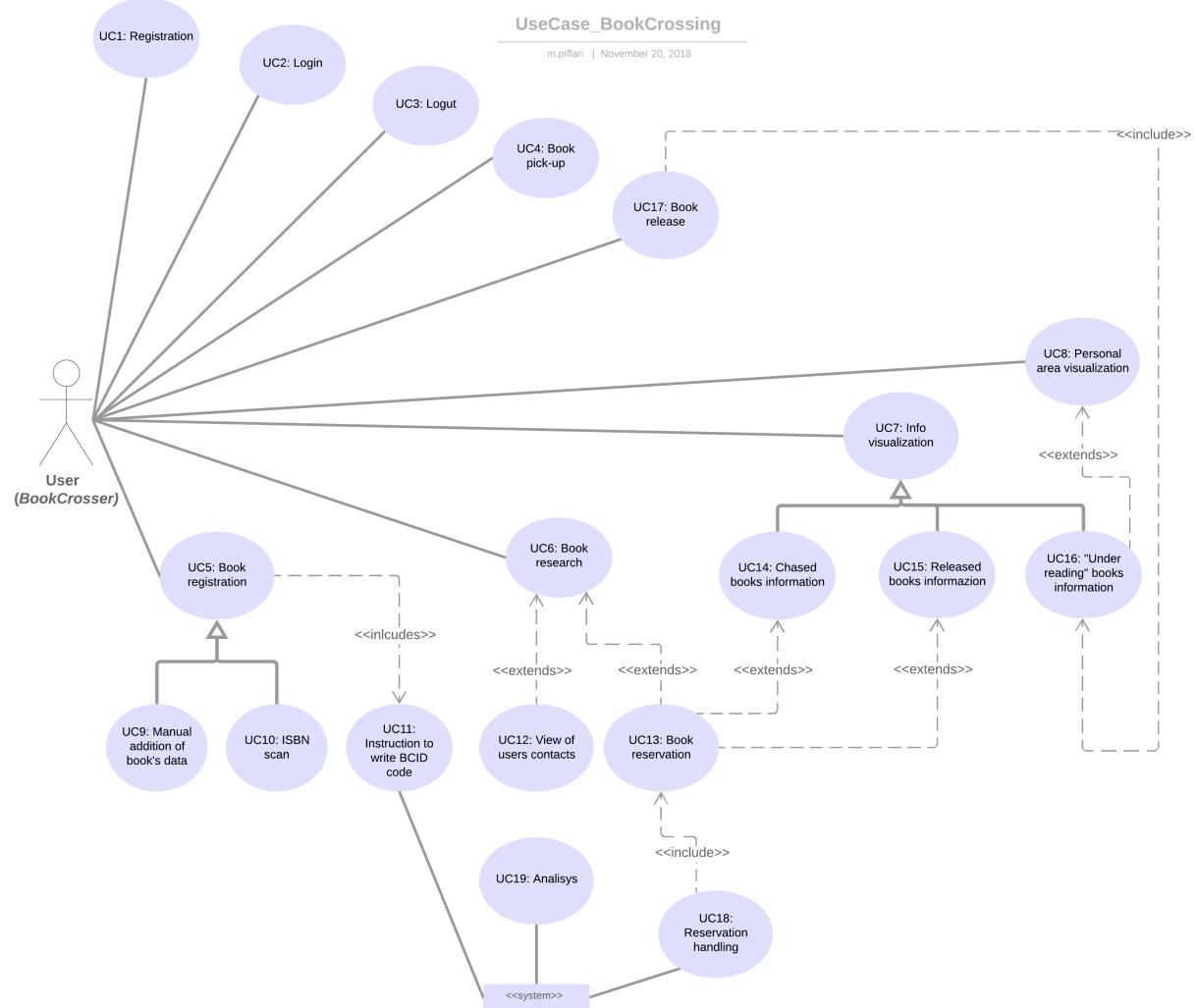


Figure 2.1: Use cases diagram

2.3 Funzionalità richieste

Table 2.1: Panoramica requisiti funzionali progettuali

Nome requisito	ID requisito	Tipologia	Priorità	Requisiti padre	Requisiti figli
Raccolta libro	UR1	funzionale	alta		UR2
Login utente	UR2	funzionale	alta	UR1	UR3, UR4
Registrazione utente	UR3	funzionale	alta	UR2	
Aggiunta libro	UR4	funzionale	alta	UR2	UR7, UR9
Ricerca libro	UR5	funzionale	media	UR2	UR10
Prenotazione libro	UR6	funzionale	bassa	UR2	
Visualizzazione info libri chased	UR7	funzionale	bassa	UR2	
Visualizzazione info libri released	UR8	funzionale	bassa	UR2, UR4	
Rilascio libro	UR9	funzionale	alta	UR2, UR4	
Visualizzazione contatti utenti	UR10	funzionale	bassa	UR2, UR5	
Visualizzazione profilo personale	UR11	funzionale	media	UR2	

Table 2.2: Descrizione requisiti non funzionali progettuali

Nome requisito	ID requisito	Descrizione
Controllo geolocalizzazione	UR12	Requisito non funzionalità che permette di verificare che la posizione GPS salvata del libro corrisponda, con margine d'accettazione, alla posizione in cui si trova l'utente nel momento in cui vuole raccogliere un libro trovato "on the go".

2.4 Stati del libro

La startup si impone l'obiettivo di andare a gestire lo scambio di libri all'interno della rete di book-crossing: come visto all'interno dei diversi use-cases, ogni libro, nel corso della propria vita all'interno della community, passa di mano in mano, attraversando diverse zone. A questo movimento fisico, corrisponde anche un continuo cambio di stato da parte del libro stesso: possiamo riassumere con una *"Finite State Machine"* il percorso che un generico libro segue durante la sua vita.

Riassumendo gli stati di un libro, possono essere:

- Out of the network
- Available
- Under reading
- Released
- Reserved

- Traveling

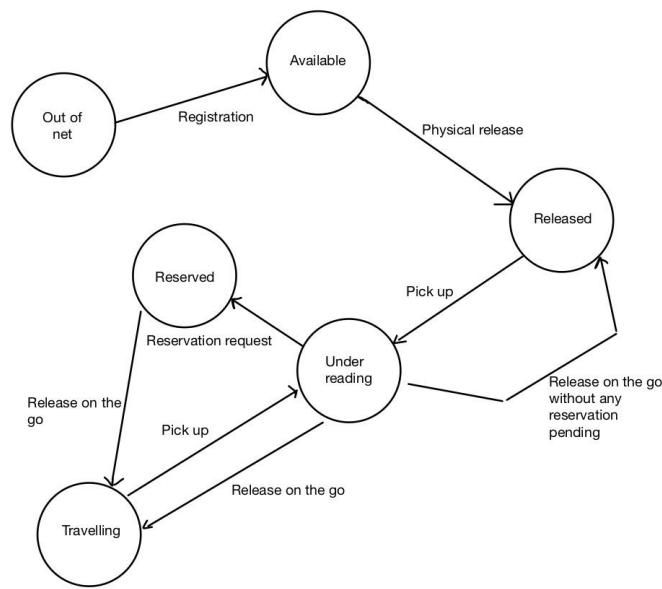


Figure 2.2: Stati del libro all'interno della community

2.5 Tool chain

TODO: tool chain

3

Architettura

3.1 Deployment diagram

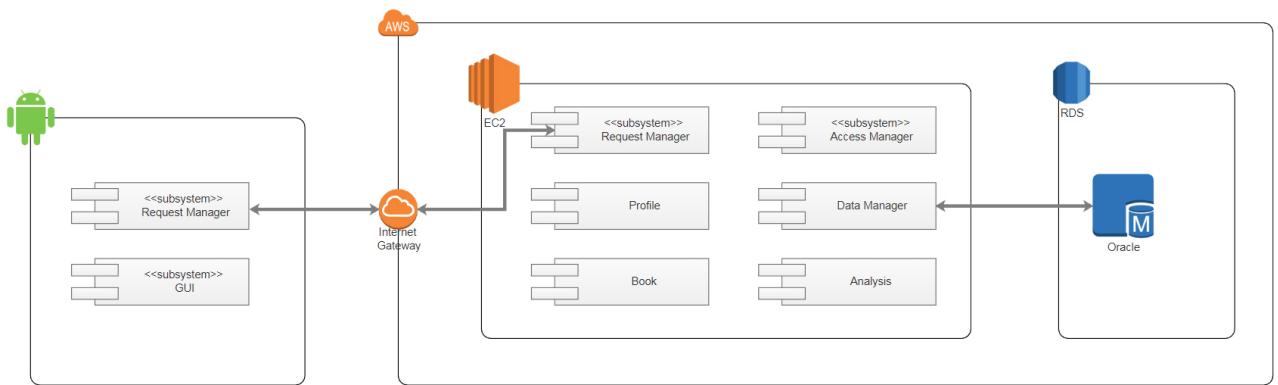


Figure 3.1: Deployment Diagram

3.2 Architecture Envisioning



Figure 3.2: Architecture Envisioning

In figura 3.1 e 3.2 sono mostrati il Deployment Diagram e l'Architecture Envisioning del sistema progettato per lo sviluppo dell'applicazione di Book Crossing. Si può osservare che si tratta di un'architettura **Three Tiers**:

1. A sinistra si individua il client, ovvero il dispositivo Android con il quale è possibile interfacciarsi. Al suo interno quindi si può osservare la presenza di un componente relativo all'interfaccia grafica e uno relativo alla gestione delle richieste per invio e ricezione di dati con il server;
2. Nella parte centrale individuiamo gli altri due layer dell'architettura: server EC2 e Database Relazionale RDS. Il fatto di utilizzare Amazon Web Services (AWS) consente di avere questi due elementi a bordo di un unico strato.

Per la comunicazione tra smartphone e EC2 utilizziamo un connettore basato su una socket TCP, mentre il server applicativo (EC2) si connette al DBMS (RDS Oracle) tramite il connettore OJDBC (Oracle Java DataBase Connectivity).

3.3 Database architecture

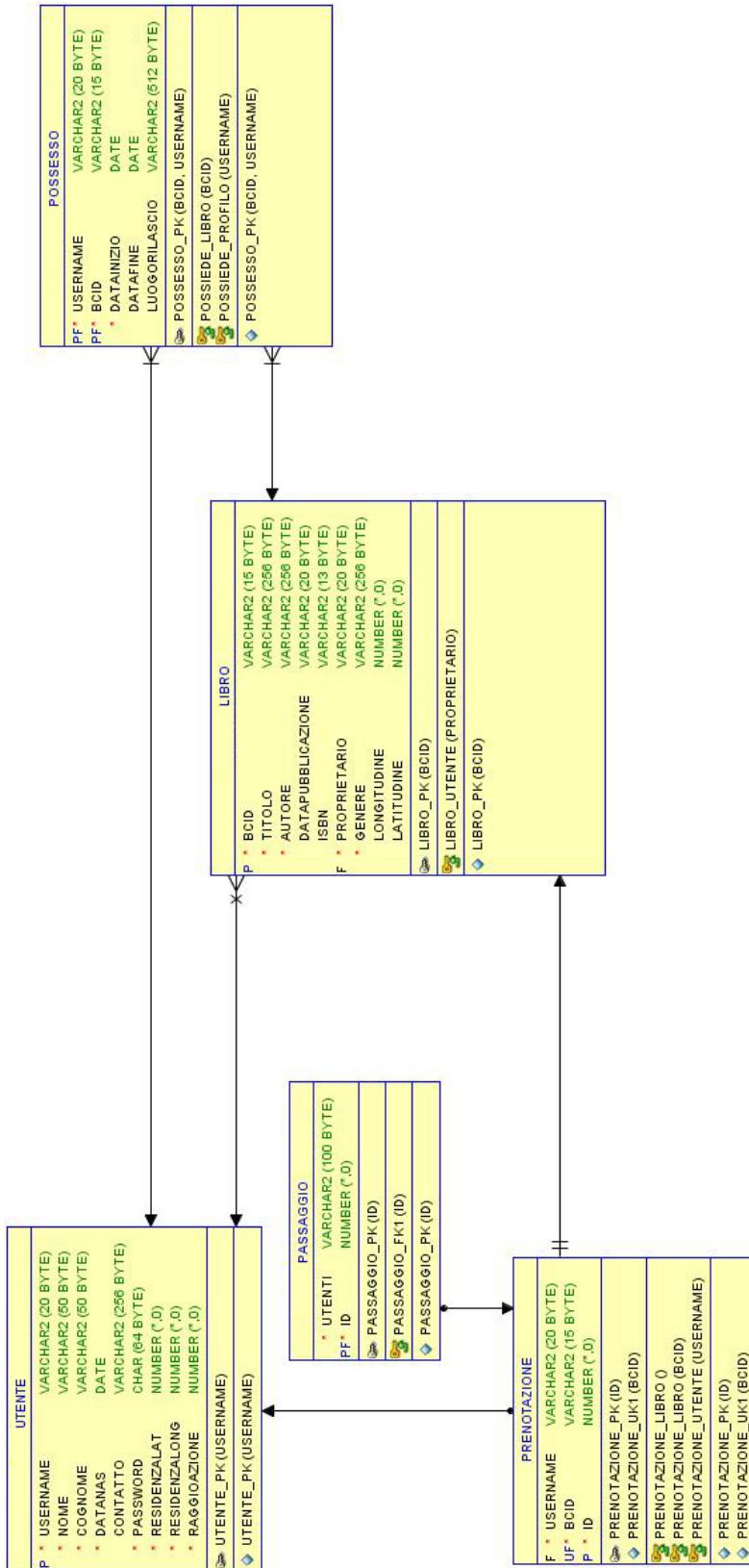


Figure 3.3: Modello del database

La figura 3.3 rappresenta il modello logico del database, le cui entità sono:

- **Utente** : contiene tutti i dati relativi all'utenza registrata, ogni utente è identificato da uno *username* univoco;
- **Libro** : contiene tutti i libri presenti nel sistema, ognuno dei quali è identificato da un BCID

generato univocamente;

- **Prenotazione** : rappresenta la relazione N:N tra Utente e Libro;
- **Passaggio** : contiene la sequenza di utenti che dovrebbero partecipare in maniera attiva alla prenotazione indicata dalla PK della tabella stessa;
- **Possesso** : questa tabella rappresenta i libri attualmente in possesso dagli utenti iscritti alla community.

