Università degli studi di Salerno

Progetto di Fondamenti di Visione Artificiale

Gruppo 2

Modulo di estrazione delle caratteristiche dell'orecchio

Autori:
Matteo Merola
Simone Scalabrino
Carlo Branca

Supervisore:
Prof. Michele Nappi
Dott.ssa Chiara Galdi

Documentazione
Versione 1.0

28 maggio 2014

Indice

1. Scopo

Il modulo ha lo scopo di estrarre delle features da immagini di orecchio segmentate attraverso l'applicazione di due algoritmi:

- SIFT (Scale Invariant Feature Transform)
- LBP (Local Binary Pattern)

Il sistema software realizzato è facilmente estensibile. Nuovi algoritmi di estrazione possono essere aggiunti con il minimo sforzo possibile in termini di sviluppo. Il sistema è progettato per essere manutenibile e scalabile grazie all'utilizzo di design pattern.

2. Specifica

L'applicazione realizzata rispetta la seguente specifica:

Input:

- N (definito nel file di configurazione, es N=4) immagini di orecchio segmentate;
- Username;
- Punteggio qualità;
- Azione: registrazione/riconoscimento/verifica;

Funzionalità principali:

- Estrazione delle caratteristiche con uno o più algoritmi (definito in un file di configurazione);
- Salvataggio degli N vettori delle caratteristiche, con associato lo username, in caso di registrazione;

Output:

- Uno o più vettori delle caratteristiche (a seconda del numero di algoritmi di feature extraction usati);
- Username;
- Punteggio qualità;
- Azione: registrazione/riconoscimento/verifica;

3. Parametri

Il modulo restituisce un oggetto di tipo Map<String, List<List<IFeature>>> (ovvero una mappa del tipo "Nome algoritmo -> Lista"). In particolare, la lista contiene n elementi, dove n è il numero di immagini passate in input e, per ogni immagine i, è presente un'ulteriore lista di feature relative all'immagine i-esima.
Ad esempio, se sono passate 2 immagini e sono attivi gli algoritmi SIFT e LBP, il valore di ritorno sarà formato così:

```
Map(
"SIFT",  List(List<IFeature>, List<IFeature>)
"LBP",   List(List<IFeature>, List<IFeature>)
)
```

Se sono passate 3 immagini con il solo algoritmo SIFT attivo:

```
Map(
"SIFT",  List(List<IFeature>, List<IFeature>, List<IFeature>)
)
```

Ad ogni immagine è associata, dunque, una lista di IFeature (per ogni algoritmo). IFeature è un'interfaccia che descrive una feature generica; l'implementazione cambia in base all'algoritmo. In allegato invio i file sorgenti di IFeature, LBPFeature, SIFTFeature (le ultime due sono implementazioni concrete della prima) e Feature (la classe della libreria per il SIFT che noi incapsuliamo in SIFTFeature).
L'algoritmo LBP restituisce, per ogni immagine, una singola IFeature (di tipo LBPFeature) che ha al suo interno, come descrittori (variabile descriptors), un array di interi.
L'algoritmo SIFT restituisce, per ogni immagine, un numero variabile di

IFeature (di tipo SIFTFeature) dipendente dall'immagine.
La classe IFeature ha un metodo (getDistance(IFeature)) adibito al calcolo della distanza tra due IFeature: il metodo è già implementato per l'algoritmo SIFT (è definito nella libreria che abbiamo utilizzato) ma non per LBP.
Le feature sono serializzate nel database in fase di registrazione.

4. Diagrammi

4.1. Strategy Pattern

Nella programmazione ad oggetti, lo strategy pattern è uno dei pattern fondamentali, definiti originariamente dalla gang of four.

