**TEMPLATE STRUTTURA RELAZIONE**

# Descrizione del problema

## Analisi e specifica dei requisiti

Il progetto consiste nel realizzare un sistema per la gestione dell’noleggio di bici pubbliche. Questo è composto da una serie di rastrelliere dove, attraverso un display possiamo eseguire il login, scegliere una delle tre tipologie di bici disponibili e ritirarla, potremmo poi restituire la bici semplicemente reinserendola in una qualsiasi rastrelliera nella morsa della tipologia corrispondente.

Sono disponibili quattro tipologie di abbonamento: giornaliero, settimanale, annuale e per studenti. I primi tre richiedono di inserire una carta di credito per i pagamenti che avvengono sia all’attivazione dell’account sia al termine di un noleggio. L’abbonamento studenti invece è gratuito ma richiede di verificare l’identità di studente durante la creazione specificando una mail universitaria.

È anche presente un tipo di account speciale per gli addetti ai lavori che permette di spostare e aggiungere bici alle rastrelliere e modificare le morse presenti.

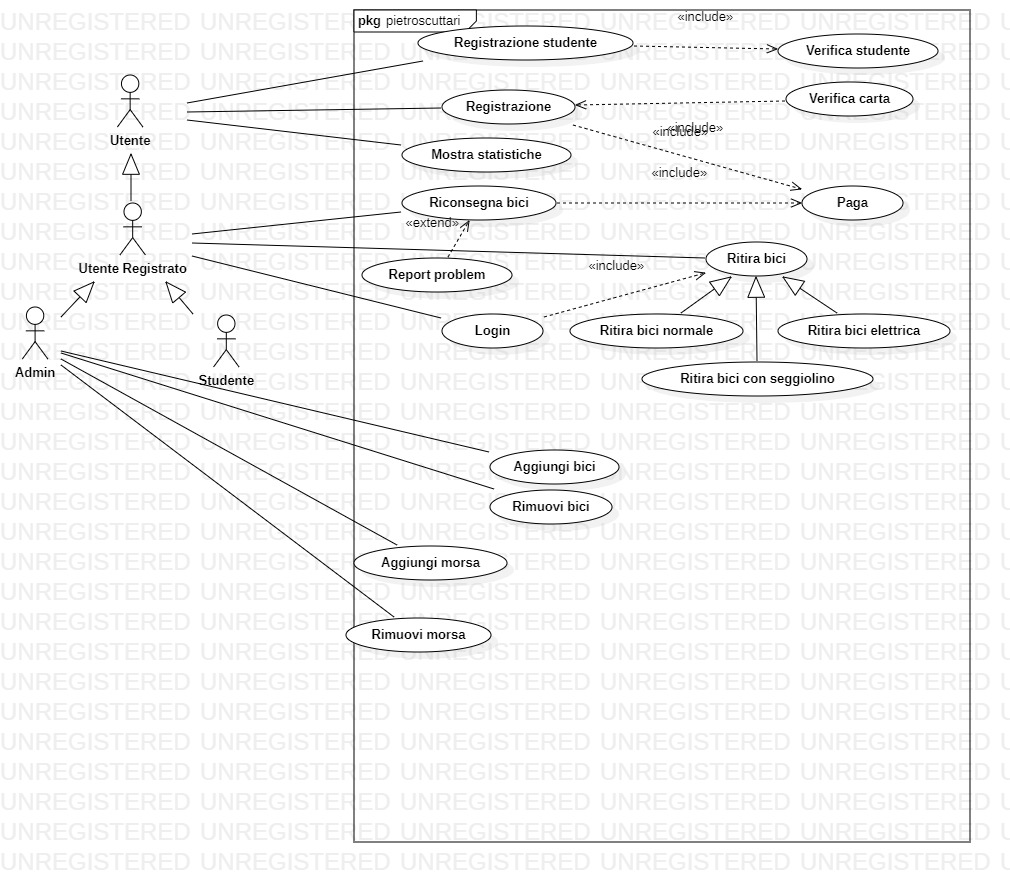
Sono inoltre disponibili delle statistiche sull’utilizzo del servizio come il numero medio di bici usate e la rastrelliera più usata.