Part II

Iterazione 1

4

Architettura

4.1 Architettura software

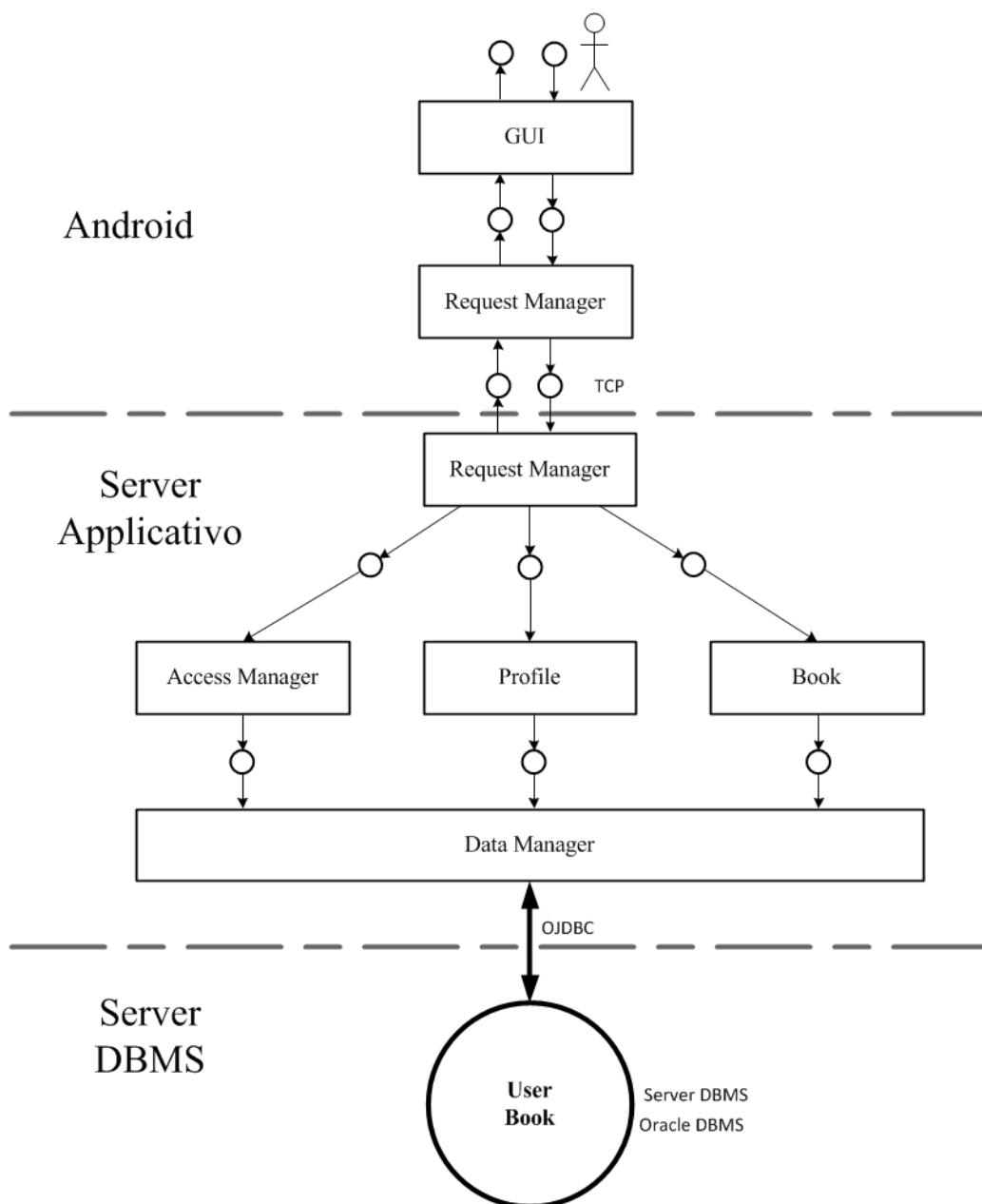


Figure 4.1: Architettura Software

Nella figura 4.1 è mostrata l'architettura software modellizzata attraverso una rete di Petri. Innanzitutto si può già osservare chè è stata definita seguendo il modello architettonale MVC:

- A monte è prevista una parte riservata all'interfaccia grafica, attraverso la quale sarà possibile inviare e ricevere informazioni dal server applicativo. Si vede, infatti, che è prevista una comunicazione bidirezionale tra dispositivo Android e Server.
- Al centro sono rappresentate tutte le richieste a cui è in grado di rispondere. Queste, quindi, saranno funzioni implementate lato Server.
- Infine, è prevista una banca dati persistente, in questo caso un database relazione, al quale il Server Applicativo accede sia per operazioni di lettura che di scrittura, sempre con lo scopo di far fronte alle richieste provenienti a monte.

Si può quindi constatare che non si trattano di strati tra loro indipendenti, poichè il flusso dei dati li coinvolge tutti.

4.2 Logical view

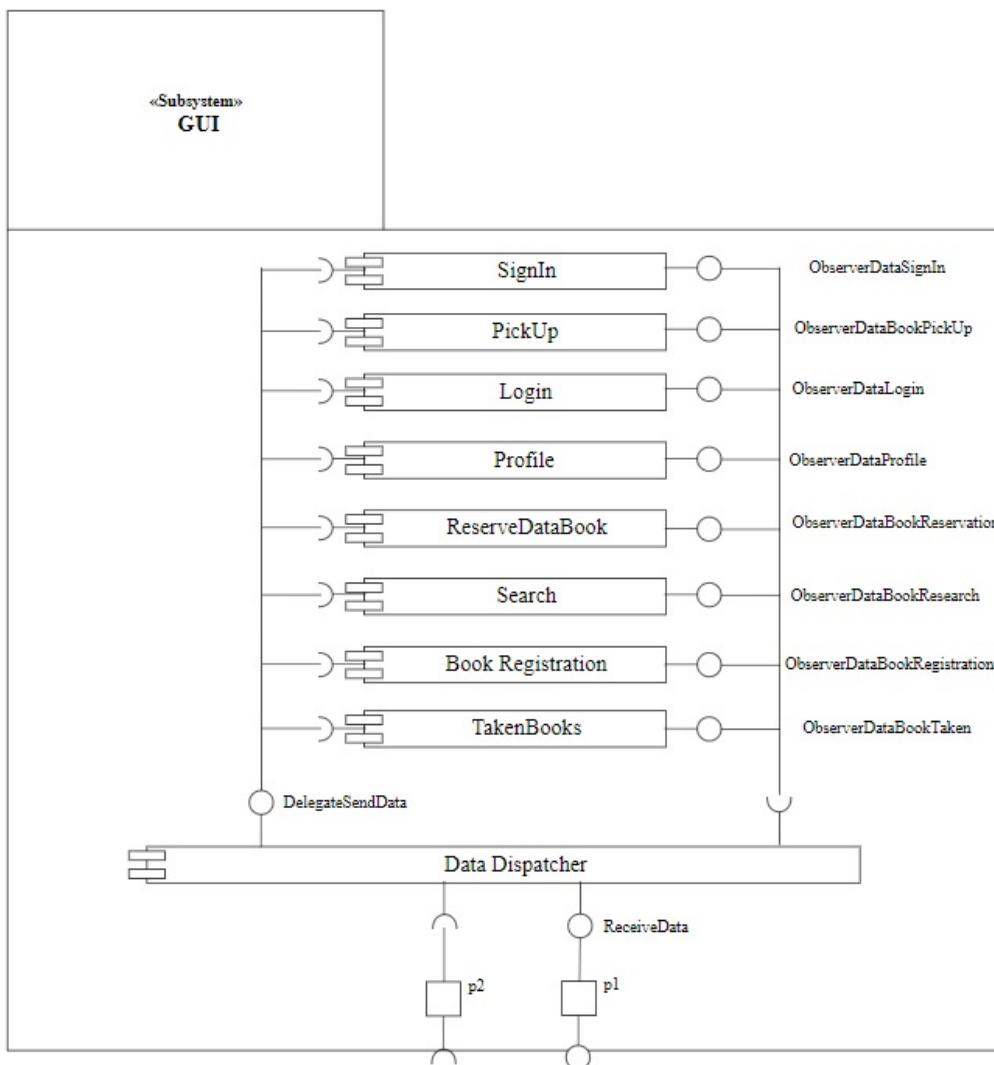


Figure 4.2: Logical view - GUI subsystem

Nelle figure 4.2, 4.3, 4.4 e 4.5 è mostrata in dettaglio la Logical View del sistema progettato. Si può osservare che segue il modello definito attraverso il pattern architettonale Model View Controller, dal momento che vengono individuati tre strati, ciascuno dei quali con le seguenti caratteristiche:

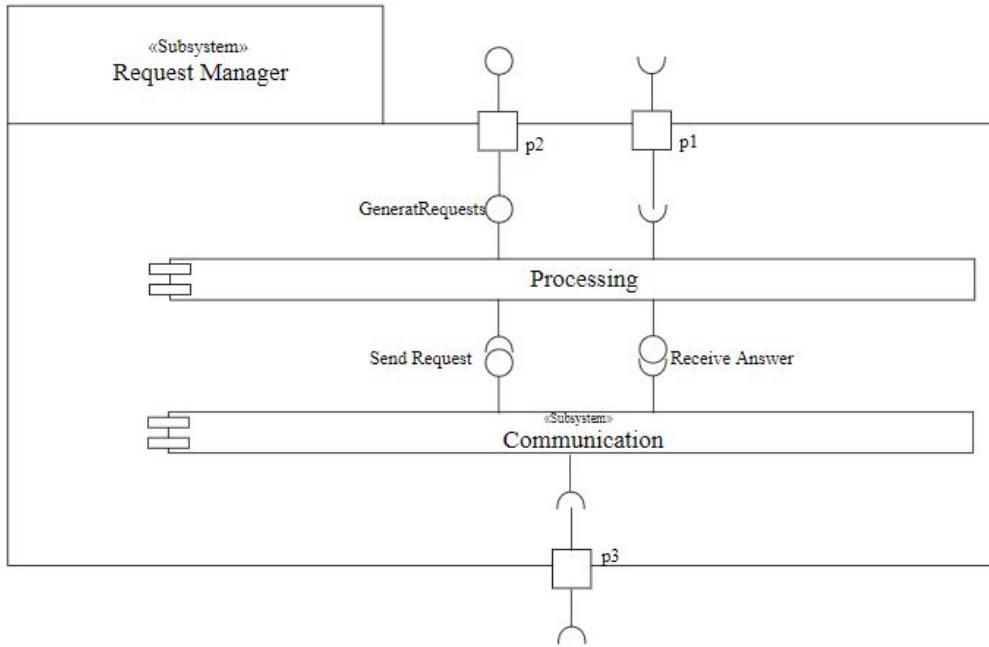


Figure 4.3: Logical view - Request Manager subsystem client side

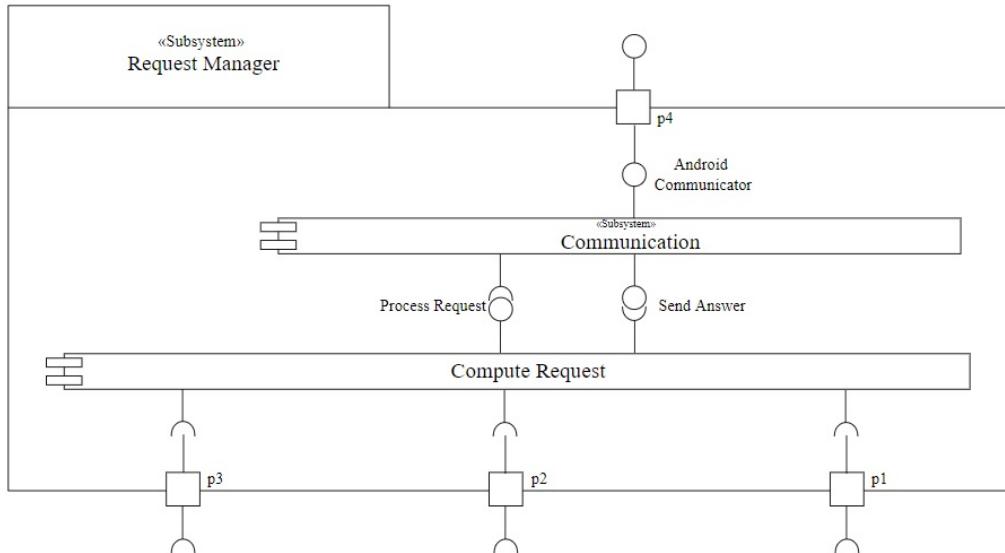


Figure 4.4: Logical view - Request Manager subsystem server side

- **Subsystem "GUI":** rappresenta l'interfaccia grafica con la quale l'applicazione si presenterà. Ciascun componente fa riferimento ad ogni view che l'applicazione può mostrare, e che quindi corrisponde a diversi casi d'uso dell'applicativo stesso, come l'accesso alla rete di Book Crossing (Login) o registrazione di un libro.
Questi componenti saranno quindi ovviamente allocati direttamente sul dispositivo mobile: ogni singolo component (*fragment*) avrà legato ad esso, in maniera intrinseca, anche un file *.xml*, il quale permette di definirne l'interfaccia grafica, ovvero come sono posizionati gli elementi visivi.
TODO: spiegazione sintetica delle funzionalità di ogni singolo fragment
- **Subsystem "Request Manager":** ha il compito di gestire le richieste provenienti da ciascun componente descritto nel subsystem *GUI*. Al suo interno sono indicati i componenti attraverso i quali si risponde alle richieste provenienti dal dispositivo mobile.
Una parte di questo *manager* sarà disposta a bordo del dispositivo, permettendo una preelabo-

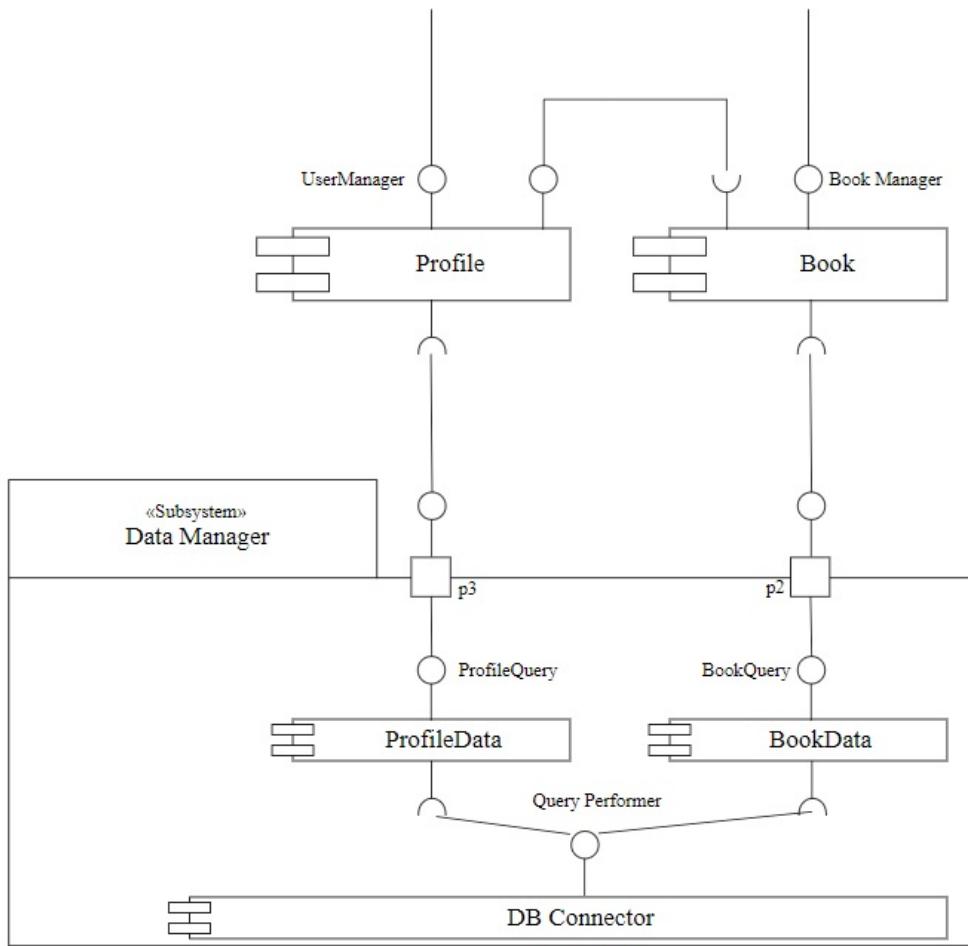


Figure 4.5: Logical view - Server functionality

razione e assemblamento delle richieste; questo subsystem avrà però anche un implementazione *server side*, che gli permetterà di ricevere tali richieste, rieditarla, secondo quello che è la necessità della richiesta, e poi interacciarsi, in un verso o nell'altro, con la parte di persistenza dei dati.

- **Subsystem "Data Manager":** rappresenta la comunicazione con il Database. Sono quindi indicati i componenti con i quali il sistema si interfacerà con la banca dati dell'architettura.

Si vede quindi come ogni parte dell'archiettura abbia un compito ben definito: la parte relativa al subsystem *GUI (Graphic User Interface)*, si occupa di gestire l'interazione con l'utente ricevendo e/o mostrando i dati forniti e/o richieste dall'utilizzatore stesso; al suo interno quindi non troveremo codice di logica applicativa ma solamente componenti di gestione *UI*. Esso rappresenta quindi la parte di *view*. Il subsystem *request manager* invece si occupa di controllare il flusso di dati dall'applicazione al server e viceversa; rappresenta quindi la sezione di *controller* dove è contenuta la *low logic* dell'applicazione. Infine, il terzo ed ultimo componente della struttura MVC, è rappresentato dal subsystem *data manager*, il quale permette di interfacciarsi direttamente con la base di dati, astrendo tutte le operazioni di controllo di accesso al database stesso. Il modello architetturale MVC è stato poi applicato anche successivamente per la progettazione delle componenti previste per ciascun elemento dell'architettura.

5

Analisi dei componenti

5.1 Scelta delle funzionalità da implementare

Come già presentato in precedenza, per la definizione dei componenti si è deciso di seguire il pattern architettonale MVC. Le componenti che si è deciso di sviluppare durante la prima iterazione sono:

- **Componente *Manual Book registration*:** La componente *Manual Book registration* fa riferimento all' UC9 (2.1) (figlio del caso d'uso più generico UC5 (2.1), ovvero alla funzione di aggiunta di un libro alla rete di Book Crossing per via manuale. La componente si presenta nel seguente modo:
 - *GUI*: Interfaccia grafica utilizzata per registrare un libro alla rete di Book Crossing. Verranno quindi messe a disposizione una serie di interfaccie grafiche, composte sostanzialmente da campi da compilare, per aggiungere le informazioni relative al proprio libro, ottenendo poi, successivamente alla registrazione, il relativo BCID;
 - *Model*: Si fa carico di ricevere le informazioni relative al libro e, sfruttando la parte Data, restituisce alla parte GUI il BCID con il quale siglare il libro;
 - *Data*: Le informazioni relative al libro che si vuole aggiungere sono memorizzate nel Database RDS, associandolo all'utente che attualmente lo possiede.
- **Componente *Ricerca*:** La componente *Ricerca* fa riferimento all' UC6 , ovvero alla funzione che permette di andare a ricercare un libro all'interno della piattaforma di Book crossing: questa ricerca può avvenire per titolo, per autore, oppure per entrambi. La componente si presenta nel seguente modo:
 - *GUI*: Interfaccia grafica composta da due text book in cui andare ad inserire titolo e/o autore. Essendo possibili tre tipologie di ricerca, come specificato in precedenza, non è necessario compilare entrambi i campi (lo è solamente nel caso in cui si è interessati a compiere una ricerca basandosi su entrambi i vincoli);
 - *Model*: Gli compete la parte relativa allo smistamento delle richiesta, a seconda del fatto che si stia eseguendo una ricerca per titolo, autore o entrambi;
 - *Data*: Fornisce, se presenti, le informazioni relative al libro oggetto della ricerca.

5.2 Aderenza UML a Logical View

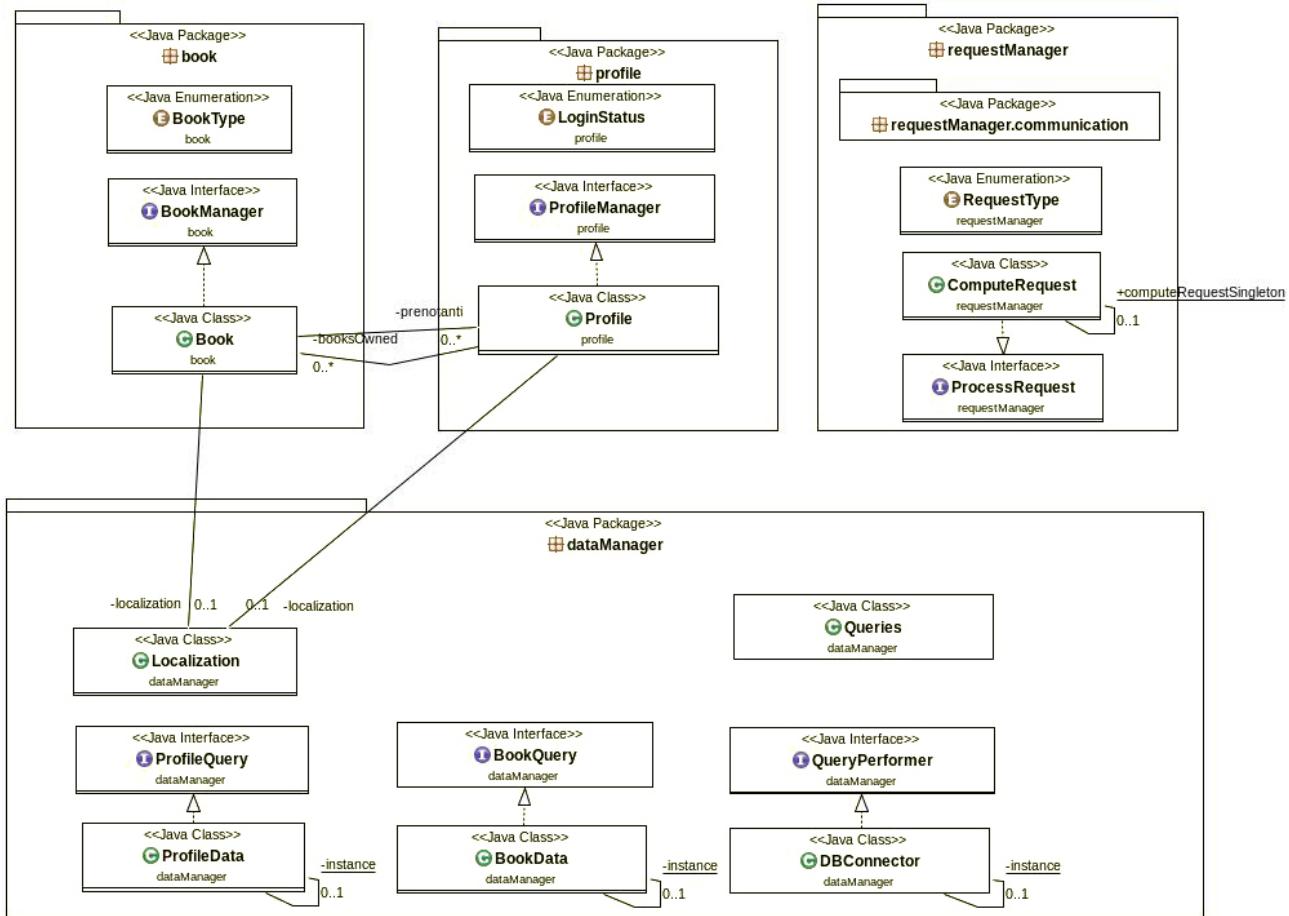


Figure 5.1: class diagram

```

1 public class ComputeRequest implements ProcessRequest{
2
3     ~
4     public final void process(String msg, String username) {
5         // First information is the request type, that
6         // is encoded in this way: REQUESTTYPE:...;
7         int i = msg.indexOf(";", 0);
8         int j = msg.indexOf(":", 0);
9         // Convert the String information of the
10        // request type to an enum
11        @Nullable RequestType requestType = RequestType.
12        getEnumReqType(msg.substring(0, i).substring(j+1));
13
14        if(requestType == null) {
15            // In this case, in the request, there's
16            // a wrong encoded request type
17            Communication.getInstance().send(username,
18                "requestType:" + 10000 + ";result:KO_RequestType");
19        }
20
21        switch(requestType) {
22            case BOOK_REGISTRATION_MANUAL:
23                Book b = new Book(msg.substring(i + 1));
24                b.setActualOwnerUsername(username);
25                b.setISBN("null");
26