Lo strategy pattern è uno dei pattern comportamentali. L'obiettivo di questa architettura è isolare un algoritmo all'interno di un oggetto. Il pattern strategy è utile in quelle situazioni dove sia necessario modificare dinamicamente gli algoritmi utilizzati da un'applicazione. Si pensi ad esempio alle possibili visite in una struttura ad albero (visita anticipata, simmetrica, posticipata): mediante il pattern strategy è possibile selezionare a tempo di esecuzione una tra le visite ed eseguirla sull'albero per ottenere il risultato voluto. Il design pattern Iterator si basa proprio su questo.

Questo pattern prevede che gli algoritmi siano intercambiabili tra loro (in base ad una qualche condizione) in modo trasparente al client che ne fa uso. In altre parole: la famiglia di algoritmi che implementa una funzionalità (ad esempio di visita o di ordinamento) esporta sempre la medesima interfaccia, in questo modo il client dell'algoritmo non deve fare nessuna assunzione su quale sia la strategia istanziata in un particolare istante.

Nell'applicazione realizzata abbiamo utilizzato questo design pattern per fornire il supporto all'estensione del sistema per l'aggiunta di nuovi algoritmi per l'estrazione delle caratteristiche.

4.2. Bridge Pattern

Il bridge pattern è un design pattern (modello di progettazione) della programmazione ad oggetti e permette di separare l'interfaccia di una classe (che cosa si può fare con la classe) dalla sua implementazione (come si fa). In tal modo si può usare l'ereditarietà per fare evolvere l'interfaccia o l'implementazione in modo separato.

Abbiamo utilizzato questo design pattern per supportare l'aggiunta di diverse implementazioni del medesimo algoritmo di estrazione. Un esempio di utilizzo di questo pattern può essere individuato nell'implementazione di due versioni dell'algoritmo LBP: una scritta in linguaggio nativo C e l'altra in linguaggio Java.

4.3. Observer Pattern

L'Observer pattern è un design pattern utilizzato per tenere sotto controllo lo stato di diversi oggetti.

È un pattern intuitivamente utilizzato come base architetturale di molti sistemi di gestione di eventi. Molti paradigmi di programmazione legati agli eventi, utilizzati anche quando ancora non era diffusa la programmazione ad oggetti, sono riconducibili a questo pattern. È possibile individuarlo in maniera rudimentale nella programmazione di sistema Windows, o in altri framework di sviluppo che richiedono la gestione di eventi provenienti da diversi oggetti, come ad esempio la funzione OnMsgProc per la gestione delle code di messaggi windows.

Sostanzialmente il pattern si basa su uno o più oggetti, chiamati osservatori o listener, che vengono registrati per gestire un evento che potrebbe essere generato dall'oggetto osservato.

Oltre all'observer esiste il concrete Observer che si differenzia dal primo perché implementa direttamente le azioni da compiere in risposta ad un messaggio; riepilogando il primo è una classe astratta, il secondo no.

Uno degli aspetti fondamentali è che tutto il funzionamento dell'observer si basa su meccanismi di callback, implementabili in diversi modi, o tramite funzioni virtuali o tramite puntatori a funzioni passati quali argomenti nel momento della registrazione dell'observer, e spesso a questa funzione vengono passati dei parametri in fase di generazione dell'evento.

Abbiamo utilizza questo design pattern per fornire un callback alla attività principale dell'applicazione Android al momento in cui il Thread asincrono che effettua l'estrazione tramite gli algoritmi scelti termina la sua esecuzione correttamente o termina a seguito di un'eccezione generata.

4.4. Pacchetti

L'applicazione si dipana nei pacchetti rappresentati in figura 4.4.

4.5. Database

L'accesso al database embedded nell'applicazione Android avviene tramite le classi diagrammate nella figura 4.5.