In fine tutti i dati generati vanno salvati in un apposito database

# Progettazione del Sistema

Una copia ad alta risoluzione delle immagini è presente nella cartella images e gli originali si trovano nel file BikeSharing.mdj

## Diagramma dei casi d’uso



* **Registrazione studente**: Un utente può diventare un utente studente registrandosi e identificandosi come studente
* **Verifica studente**: Per l’iscrizione come studente va verificato lo status di studente
* **Registrazione**: Un utente può diventare un utente registrato
* **Verifica carta**: Durante la registrazione viene verificato che i dati della carta siano corretti
* **Mostra statistiche**: che sistema la capacità di mostrare le statistiche di utilizzo del sistema
* **Riconsegna bici**: Utente registrato dopo aver ritirato una bici la può riconsegnare
* **Paga**: Durante la registrazione dell'account e durante la riconsegna della bici l'utente paga la somma dovuta con la carta inserita nell'account
* **Report problem**: Quando un utente riconsegna una bici ha possibilità di segnalare un problema con essa
* **Ritira bici**: Un utente con un account attivo puoi ritirare una bici per usarla
  + Ritira bici normale: È possibile ritirare una bici normale
  + Ritira bici elettrica: È possibile ritirare una bici elettrica
  + Ritira bici con seggiolino: È possibile ritirare una bici elettrica con seggiolino
* **Aggiungi bici**: Un utente admin può aggiungere una bici a una rastrelliera
* **Rimuovi bici**: Un utente admin può rimuovere una bici da una rastrelliera
* **Aggiungi morsa**: Un utente admin può aggiungere una morsa a una rastrelliera
* **Rimuovi morsa**: Un utente admin può rimuovere una morsa da una rastrelliera