```

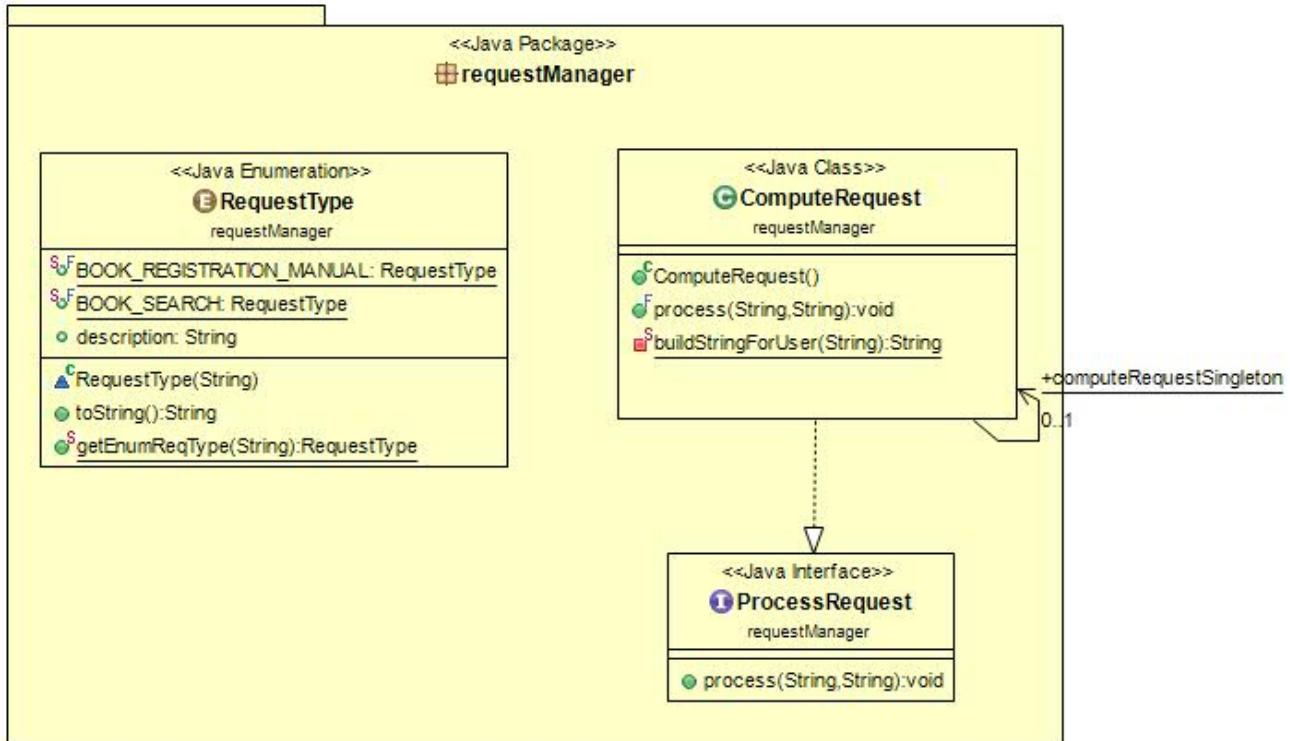


Figure 5.2: Diagramma della classe ComputeRequest

```

27     boolean result = b.insert ();
28     Communication.getInstance ().send (username ,
29     "requestType:0;result:" + (result ? 1 : 0) +
30     ";BCID:" + b.getBCID ());
31     break;
32
33     case BOOK_SEARCH:
34         ~
35         if (type.equals ("TITLE")) {
36             books = BookData.searchBookByTitle (title);
37         } else if (type.equals ("AUTHOR")) {
38             books = BookData.searchBookByAuthor (author);
39         } else {
40             books = BookData.searchBook (title, author);
41         }
42         ~
43         Communication.getInstance ().send (username ,
44         "requestType:8;result:" +
45         + 1 + ";Books:" + jsonResponse);
46         break;
47
48         ~
49     }
50 }
51

```

Così possiamo osservare che la classe `ComputeRequest` si preoccupa di gestire le richieste previste dalla prima iterazione provenienti dal client nel formato descritto.

6

Observer-delegate pattern: la nostra implementazione

L'applicativo, oltre a seguire un design MVC, per la separazione dei componenti, implementa anche un pattern di comunicazione ad eventi, meglio conosciuto come *pattern observer-delegate*. In sostanza, ogni componente che necessita di una certa tipologia di informazioni, si mette in ascolto, registrandosi ad un "servizio" offerto nel nostro caso dal *DataDispatcherSingleton*, il quale va a salvarsi al suo interno un gruppo di riferimenti (delegati) che dovrà poi andare a notificare nel momento in cui risulteranno essere presenti nuove informazioni a cui gli observers stessi sono interessati. Come si può ben capire si tratta quindi di un pattern molto vicino alla struttura *Publish-Subscribe*, con il quale si definisce una dipendenza uno a molti fra gli oggetti: in sostanza abbiano un oggetto che viene *osservato* e tanti oggetti che *osservano* i cambiamenti di quest'ultimo, come si può vedere nella struttura generica rappresentata in figura 6.1:

Questa struttura è stata riprodotta anche nel nostro applicativo, dove possiamo vedere la presenza di alcune struttura caratteristiche di questo pattern: abbiamo una funzionalità di *register*, che, a seconda di quale observer chiama questa funzione, lo pone nel vettore specifico (figura 6.2). Per evitare di avere dipendenze cicliche, o comunque per mantenere una certa pulizia nei riferimenti tra classi, è necessario che ogni observer, nel momento in cui non necessita più di una certa tipologie di informazioni, vada a deregistrarsi: questa operazione consiste nell'andare a togliere il qualsiasi riferimento all'oggetto in esame dall'interno del *DataDispatcherSingleton*, come fatto in figura 6.3.

Per quanto riguarda invece la parte di *notifica*, abbiamo introdotto un pattern ad eventi custom, che permettesse così di mantenere separata l'implementazione delle informazioni ottenuto da ogni singolo fragment, come si vede nella figura 6.4.

Ogni singola interfaccia implementa un'interfaccia comune: questa scelta è stata adotta per rendere del tutto generico il tipo di observer che va a registrarsi per gli eventi forniti dal delegate: infatti, come si può vedere nell'interfaccia *DelegateSendData* (figura 6.5), chi è interessato ad ottenere informazioni specifiche, si registra come un oggetto di tipo *ObserverForUiInformation*; sarà poi compito del delegate fornire le corrette informazioni, richiamando le corrette funzioni di notifica, le quali sono direttamente implementata all'interno di ogni singola interfaccia concreta.

Si può quindi vedere che, nella struttura predisposta, il *DataDispatcher* lavora come se fosse un **repository**, ovvero un contenitore di informazioni, che fornisce a tutti gli elementi che si registrano e che ne richiedono la ricezione.

Questi elementi, allo stadio implementativo attuale, sono rappresentati da tutti i fragment, che necessitano di ricevere/inviare informazioni per soddisfare l'interazione con l'utente: come si può notare nelle figure 6.6- 6.6- 6.7- ??- ??- ??- ??, ogni singolo fragment è predisposto per ricevere le informazioni di cui ha bisogno, tramite la chiamata delle callback da parte del dispatcher stesso.

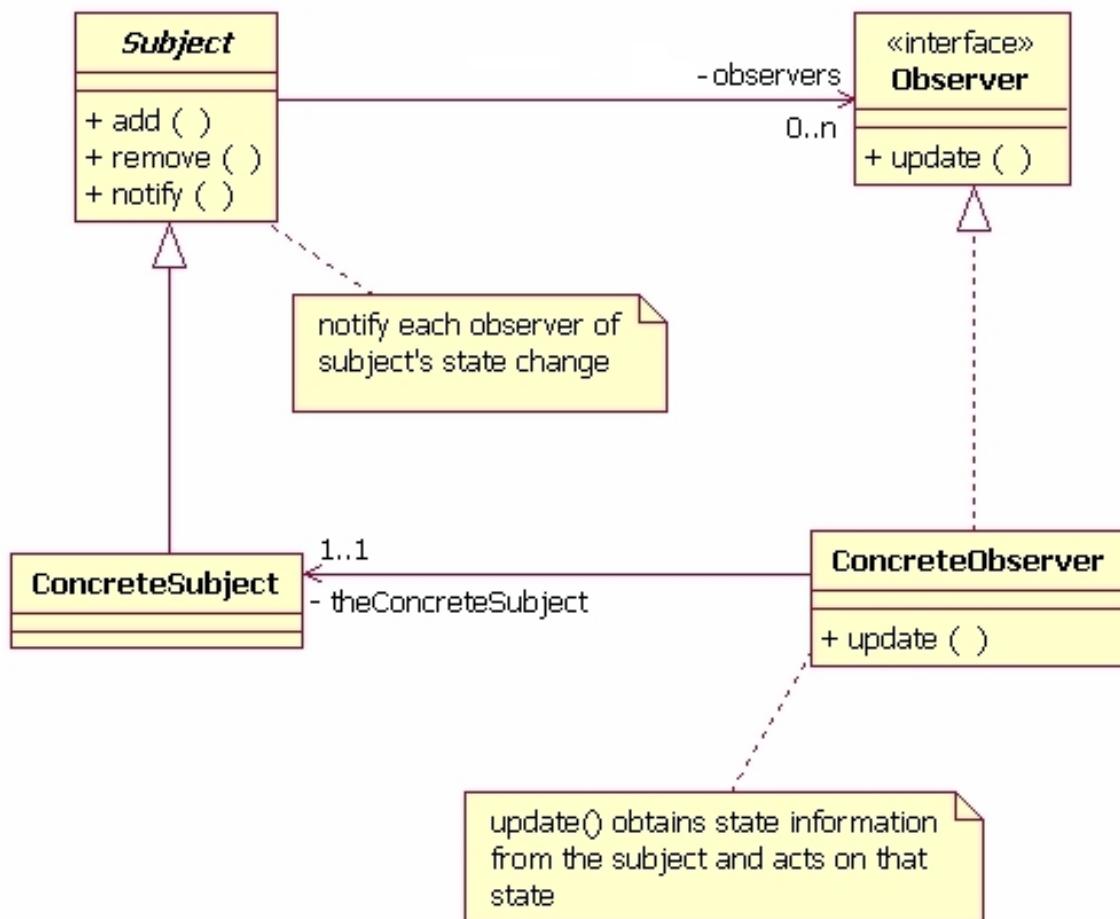


Figure 6.1: Struttura generica pattern observer

```

@Override
public void register(ObserverForUiInformation observerForUiInformation) {
    if ((observerForUiInformation instanceof ObserverDataBookRegistration) && (!observersDataBookRegistration.contains(observerForUiInformation))) {
        observersDataBookRegistration.add((ObserverDataBookRegistration) observerForUiInformation);
    } else if ((observerForUiInformation instanceof ObserverDataBookPickUp) && (!observersDataBookPickUp.contains(observerForUiInformation))) {
        observersDataBookPickUp.add((ObserverDataBookPickUp) observerForUiInformation);
    } else if ((observerForUiInformation instanceof ObserverDataBookTaken) && (!observersDataBookTaken.contains(observerForUiInformation))) {
        observersDataBookTaken.add((ObserverDataBookTaken) observerForUiInformation);
    } else if ((observerForUiInformation instanceof ObserverDataLogin) && (!observersDataLogin.contains(observerForUiInformation))) {
        observersDataLogin.add((ObserverDataLogin) observerForUiInformation);
    } else if ((observerForUiInformation instanceof ObserverDataSignIn) && (!observersDataSignIn.contains(observerForUiInformation))) {
        observersDataSignIn.add((ObserverDataSignIn) observerForUiInformation);
    } else if ((observerForUiInformation instanceof ObserverDataProfile) && (!observersDataProfile.contains(observerForUiInformation))) {
        observersDataProfile.add((ObserverDataProfile) observerForUiInformation);
    } else if ((observerForUiInformation instanceof ObserverDataBookResearch) && (!observersDataBookResearch.contains(observerForUiInformation))) {
        observersDataBookResearch.add((ObserverDataBookResearch) observerForUiInformation);
    } else if ((observerForUiInformation instanceof ObserverDataBookReservation) && (!observersDataBookReservation.contains(observerForUiInformation))) {
        observersDataBookReservation.add((ObserverDataBookReservation) observerForUiInformation);
    }
}
  