Figura 4.1: Strategy

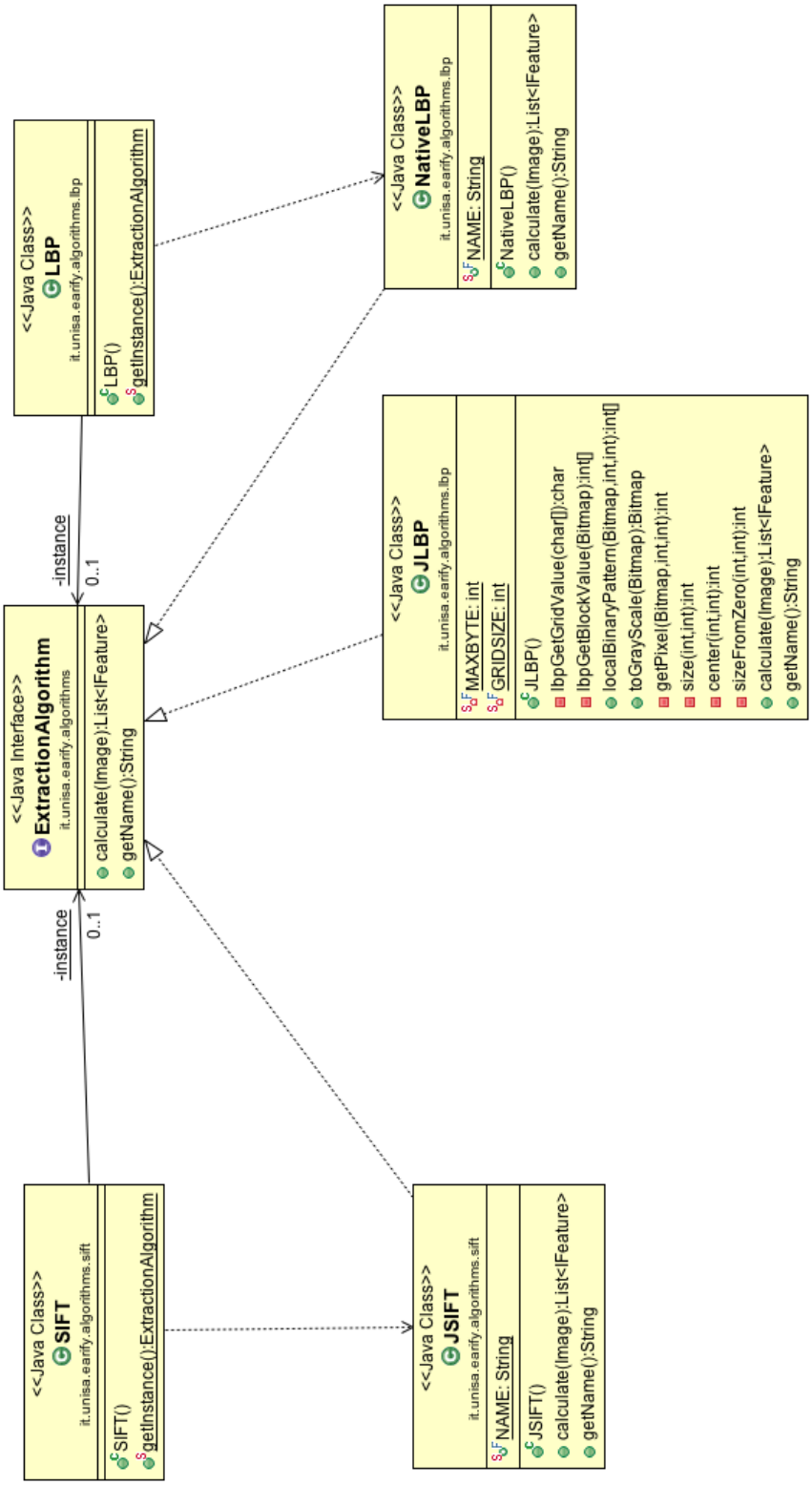