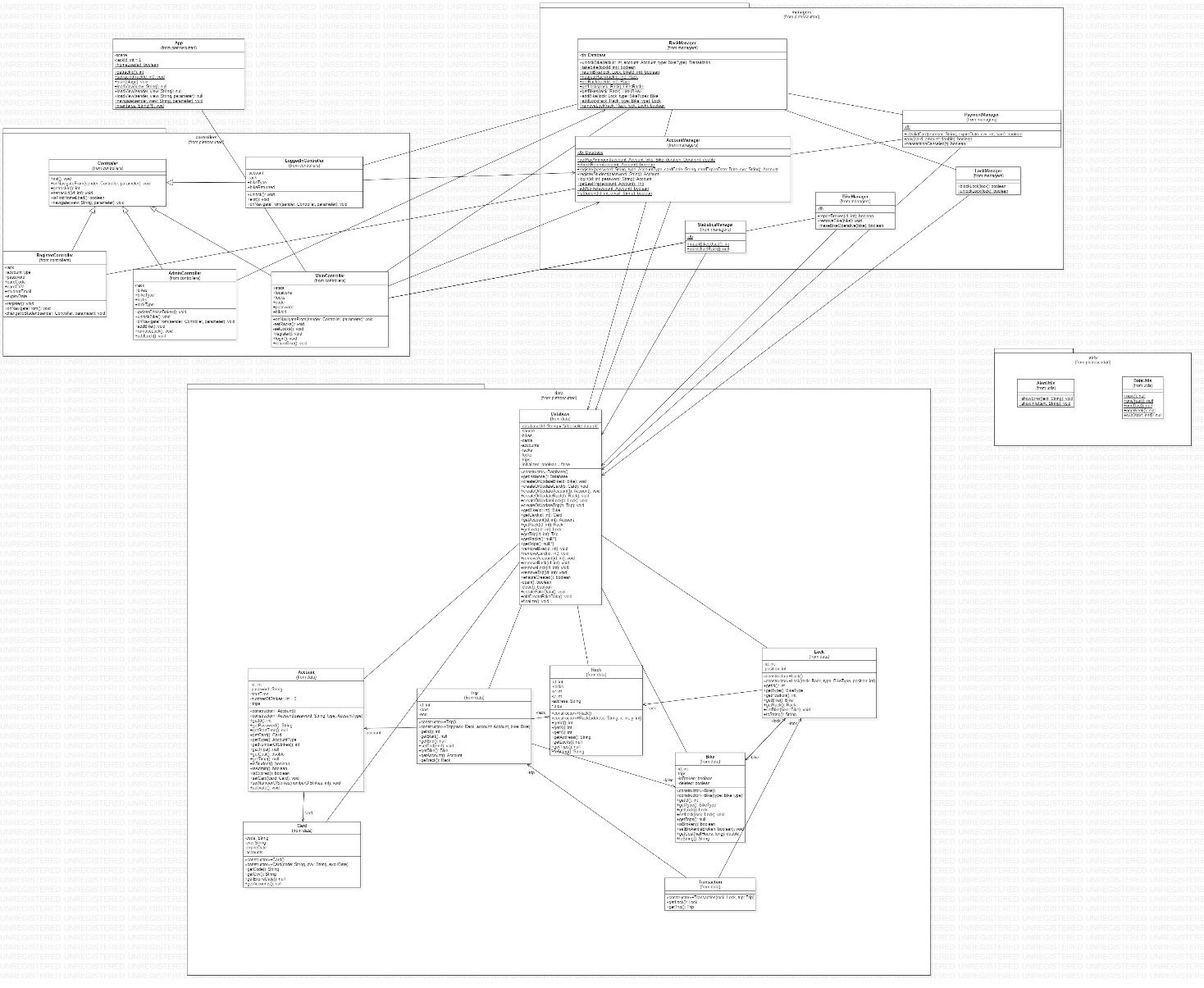
## Descrizione degli scenari

Tabella 1- Template per la descrizione di un caso d'uso.

|  |  |
| --- | --- |
| Nome | Aggiungi morsa |
| Scopo | Permettere agli admin di aggiungere delle morse alle rastrelliere |
| Attore/i | Admin |
| Pre-condizioni | L’admin ha fatto il login |
| Trigger | Premere sull’bottone “Add lock” |
| Descrizione sequenza eventi | L’admin dopo aver fatto il login seleziona il tipo di morsa da aggiungere e preme “Add lock” |
| Alternativa/e |  |
| Post-condizioni | La rastrelliera attuale ha una morsa in più vuota del tipo selezionato |

|  |  |
| --- | --- |
| Nome | Ritira bici |
| Scopo | Permettere agli utenti registrati di ritirare una bici |
| Attore/i | Utente registrato |
| Pre-condizioni | L’utente è registrato e conosce le sue credenziali d’accesso |
| Trigger |  |
| Descrizione sequenza eventi | * L’utente inserisce le sue credenziali e fa il login * Seleziona il tipo di bici desiderato e preme “Unlock” * Il sistema apre una morsa piena del tipo selezionato * All’utente è mostrato il numero della morsa aperta e può ritirare la bici |
| Alternativa/e | * Se le credenziali d’accesso sono errate l’utente riceve un messaggio d’errore e può riprovare * Se nella rastrelliera non sono presenti bici del tipo richiesto viene mostrato un errore e l’utente può selezionare una bici di tipo diverso |
| Post-condizioni | La morsa è aperta e il sistema ha registrato un nuovo noleggio sull’account dell’utente |

## Diagramma delle classi (modello di progetto)



Le classi sono divise in tre gruppi principali che ricalcano il design pattern MVC. Il gruppo data è quello che si occupa del salvataggio dei dati. La classe principale è quella del database che è un singleton che fornisce l’accesso al database, per fare questo utilizza le rimanenti classi che corrispondono agli oggetti che vogliamo salvare

La sezione dei managers invece si interfaccia con il database e provvede le funzioni richieste dai requisiti. Queste sono opportunamente divisi in diversi manager per aumentare la coesione delle classi. Un dettaglio importante è che i managers sono realizzate con classi interamente statiche ovvero in cui ogni loro metodo è statico questo ci permette di ridurre lo stato, che è invece salvato nel database, inoltre questo riduce il numero di oggetti presenti nella JVM aumentando l’efficienza della memoria e semplificando il lavoro del garbage collector.