```

Figure 6.2: Register function

```

@Override
public boolean unRegister(ObserverForUiInformation observerForUiInformation) {
    if (observerForUiInformation instanceof ObserverDataBookRegistration) {
        return observersDataBookRegistration.remove(observerForUiInformation);
    } else if (observerForUiInformation instanceof ObserverDataBookPickUp) {
        return observersDataBookPickUp.remove(observerForUiInformation);
    } else if (observerForUiInformation instanceof ObserverDataBookTaken) {
        return observersDataBookTaken.remove(observerForUiInformation);
    } else if (observerForUiInformation instanceof ObserverDataLogin) {
        return observersDataLogin.remove(observerForUiInformation);
    } else if (observerForUiInformation instanceof ObserverDataProfile) {
        return observersDataProfile.remove(observerForUiInformation);
    } else if (observerForUiInformation instanceof ObserverDataSignIn) {
        return observersDataSignIn.remove(observerForUiInformation);
    } else if (observerForUiInformation instanceof ObserverDataBookResearch) {
        return observersDataBookResearch.remove(observerForUiInformation);
    } else if (observerForUiInformation instanceof ObserverDataBookReservation) {
        return observersDataBookReservation.remove(observerForUiInformation);
    }
    return false;
}

```

Figure 6.3: Unregister function



Figure 6.4: Interfaccie per la ricezione degli eventi

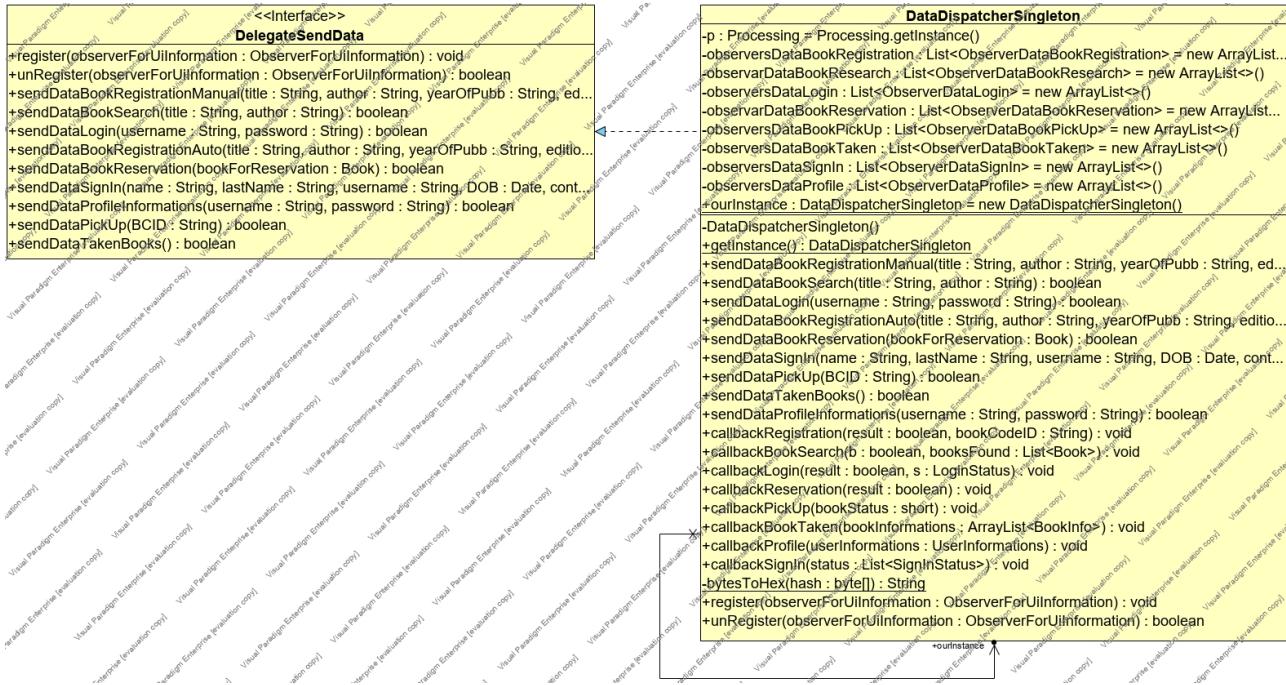


Figure 6.5: Struttura del componente *DataDispatcher*

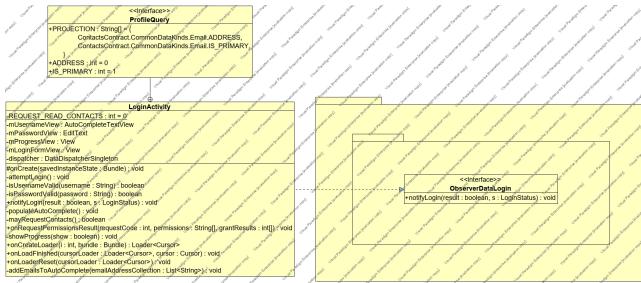


Figure 6.6: Struttura del componente *LoginFragment*

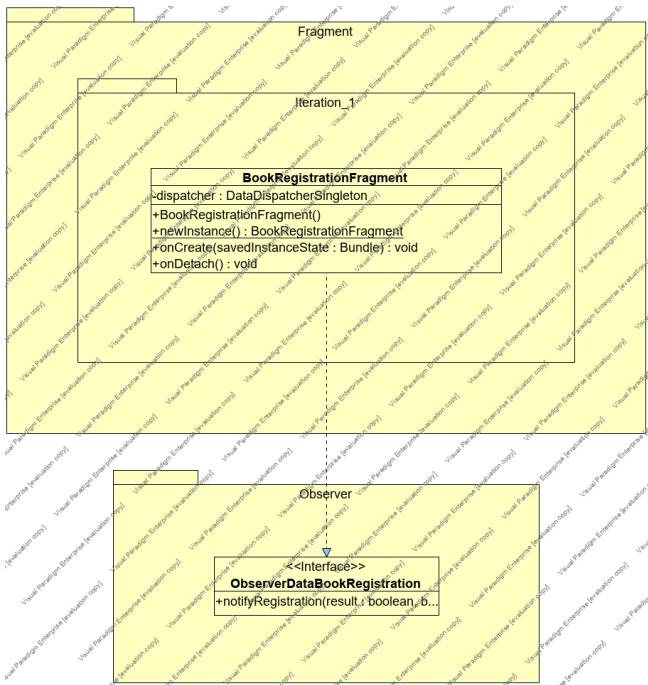
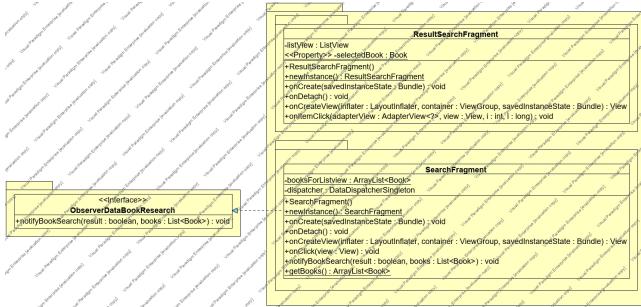
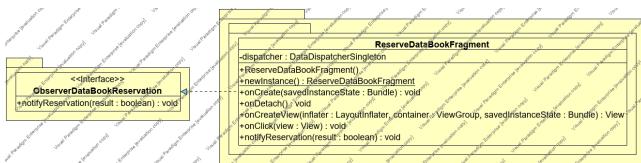


Figure 6.7: Struttura del componente *BookRegistrationFragment*

Figure 6.8: Struttura del componente *ResearchFragment*Figure 6.9: Struttura del componente *ReservationFragment*

Implementazione comunicazione server

7.1 il framework *Netty*

Netty è un NIO (Non-blocking Input Output) client server framework (i.e. permette una comunicazione asincrona tra client e server di tipo event-driven) che facilita lo sviluppo di applicazioni di rete che utilizzano socket TCP e UDP.

Il framework è disponibile sul Maven Central Repository e può essere scaricato includendo la seguente dependency nel pom.xml del progetto:

```

1 <dependencies>
2     <dependency>
3         <groupId>io.netty</groupId>
4         <artifactId>netty-all</artifactId>
5         <version>4.1.32.Final</version>
6     </dependency>
7 </dependencies>
```

Lato Server creiamo un oggetto singleton *Communication* che inizializza un oggetto di tipo *ServerBootstrap* (io.netty.bootstrap.ServerBootstrap) il quale effettua il bootstrap del *ServerChannel* ovvero del link "primario" (sulla porta 5000) che accetta ogni connessione di un client e crea un canale "figlio" sul quale avverrà la comunicazione:

```

1 private Communication() throws Exception {
2     ~
3     ServerBootstrap b = new ServerBootstrap();
4
5     b.group(bossGroup, workerGroup).channel(NioServerSocketChannel.class)
6     .handler(new LoggingHandler(LogLevel.INFO)).childHandler(
7         new ServerInitializer(sslCtx));
8
9     b.bind(PORT).sync().channel().closeFuture().sync();
10    ~
11 }
```

al server aggiungiamo un handler *ServerInitializer* che aggiunge alla pipeline, oltre agli encoder e decoder di default, il *ServerHandler*

```

1 class ServerInitializer extends ChannelInitializer<SocketChannel> {
2
3     private static final StringDecoder DECODER = new StringDecoder();
4     private static final StringEncoder ENCODER = new StringEncoder();
5     private static final ServerHandler SERVER_HANDLER = new ServerHandler();
6
7     @Override
8     public void initChannel(SocketChannel ch) throws Exception {
9         ChannelPipeline pipeline = ch.pipeline();
10        ~
11         pipeline.addLast(DECODER);
```

```

12         pipeline.addLast(ENCODER);
13         // and then business logic.
14         pipeline.addLast(SERVER_HANDLER);
15     }
16 }
```

la classe *ServerHandler* è un sottotipo di *SimpleChannelInboundHandler* (`io.netty.channel.SimpleChannelInboundHandler`) il quale fornisce la stringa inviata dal client tramite il metodo:

```

1 class ServerHandler extends SimpleChannelInboundHandler<String> {
2     ~
3     @Override
4     public void channelRead0(ChannelHandlerContext ctx, String request)
5             throws Exception {
6         int j = request.indexOf(";");
7         String username = request.substring(0, j);
8         Communication.chcMap.put(username, ctx);
9
10        System.out.println("Process\u00e0request:\u00e0" + request);
11        computeRequest.process(request.substring(j + 1), username);
12    }
13    ~
14 }
```

l'attento lettore avrà notato che il metodo precedente memorizza la coppia (`username`, `channelHandlerContext`) all'interno dell'oggetto `chcMap` di *Communication*, il quale è di tipo `HashMap<String, ChannelHandlerContext>`, tale informazione è fondamentale al fine di rispondere al clinet tramite canale "figlio" ad esso dedicato e chiuderlo al termine della comunicazione; il metodo invoca l'elaborazione del messaggio ricevuto tramite l'interfaccia *ProcessRequest* (la semantica della stringa `request` verrà discussa nel paragrafo successivo).