Figura 4.2: Bridge

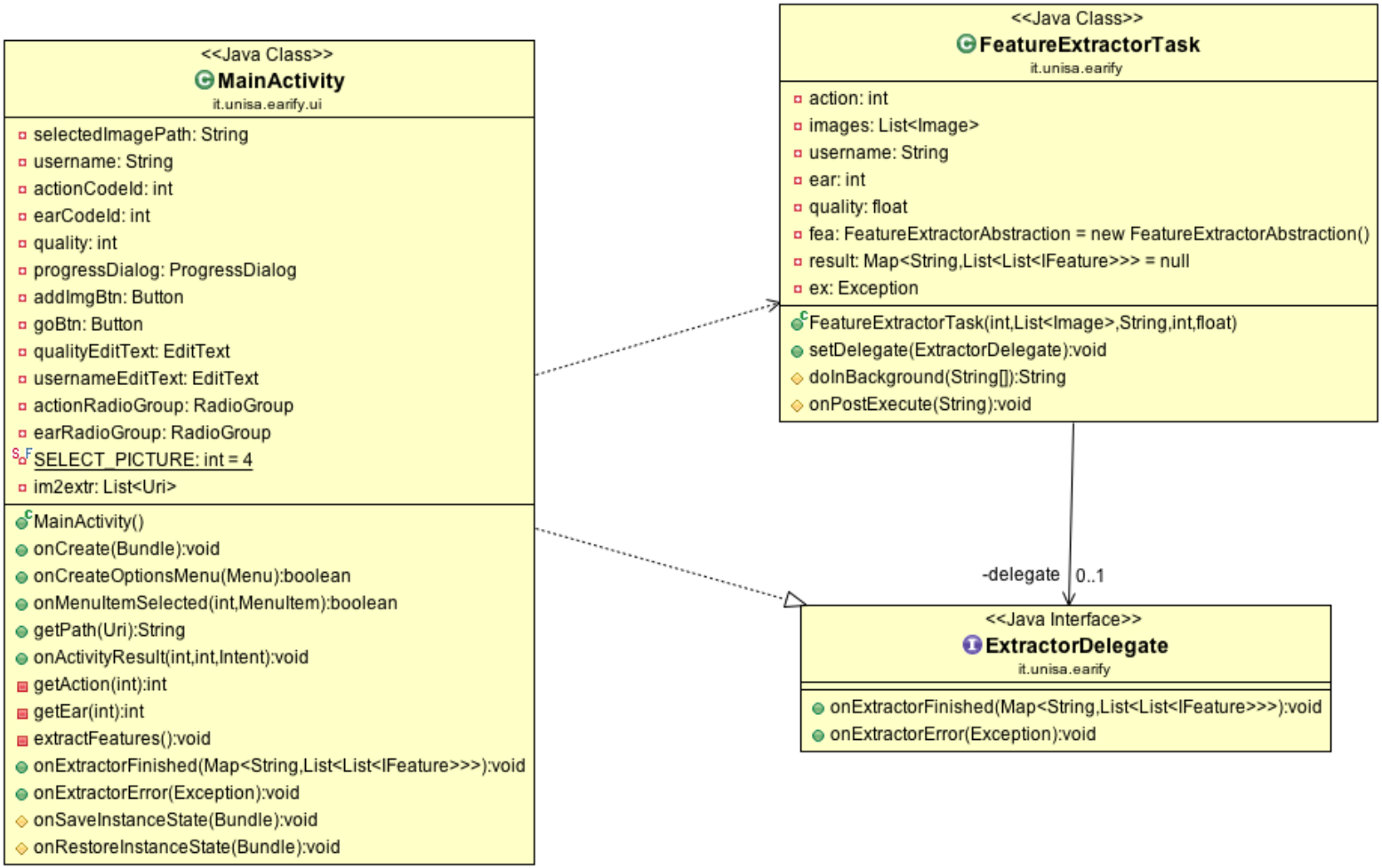

Figura 4.3: Observer