L’ultimo gruppo è quello dei controllers, questi sono responsabili per l’interazione con l’UI. Sostanzialmente devono rispondere agli input dell’utente chiamando le funzioni necessarie dei managers per soddisfare le sue richieste.

## Diagrammi di sequenza

**Registrazione account:**

Il diagramma di sequenza mostra il processo di registrazione di un utente

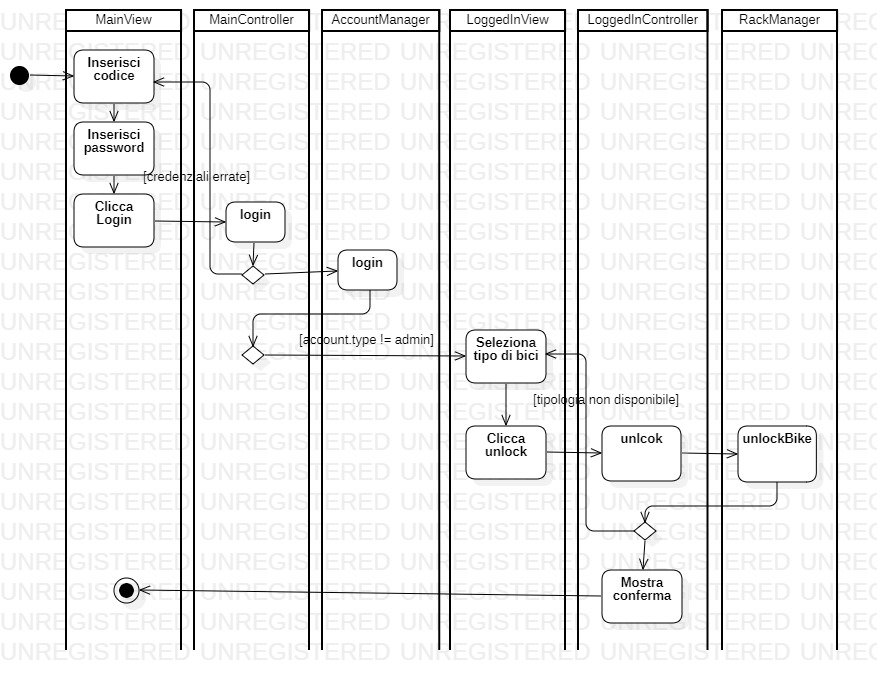
Immagine che contiene tavolo

Descrizione generata automaticamente

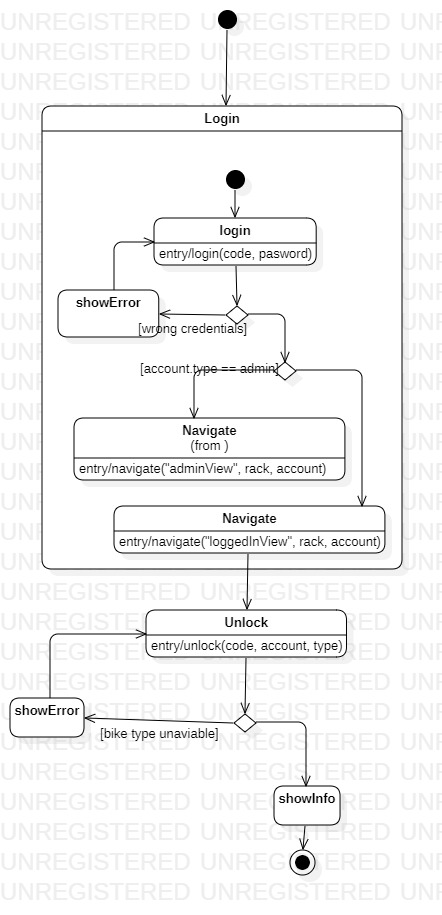
## Diagrammi delle attività

**Noleggio bici:**

Il diagramma delle attività mostra il processo di ritiro di una bici dall'inserimento delle informazioni di login allo sblocco della morsa