Infine, ma non per importanza, discutiamo l'implementazione dell'interfaccia *SendAnswer* da parte della classe *Communication*:

```

1 public final class Communication implements SendAnswer{
2     ~
3     public void send(final String username, final String msg) {
4         ~
5         ChannelFuture oo = chcMap.get(username).writeAndFlush(msg + "\r\n");
6
7         oo.addListener(new ChannelFutureListener() {
8
9             public void operationComplete(ChannelFuture future)
10                     throws Exception {
11                 int count = 0;
12                 boolean isError = false;
13
14                 while (!future.isSuccess()) {
15                     chcMap.get(username).writeAndFlush(msg + "\r\n");
16                     System.out.println("Retry\u00e0send\u00e0response!");
17                     if(count == 3) {
18                         isError = true;
19                         break;
20                     } else
21                         count++;
22                 }
23                 ~
24
25                 future.addListener(ChannelFutureListener.CLOSE);
26                 chcMap.remove(username);
27             }
28         });
29     }
30 }
```

```

28
29         }
30     }
31     ~
32 }

```

il metodo send si occupa di inviare la risposta relativa alla richiesta ricevuta e nel caso di fallimento ripeterà l'invio fino a un limite di 3 tentativi.

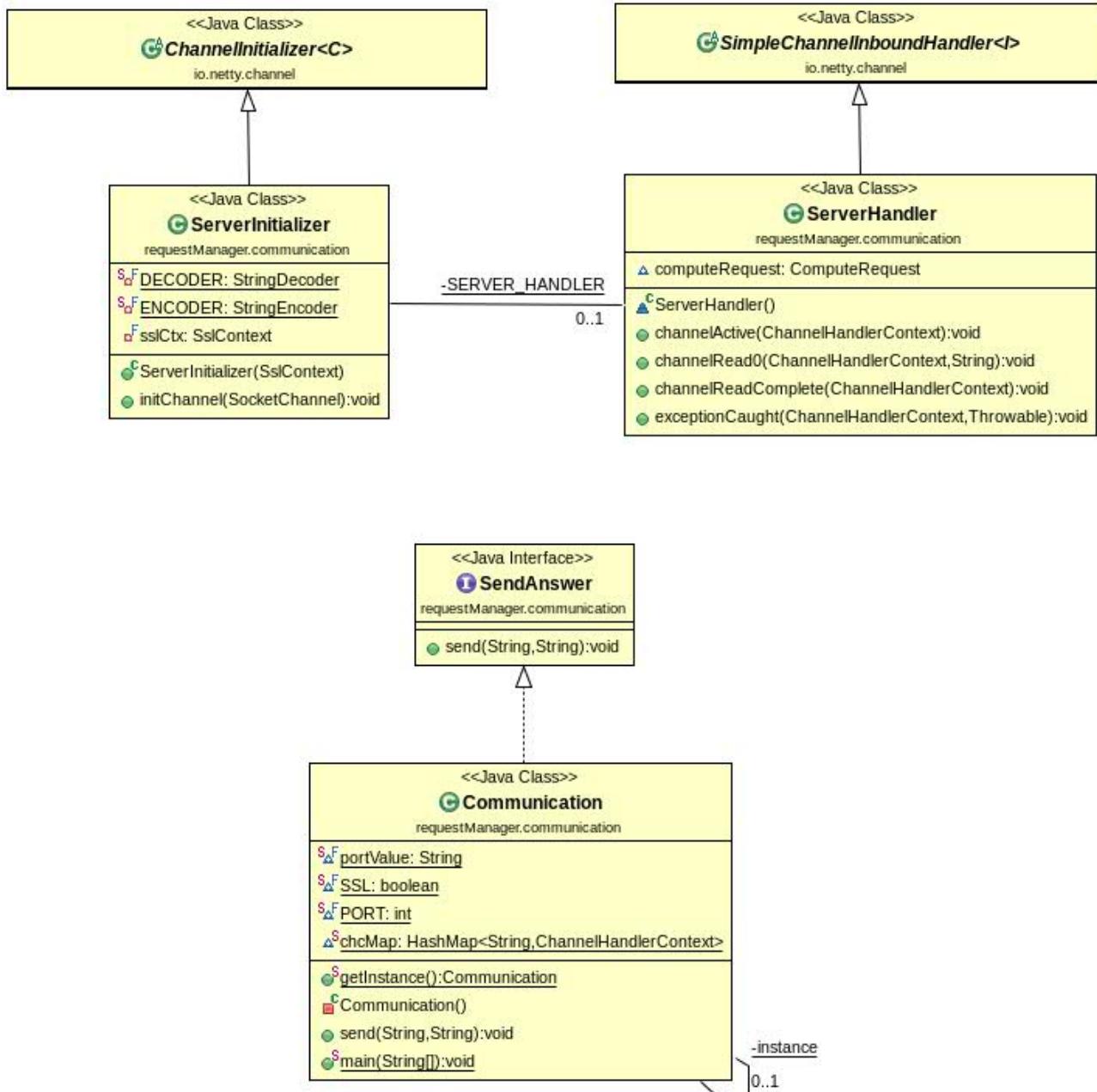


Figure 7.1: Struttura del package `requestManager.Communication`

Lato Client creiamo un singleton *Communication* che implementa il seguente metodo dell’interfaccia *SendRequest* aprendo la connessione con il Server sulla socket (35.180.103.132,5000) e inviando il messaggio al server:

```

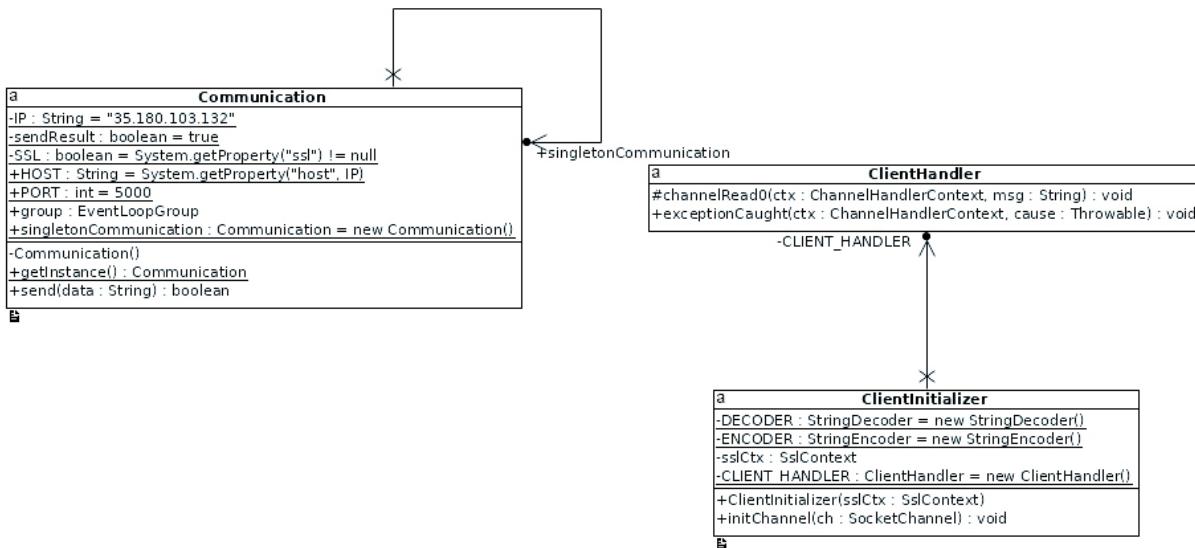
1 public class Communication implements SendRequest {
2     private static final String IP = "35.180.103.132";
3     public static final String HOST = System.getProperty("host", IP);
4     public static final int PORT = 5000;
5     ~
6     @Override
7     public boolean send(String data) {
8         ~
9         Bootstrap b = new Bootstrap();
10        b.group(group).channel(NioSocketChannel.class).handler(new
11                               ClientInitializer(sslCtx));
12        ChannelFuture ch = null;
13        ~
14        ch = b.connect(HOST, PORT);
15        ~
16        ChannelFuture lastWriteFuture = null;
17        lastWriteFuture = ch.channel().writeAndFlush(data + "\r\n");
18
19        //Wait until all messages are flushed before closing the channel
20        if (lastWriteFuture != null) {
21            ~
22            lastWriteFuture.sync();
23            ~
24        }
25    }
26    ~
27 }
```

Tralasciamo la discussione dell’implementazione della classe *ClientInitializer* in quanto simile a quella della classe duale; invece approfondamo l’implementazione di *ClientHandler*:

```

1 class ClientHandler extends SimpleChannelInboundHandler<String> {
2     @Override
3     protected void channelRead0(ChannelHandlerContext ctx, String msg) {
4         Processing.getInstance().processAnswer(msg);
5         Communication.getInstance().group.shutdownGracefully();
6     }
7     ~
8 }
```

di particolare interesse è il metodo *channelRead0* ereditato da *SimpleChannelInboundHandler*, tale metodo viene invocato dal framework ogni volta il client riceve un messaggio, quest’ultimo viene elaborato dal singleton *Processing* e il canale di comunicazione viene chiuso.

Figure 7.2: Struttura del package `requestManager.Communication`

7.2 Semantica dei messaggi

I messaggi inviati dai client sono stringhe che possono essere identificate dalla seguente *RegEx* (*Regular Expression*):

1 $^{\wedge} [a-zA-Z0-9]+ ; [a-zA-Z0-9]+ : [0-9]+ ; ([a-zA-Z0-9]+ : [a-zA-Z0-9]+ ;) + \$$

la cui semantica può essere così rappresentata:

`"<username>;requestType:<tipo>;<richiesta>"`

I valori assunti da `<tipo>` sono i seguenti e corrispondono ai campi definiti all'interno dell'enumerativo `RequestType`. Durante l'attuale iterazione siamo andati a gestire le richieste relative alle sole due seguenti tipologie di richieste:

- 0 -> BOOK_REGISTRATION_MANUAL
- 8 -> BOOK_SEARCH

Nella seconda iterazione, e nelle successive, è stata pianificata l'implementazione della parte di gestione delle restanti richieste:

- 1 -> BOOK_RESERVATION
- 2 -> LOGIN
- 3 -> SIGN_IN
- 4 -> BOOK_REGISTRATION_AUTOMATIC
- 5 -> PROFILE_INFO
- 6 -> TAKEN_BOOKS
- 7 -> PICK_UP

La richiesta inviata dal client Android verso il server, ad esempio, per la ricerca di un libro assume la seguente forma:

```
1           username + ";" + "requestType:" + 0 + ";" + book.encode();
```

Lato Android, la richiesta vera e propria è preceduta dall'username dell'utente collegato in modo che lato server, qualora, per diversi motivi, l'username fosse invalido, la richiesta venga ignorata a priori. Analogamente possiamo identificare i messaggi inviati dal server come:

```
1           ^[a-zA-Z0-9]+:[0-9]+;([a-zA-Z0-9]+:[a-zA-Z0-9]+;)+$
```

la cui semantica può essere così rappresentata:

```
"requestType:<tipo>;<risultato>"
```

Ad esempio, nel caso di invio di una richiesta da parte del server verso lato Android in merito alla registrazione manuale di un libro andata a buon fine, la stringa assume il seguente formato:

```
1           "requestType:0;result:" + 1 + ";BCID:" + bcid
```

dove *bcid* è il codice alfanumerico generato casualmente dal server per identificare il libro all'interno della rete di Book Crossing. Qual'ora, invece, il server riceva una richiesta con un tipo errato la risposta del server è la seguente:

```
1           "requestType:10000;result:KO_RequestType"
```