11

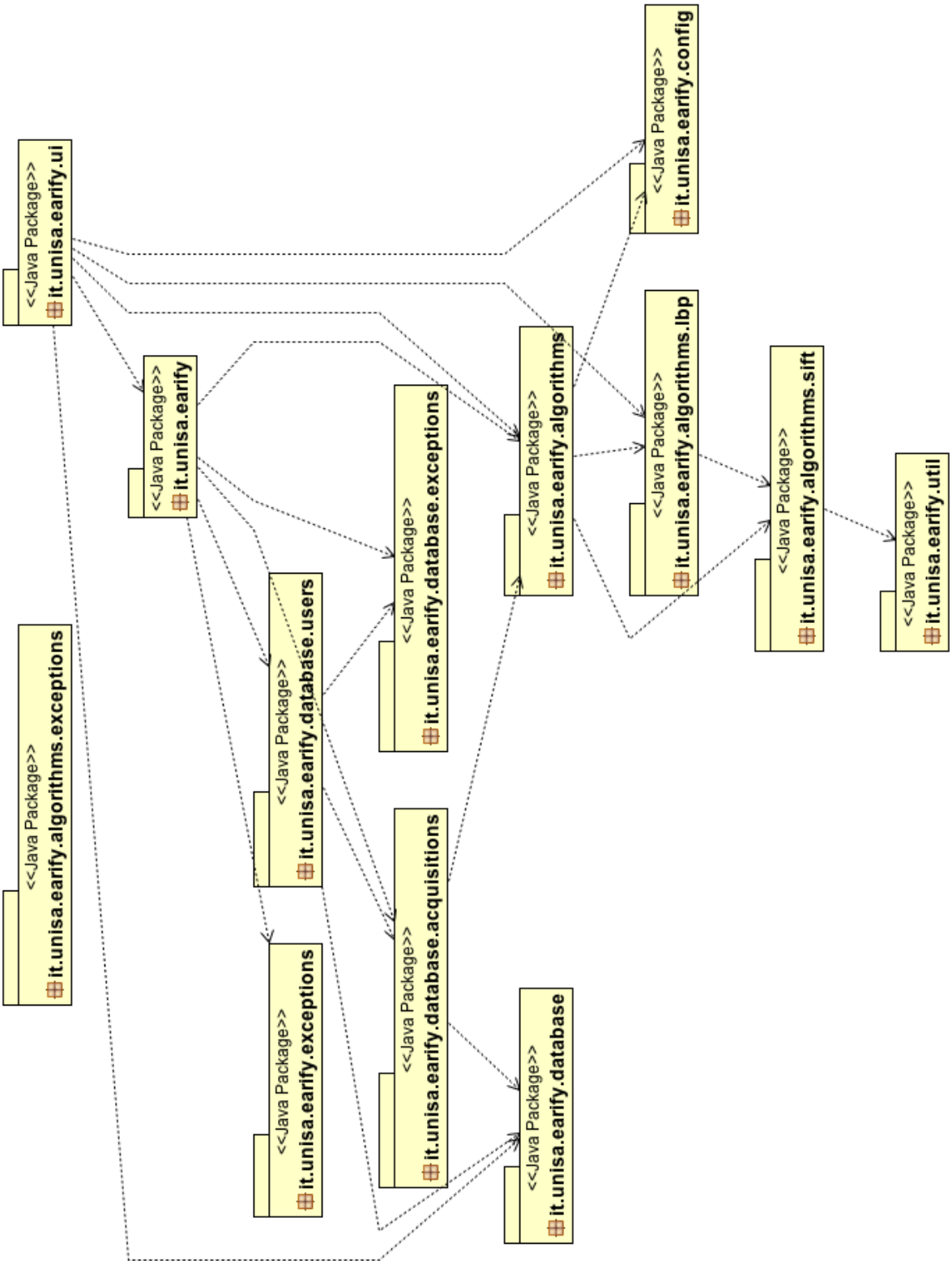

Figura 4.4: Packages