## Macchine di stato



## Diagramma dei componenti



# Implementazione del sistema

## Discussione dei Design Pattern utilizzati

* **Singleton**: la classe database che è responsabile per l’accesso al database SQLite è implementata attraverso un singleton cosa ci permette di non avere molteplici connessioni al database chi affidare a Ormlite la gestione della concorrenza.
* **MVC**: l'intero progetto è strutturato secondo il design pattern model view controller; infatti, le classi sono separate in classe di modelli che si interfacciano col database ed eseguono la computazione effettiva, classi controller che interpretano gli input dell'utente e lo convertono in istruzioni per i modelli e le view, realizzate in javaFX, realizzano interfaccia grafica effettiva.
* **DAO**: L’accesso al database è stato affidato a Ormlite, una libreria che si occupa di interfacciarsi con il database e fornire DAO.
* **Observer**: Nel RegisterController è usato un observer per rispondere ai cambiamenti della tipologia di account.
* **Indirection**: Per favorire il low coupling le classi adapter fanno da tramite tra il database e i controller, inoltre la separazione di queste funzioni in più classi separate aumenta la cohesion.

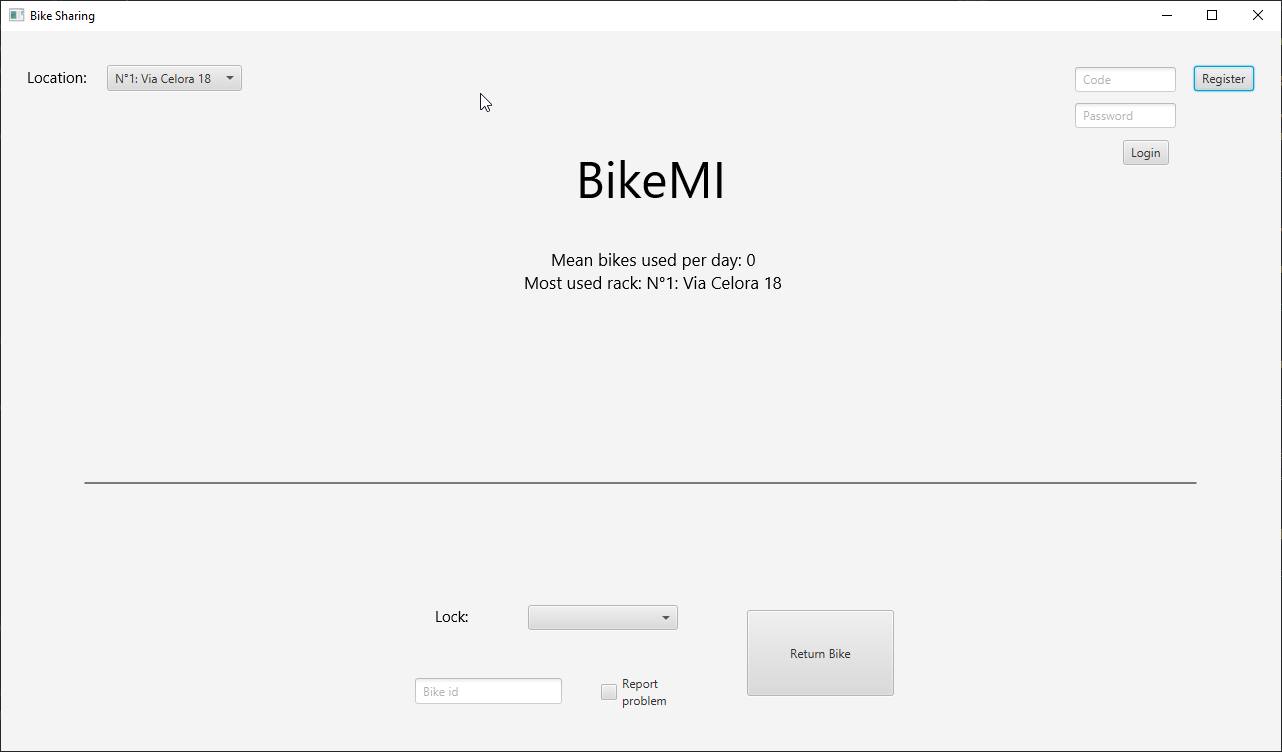
## Gestione dei dati persistenti

Il database usato è SQLite ma l’accesso è completamente mediato dalla libreria Ormlite. Sono state create una serie di classi corrispondenti alle tabelle del database, come Bike, Card, Account …. Attraverso delle annotazioni viene specificato quali attributi sono propri della tabella e quali sono ottenuti con relazioni ad altre tabelle. Nell’inizializzazione dell’database vengono create le tabelle necessarie per salvare le classi specificate e i corrispondenti DAO che verranno poi utilizzati per interfacciarsi con il database.

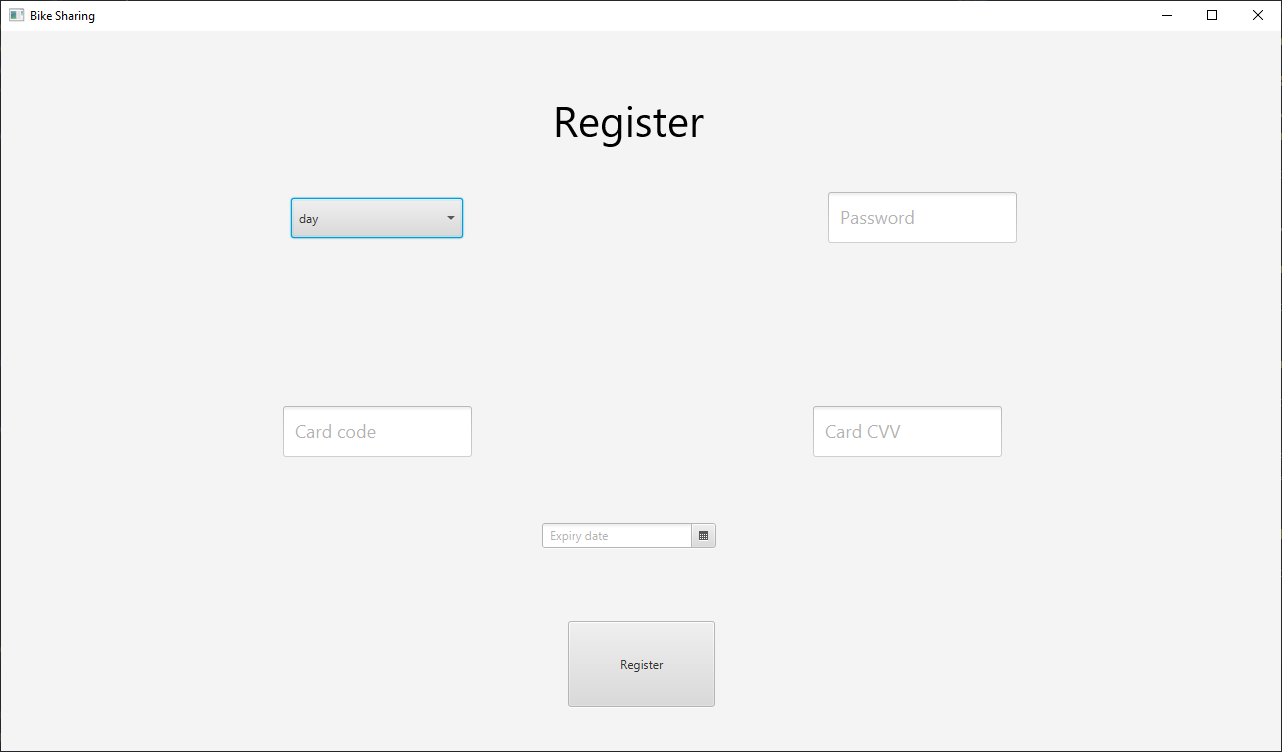
Il database risultante ha sei tabelle: Card, Account, Bike, Lock, Trip e Rack. E l’accesso è fatto dalla classe Database attraverso i DAO di Ormlite.