8

Creazione della base di dati

Le seguenti sono le query ddl per la creazione del database che abbiamo utilizzato:

```
1 — tables
2 — Table: Libro
3 CREATE TABLE Libro (
4 BCID varchar2(15) NOT NULL,
5 Titolo varchar2(256) NOT NULL,
6 Autore varchar2(256) NOT NULL,
7 DataPubblicazione VARCHAR2(20 BYTE) NULL,
8 ISBN varchar2(13),
9 Proprietario varchar2(20) NOT NULL,
10 Genere varchar2(256),
11 Longitudine number(38,0),
12 Latitudine number(38,0),
13 CONSTRAINT Libro_pk PRIMARY KEY (BCID)
14 ) ;
15
16 — Table: Possesso
17 CREATE TABLE Possesso (
18 Username varchar2(20) NOT NULL,
19 bcid varchar2(15) NOT NULL,
20 DataInizio date NOT NULL,
21 DataFine date,
22 LuogoRilascio varchar2(512),
23 CONSTRAINT Possesso_pk PRIMARY KEY (Username, bcid)
24 ) ;
25
26 — Table: Prenotazione
27 CREATE TABLE Prenotazione (
28 Username varchar2(20) NOT NULL,
29 bcid varchar2(15) NOT NULL,
30 ID number(38, 0) NOT NULL,
31 CONSTRAINT Prenotazione_pk PRIMARY KEY (ID)
32 ) ;
33
34 — Table: Utente
35 CREATE TABLE Utente (
36 Username varchar2(20) NOT NULL,
37 Nome varchar2(50) NOT NULL,
38 Cognome varchar2(50) NOT NULL,
39 DataNas date NOT NULL,
40 Contatto varchar2(256) NULL,
41 Password char(256) NOT NULL,
42 ResidenzaLat integer NOT NULL,
43 ResidenzaLong integer NOT NULL,
44 RaggioAzione integer NOT NULL,
```

```
45 CONSTRAINT Utente_pk PRIMARY KEY (Username)
46 ) ;
47
48 — Table: Passaggio
49 CREATE TABLE Passaggio (
50 Utenti varchar2(100) NOT NULL,
51 ID number(38, 0) NOT NULL,
52 CONSTRAINT Passaggio_pk PRIMARY KEY (ID)
53 );
54
55 — foreign keys
56 — Reference: Libro_Utente (table: Libro)
57 ALTER TABLE Libro ADD CONSTRAINT Libro_Utente
58 FOREIGN KEY (Proprietario)
59 REFERENCES Utente (Username);
60
61 — Reference: Possiede_Libro (table: Possesso)
62 ALTER TABLE Possesso ADD CONSTRAINT Possiede_Libro
63 FOREIGN KEY (bcid)
64 REFERENCES Libro (BCID);
65
66 — Reference: Possiede_Profilo (table: Possesso)
67 ALTER TABLE Possesso ADD CONSTRAINT Possiede_Profilo
68 FOREIGN KEY (username)
69 REFERENCES Utente (Username);
70
71 — Reference: Prenotazione_Libro (table: Prenotazione)
72 ALTER TABLE Prenotazione ADD CONSTRAINT Prenotazione_Libro
73 FOREIGN KEY (bcid)
74 REFERENCES Libro (BCID);
75
76 — Reference: Prenotazione_Utente (table: Prenotazione)
77 ALTER TABLE Prenotazione ADD CONSTRAINT Prenotazione_Utente
78 FOREIGN KEY (username)
79 REFERENCES Utente (Username);
80
81 ALTER TABLE Passaggio ADD CONSTRAINT Passaggio_Prenotazione
82 FOREIGN KEY (ID)
83 REFERENCES Prenotazione (ID);
84
85 — End of file .
```

Part III

Iterazione 2

9

Analisi dei componenti

Le componenti che invece si è deciso di andare a sviluppare durante la seconda iterazione sono:

- **Componente *Automatic Book registration*:** La componente *Automatic Book registration* fa riferimento all' UC10 (figlio del caso d'uso più generico UC5); questa funzionalità consente di ottenere, in maniera del tutto automatica, tutte le informazioni necessarie per registrare uno specifico libro all'interno della piattaforma di book crossing. La componente si presenta nel seguente modo:
 - *GUI*: Interfaccia grafica utilizzata per registrare un libro, in maniera automatica, nella rete di Book Crossing. Verranno quindi messe a disposizione una serie di interfaccie grafiche, le quali permetteranno di scansionare il codice ISBN del libro desiderato, unitamente ad una parte GUI utilizzata per mostrare le informazioni relative al libro che è appena stato scansionato.
 - *Model*: Si fa carico di ricevere le informazioni relative al libro e, sfruttando la parte Data, restituisce alla parte GUI il BCID con il quale siglare il libro;
 - *Data*: Le informazioni relative al libro che si vuole aggiungere sono memorizzate nel Database RDS, associandolo all'utente che attualmente lo possiede.
- **Componente *Login*:** La componente *Login* fa riferimento all' UC13 , ovvero alla funzione che permette ad un utilizzatore dell'applicazione di loggarsi all'interno della piattaforma di Book Crossing, potendo così effettuare operazioni di suo interesse sui testi disponibili. La componente si presenta nel seguente modo:
 - *GUI*: Interfaccia grafica, composta da due caselle di testo, le quali devono essere riempite con username e password, unitamente ad un pulsante che permette di inviare la richiesta/verifica di login corretto. Per chi non fosse già registrato, è fornita la possibilità di iscriversi alla piattaforma (questo però rappresenta un caso d'uso separato);
 - *Model*: Si fa carico di verificare la coerenza dei dati inseriti tramite il componente grafico, restituendo l'esito a chi ha appena tentato di effettuare il login.
 - *Data*: Le informazioni relative all'utente che sta tentando di loggarsi.
- **Componente *Book reservation*:** La componente *Book reservation* fa riferimento all' U13 (2.1), ovvero alla funzione di prenotazione di un libro, la quale richiede prima un'operazione di ricerca dello stesso all'interno della community, e poi, se possibile, permette di effettuare la prenotazione effettiva. La componente si presenta nel seguente modo:
 - *GUI*: Interfaccia grafica utilizzata per poter procedere con la prenotazione di un libro della rete di Book Crossing. In questo caso sarà possibile compiere tale azione attraverso una procedura di ricerca del libro, oppure navigando nella propria sezione personale del profilo;
 - *Model*: Si fa carico di gestire la coda relativa alla prenotazione di un determinato libro, in modo da poterle soddisfare. La gestione e la computazione della prenotazione è affidata

ad un algoritmo il quale va ad appoggiarsi alla parte Data per poter risalire ai dati dei richiedenti;

- *Data*: Le informazioni relative al libro che si vuole prenotare e agli utenti richiedenti le quali sono memorizzate nel Database.

Partendo dalla figura 5.2 relativa a quanto sviluppato nella precedente iterazione possiamo ora osservare che nella successiva il medesimo diagramma delle classi si completa nel seguente modo:

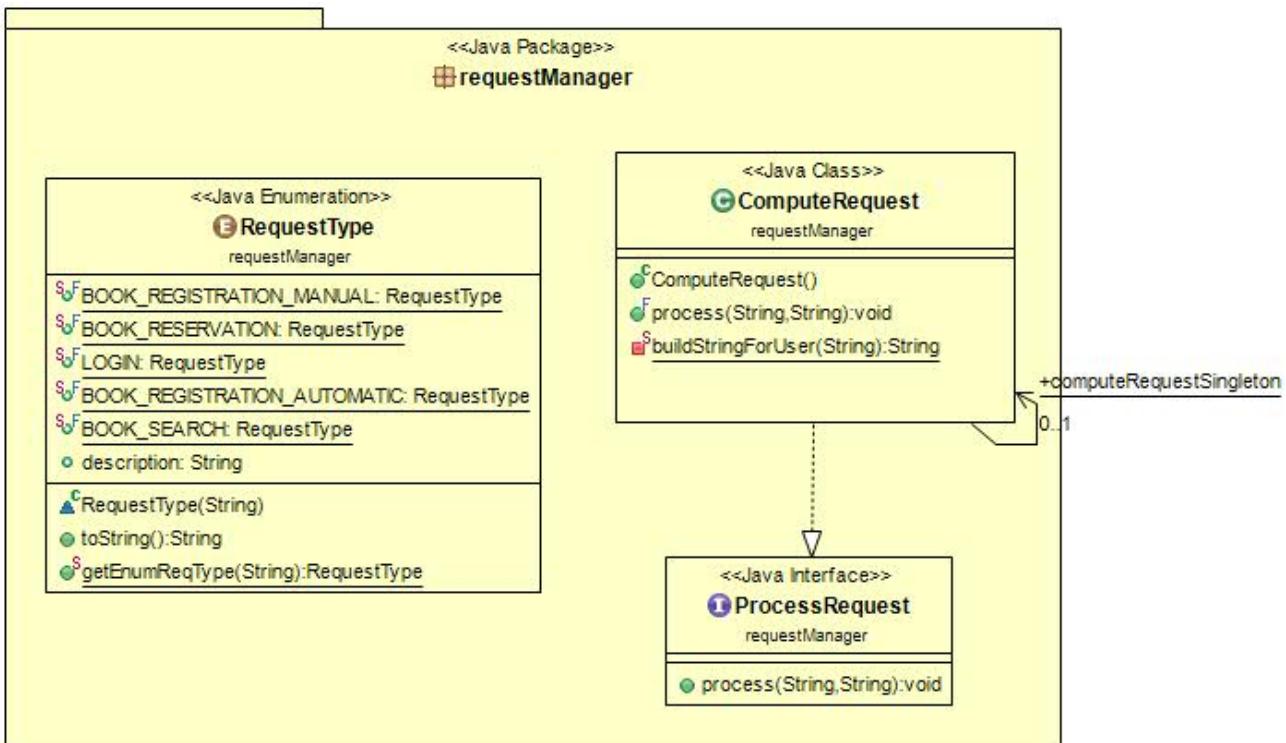


Figure 9.1: Diagramma della classe `ComputeRequest`

Si può notare come alla ***Request Type*** si aggiungono 3 casi in più, relativi proprio ai nuovi componenti sviluppati in questa seconda iterazione. Allo stesso modo anche il codice //TODO:reference to code si completa così

```

1 case BOOK_RESERVATION:
2     ~
3     boolean rs = book.reserve(username);
4     Communication.getInstance().send(username, "requestType:1;result:" +
5         (rs ? 1 : 0));
6     break;
7 case LOGIN:
8     ~
9     Communication.getInstance().send(username, "requestType:2;result:" +
10        "Success" + ";" + buildStringForUser(username));
11    ~
12    break;
13 case BOOK_REGISTRATION_AUTOMATIC:
14     ~
15     result = b.insert();
16     Communication.getInstance().send(username, "requestType:4;result:" +
17         (result ? 1 : 0) + ";BCID:" + b.getBCID());
18     break;

```

Questo si va ad aggiungere a quanto sviluppato nella prima iterazione; a questo punto possiamo quindi gestire le richieste del client compilate nel formato descritto relative ai componenti disponibili nella prima e seconda iterazione.

10

Parte algoritmica: *reservation handler*

La gestione delle prenotazioni dei libri è una delle parte innovative introdotte dalla start up: questo servizio mira a sfruttare la flessibilità della community di sharing, basata sull'ideale di open-source, per comunque offrire un servizio mirato ed attento alle necessità del lettore. Ogni utente, purchè sia registrato all'interno del servizio di Book-sharing, può prenotare un determinato libro che si trova nello stato "Under reading".

Andiamo ad evidenziare gli attori coinvolti in questa operazione:

- **Lettore [L]:** esso rappresenta l'utente, registrato nella community, che possiede il libro oggetto della prenotazione. Indichiamo con:
 - r_L : raggio d'azione del lettore;
 - z_0 : zona di residenza (espressa come coordinate puntuali).
- **Prenotanti** [P_i con $i = 1, \dots, N$]: rappresentano l'insieme degli utenti, tutti interessati ad uno specifico libro in possesso dell'utente **L**. Nello specifico l'indice *i-esimo* indica l'ordine temporale con cui sono giunte le prenotazioni per lo specifico libro. Oltre a questa informazione, ogni utente, rappresentando un generico utente della community, avrà fornito, al momento della registrazione, le seguenti informazioni:
 - r_i^P con $i = 1, \dots, N$: raggio d'azione del lettore;
 - z_i^P con $i = 1, \dots, N$: zona di residenza.

L'algoritmo può essere scomposto in due macro-blocchi:

- **Step 0:** questo fase viene realizzata nel momento in cui il sistema inizia ad analizzare tutti gli utenti che hanno effettuato una prenotazione per un determinato libro che si trova nello stato di "Under reading".

Tutti gli N prenotanti P_i vengono ordinati in base alla distanza dal lettore L , indipendentemente da quello che è l'ordine temporale con cui è stata effettuata la prenotazione: la quantità di cui si terrà (come si può vedere nella figura 10.1) conto sarà quindi la distanza

$$|z_i^p - z_0| \quad (10.1)$$

Questo ordinamento corrisponde quindi sostanzialmente a creare una *priority queue* in cui si va ad assegnare una maggiore priorità all'utente la cui zona di residenza è più vicina a quella del lettore in possesso del libro richiesto.

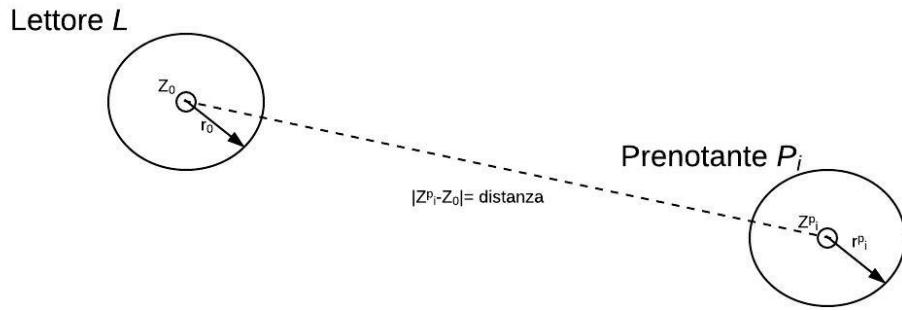


Figure 10.1: Distanza tra lettore e prenotante i-esimo

– **Step 1:** in questo macro-blocco andiamo effettivamente ad applicare l'algoritmo *smart* per poter soddisfare, nella maniera migliore, le esigenze di ogni utente della community.