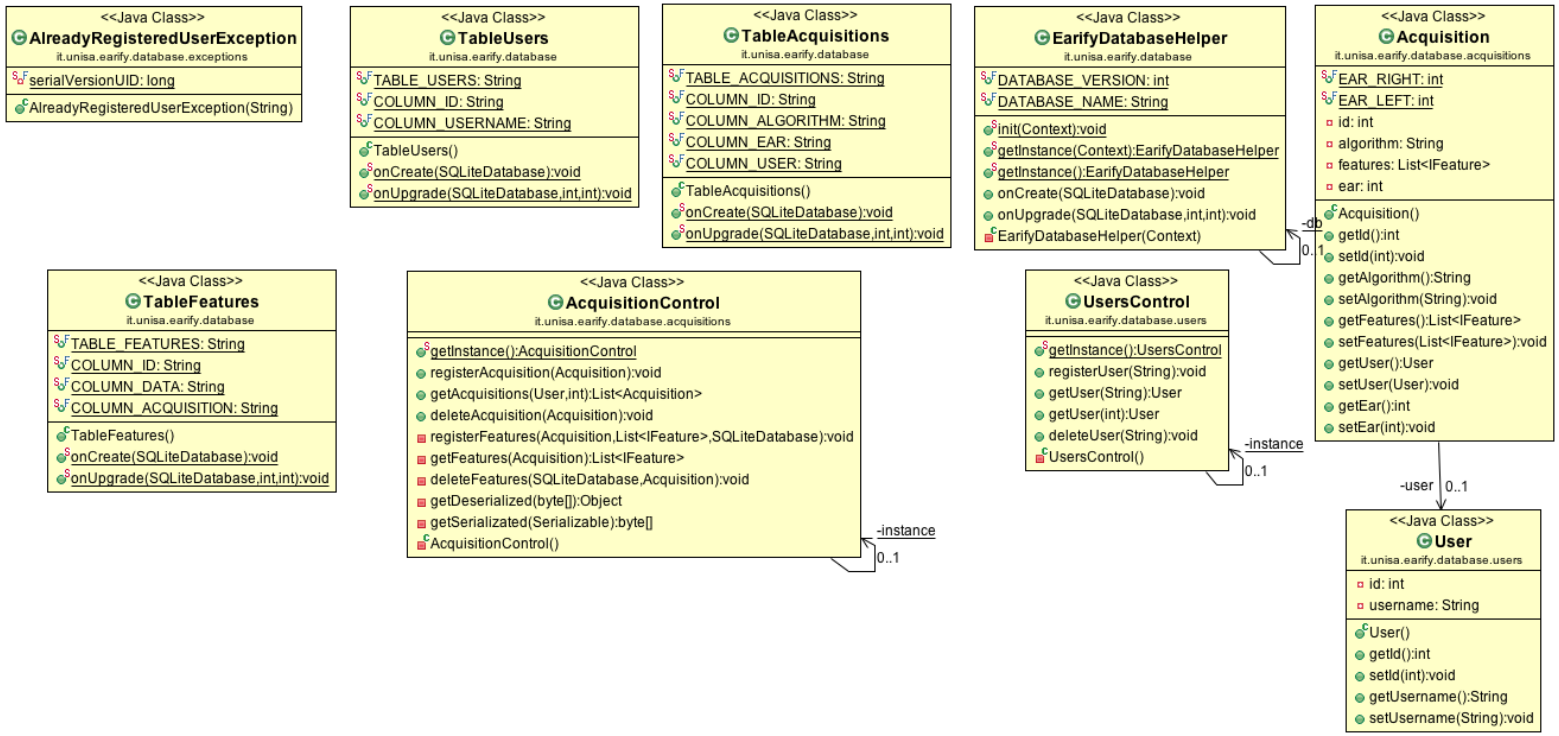

Figura 4.5: Classi database