## Descrizione dell’Interfaccia Grafica

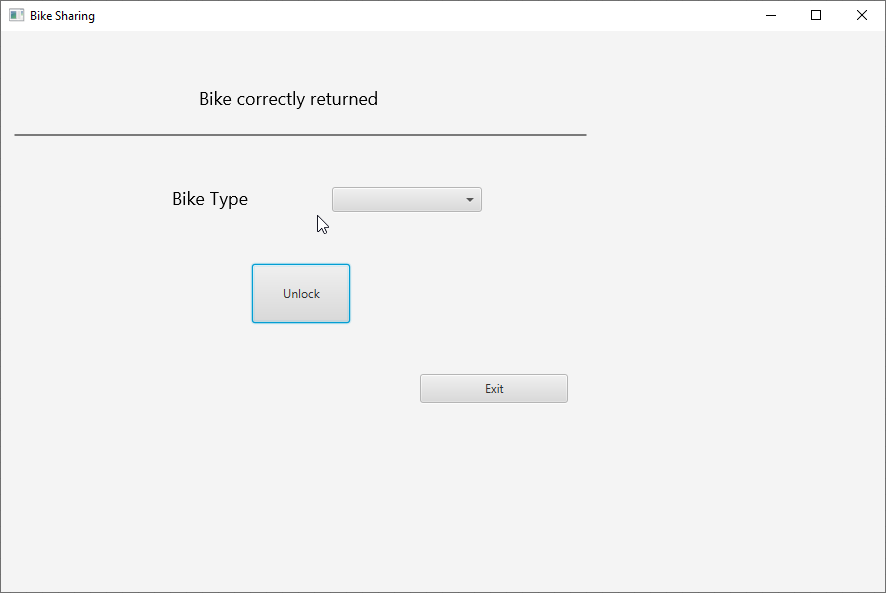
La schermata iniziale è MainView qui sono presenti le statistiche di utilizzo del servizio e viene data l'opportunità di registrarsi o effettuare il login. È inoltre presente una selezione della location e una sezione per restituire le bici: queste non sarebbero presenti nel sistema reale, infatti, la location sarebbe semplicemente il luogo fisico dove si trova la rastrelliera e la restituzione delle bici verrebbe fatta inserendo la bici fisicamente nella morsa.