L'idea di base è che, se lettore e prenotante hanno possibilità di incontrarsi, ovvero se i loro raggi d'azione si sovrappongono, essi potranno accordarsi direttamente sul luogo dello scambio, rendendo "*safety*" il passaggio del libro: questo scambio avverrà ovviamente in una zona all'interno dell'intersezione dei raggi d'azione, come mostrato in figura 10.2.

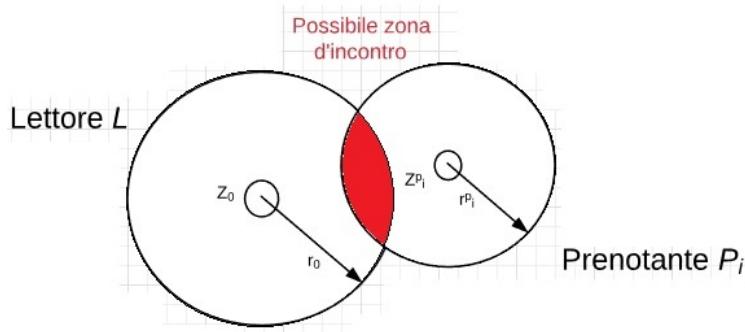


Figure 10.2: Zona d'incontro tra lettore e prenotante

Nel caso in cui invece, i due utenti interessati non abbiano la possibilità di trovare un luogo comune in cui potersi scambiare il libro direttamente, si avrà che, la rete di users appartenenti alla community farà da tramite, per portare il libro "*coast-to-coast*". Quindi, tramite un semplice pseudo-codice, possiamo scrivere il nostro algoritmo come

Algorithm 1: Algoritmo di gestione della prenotazione

Data: Users and their informations

Result: Optimum path from reader to reserver

Step 0 (initialization);

if $Distanza \leq 0$ **then**

Trovare un punto d'incontro nell'unione delle aree;

Notificare gli utenti di dove potersi scambiare direttamente il libro;

else

Creare la rete di utenti che faranno da tramite tra lettore e prenotante;

Ricercare il cammino ottimo (il libro si muoverà *hand-to-hand*);

end

Nello specifico il calcolo della distanza avverrà tramite la funzionalità *checkOverlap(Lettore,*

Prenotante), la quale andrà a verificare che:

$$|z_i^p - z_0| - r_0 - r_i \leq 0 \quad (10.2)$$

ovvero che i raggi d'azione si sovrappongano o meno.

Nel caso in cui i due utenti non abbiano possibili punti d'incontro ($\text{Distanza} \geq 0$), dobbiamo selezionare gli utenti tramite il quale il libro in questione potrà viaggiare: l'idea base è quindi quella di costruirsi un'area circolare di centro pari alla metà della congiungente del punto z_0 (zona di residenza lettore) e z_i^p (zona di residenza prenotante). Andremo poi a selezionare tutti gli utenti che si trovano all'interno di questa circoscrizione. In passi sequenziali, possiamo scrivere:

Algorithm 2: Creazione del percorso tra lettore e prenotante

Data: Zona di residenza e raggio d'azione di utente lettore e prenotante

Result: Elenco di utenti attraverso cui il libro dovrà spostarsi *hand-to-hand*

Il raggio della circoscrizione di utenti coinvolti sarà pari alla distanza

$$\bar{Z} = \frac{1}{2}|z_i^p - z_0|$$

```

for Tutti gli utenti  $z_i^U$  nella community do
  if ( $\text{Distanza}(z_0, z_i^U) \leq \bar{Z}$  oppure  $(\text{Distanza}(z_i^P, z_i^U) \leq \bar{Z})$  then
    | Seleziono l'utente  $z_i^U$  e lo inserisco nella lista (HandToHandUsers) dei possibili utenti
    | che potrebbero partecipare attivamente al prestito;
  end
end

```

Creiamo il collegamento tra gli users della lista *HandToHandUsers* il cui raggio d'azione si sovrappone;

Quindi, alla fine dello step 1, avremo individuato tutti gli utenti i quali potrebbero partecipare attivamente alla realizzazione di un prestito: il risultato ottenuto sarà quindi come quello in figura 10.3.

//TODO: move this comment to documentation

// Insieme di candidati: users == overlapping users // Funzione obiettivo: minimizzare i km
che il libro deve fare / minimizzare il numero di utenti tra cui fare // il passamano -> la scelgo in
maniera epsilon - greedy

// Epsilon for epsilon greedy algorithm: during the path calculation for the reservation // we
choose usually the next user, as the nearest one. // Using the greedy algorithm, with a probability
that depends on the epsilon variable value, we choose the user // with the biggest radius. // epsilon
lower -> algorithm stronger (always foound path to me) // epsilon higher -> algorithm found sorhest
path

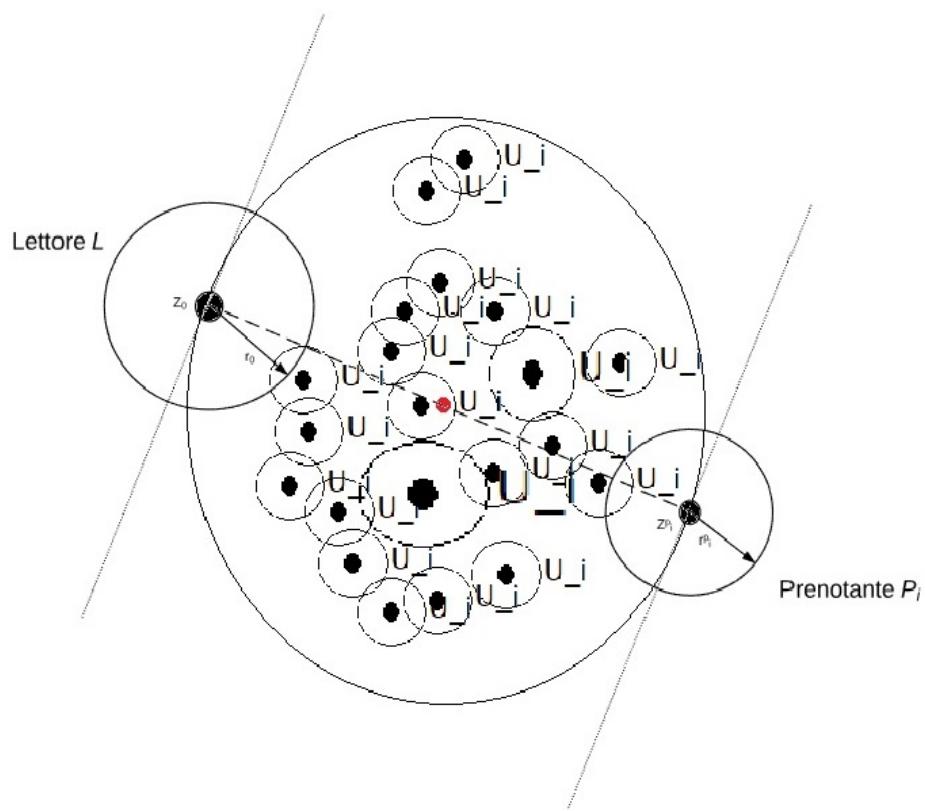


Figure 10.3: Rete di utenti che potrebbero essere attivi nel prestito

11

Test ed analisi dei componenti implementati

11.1 Lato Android

Per quanto riguarda la fase di testing di quanto implementato per il client Android sono stati sfruttati principalmente due tool:

- **Junit** per l'analisi dinamica del codice, con lo scopo, quindi di verificare che il codice scritto funzioni in maniera coerente a quanto richiesto e che vengano seguiti le richieste descritte nei casi d'uso.
- **Espresso Junit** per l'analisi dell'interfaccia grafica costruita per la gestione delle varie richieste. Anche in questo caso si è fatto riferimento a quanto descritto dai casi d'uso riguardo alle view e al loro formato per osservare la correttezza di quanto implementato.

11.1.1 Espresso Junit

Come detto, **Espresso Junit** è un tool attraverso il quale è possibile verificare le interazioni di un applicativo Android con l'utente. In questo caso, quindi, gli *assert* verificano, ad esempio, che il fragment aperto sia quello corretto e che i campi dei vari componenti inseriti abbiano il valore desiderato.

Nell'ambito di questa iterazione, le componenti che sono state testate con questo applicativo sono:

1. Login;
2. Registrazione manuale di un libro;
3. Ricerca di un libro.

Consideriamo, come esempio per la descrizione, il test relativo alla ricerca di un libro all'interno della rete di Book Crossing. Il test si compone nel seguente modo:

```
1 @Test
2 public void searchBookTestOK(){
3     Globals.isLoggedIn = true;
4     Globals.usernameLoggedIn = "A";
5     onView(withId(R.id.navigation)).check(matches(isDisplayed()));
6     onView(withId(R.id.book_search)).perform(click());
7     onView(withId(R.id.book_search)).check(matches(isDisplayed()));
8
9     ViewInteraction checkTitle = onView(withId(R.id.title_search));
10    ViewInteraction checkAuthor = onView(withId(R.id.author_search));
11    ViewInteraction checkBtnSearch = onView(withId(R.id.search_button));
12
13    checkTitle.check(matches(isDisplayed()));
```

```

14     checkAuthor.check(matches(isDisplayed()));
15     checkAuthor.check(matches(isDisplayed()));
16
17     checkTitle.perform(setTextInTextView(""));
18     checkAuthor.perform(setTextInTextView(""));
19
20     checkTitle.perform(setTextInTextView("questa storia"));
21     checkBtnSearch.perform(click());
22
23     onView(withId(R.id.books_found)).check(matches(isDisplayed()));
24     onData(anything()).inAdapterView(withId(R.id.books_found)).
25         atPosition(0).perform(click());
26     onView(withText(containsString("BOOKABLE"))).
27         inRoot(withDecorView(not(MainActivityActivityTestRule.getActivity().getWindow().
28             getDecorView()))).check(matches(isDisplayed()));

```

Questo caso di test verifica che la ricerca di un libro svolta in maniera corretta garantisce la visualizzazione dei corretti *fragment* e dei *Toast* formattati in maniera corretta. La medesima procedura è stata implementata per le altre componenti descritte precedentemente; il fatto di utilizzare questo tool consente di svolgere anche un'analisi dinamica del codice, dal momento che se ci fosse un errore, la successione grafica subirebbe un crash o mostrerebbe qualcosa di sbagliato facendo fallire il caso di test previsto.

11.2 Lato Server

11.2.1 Analisi Statica

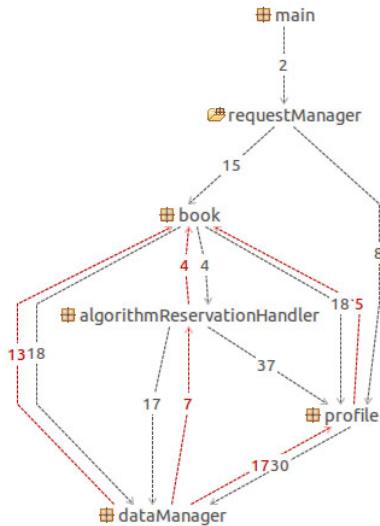


Figure 11.1: Analisi statica del codice

TODO: commentare immagine e controllare che corrisponda a quanto pianificato precedentemente
 Nel seguente piano sono rappresentate le metriche di:

- Abstractness: misura quanto facilmente il sistema può essere espanso;
- Instability: tentativo di misurare la facilità di cambiamento.

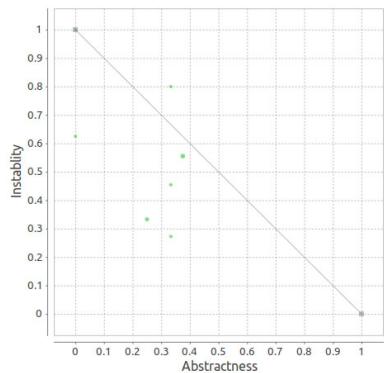


Figure 11.2: analisi statica grafico

i packages vicino al punto (0,0) sono rigidi mentre i packages vicino a (1,1) sono totalmente astratti e quindi inutili, idealmente vorremmo trovarci sulla linea rappresentata.