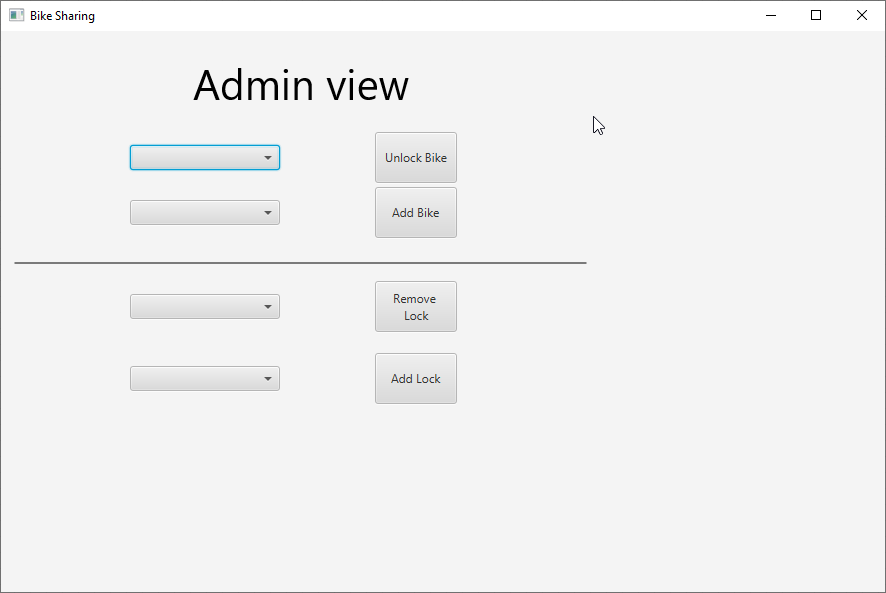
La schermata register viene aperta cliccando sul bottone register nella MainView e permette a un utente di registrarsi. Questo viene fatto selezionando il tipo di account desiderato e inserendo i dati necessari come la password e i dati della carta. Da qui non è possibile creare un account admin perché questa schermata è destinata agli utenti finali.



La LoggedInView è la pagina a cui accediamo dopo aver fatto il login, da qui possiamo ritirare una bici selezionando il tipo desiderato e premendo “unlock”, riceveremo un messaggio di conferma se il noleggio è andato a buon fine, altrimenti un errore se non sono disponibili bici del tipo selezionato. Se è stata consegnata una bici da questo account negli ultimi due minuti verrà mostrato un messaggio che conferma l’avvenuta restituzione.



L’AdminView è accessibile facendo il login con un account admin e mostra una serie di opzioni per gestire la rastrelliera.



## Diagramma di deployment



Ogni rastrelliera avrà il programma al suo interno e comunicherà con un server centrale in modo da mantenere i datai aggiornati e sincronizzati tra le diverse location.

## Specifica e verifica dei vincoli

La mappatura in JML è stata fatta come commento sopra i metodi e gli attributi corrispondent.

* Context: Account

inv: numberOfStrikes >= 0

* Context: Card

inv: code.length = 16

* Context: Lock

inv: position >= 0

* Context: Rack

inv: address != null

* Context: AccountManager.register(String password, AccountType type, String cardCode, Date cardExpireDate, String cvv)

pre: type != null and password != null and cardExpiryDate != null

* Context: AccountManager.registerStudent(String password)

pre: type != null and password != null and cardExpiryDate != null

* Context: PaymentManager.isValidCard(String number, Date expireDate, String cvv, AccountType type)

pre: exipiryDate != null and number.Length == 16

* Context: PaymentManager.pay(Card card, double amount)

pre: amount >= 0 and card != null

* Context: PaymentManager.transactionCanceled(Transaction t)

pre: t != null and t.getLock() != null and t.getTrip() != null#

* Context: RackManager.unlockBike(int rackId, Account account, BikeType type)

pre: account != null and !account.isExpired()

* Context: RackManager. getBikes(Rack rack)

pre: rack != null

## Descrizione del testing

I test sono implementati in JUnit ed eseguiti con l’estensione “Test runner for Java” per Visual Studio Code. I test verificano una serie di proprietà del codice puntando al branch coverage.

## Note per l’installazione e l’utilizzo

Per l’esecuzione è sufficiente installare l’estensione di Visual Studio Code “Extension pack for java” (<https://marketplace.visualstudio.com/items?itemName=vscjava.vscode-java-pack>) aprire la cartella principale del progetto e eseguirlo premendo f5.

Nel database sono presenti tre account di prova:

* Id: 1 password: “admin” per l’account admin
* Id: 2 password: “normal” per un account di una settimana
* Id: 3 password: “student” per un account student
* Id: 4 password: “expired” per un account con 3 strikes