RUBIK'S CUBE SOLVER

INDICE

1) Specifica dei requisiti
1.1) Requisiti funzionali 3
1.2) Fattori di successo 5
1.3) Requisiti non funzionali 5

2) Funzionamento generale
2.1) Use case diagram 6
2.2) Package diagram 7
2.3) Activity e State diagram 8

3) Design pattern implementati
3.1) Composite 10
3.2) Iterator 11
3.3) Command 12
3.4) Singleton 13
3.5) Adapter 13
3.6) Class diagram 14

4) Design principles
4.1) Principi SOLID 15
4.2) Coesione e Accoppiamento dei moduli 15
4.3) CLEAN Architecture 16

Conclusioni e Note *..... 17*

PREMESSA

Il software descritto è stato sviluppato per l'esame di Programmazione ad Oggetti, assieme ad un altro collega (l'applicazione è scaricabile su GitHub seguendo il link nelle note [1]).
Siccome ho personalmente sviluppato i moduli relativi allo scanner e la gestione del cubo, ho scelto di trattare approfonditamente quella parte.

Dichiaro che questo elaborato è frutto del mio personale lavoro, svolto sostanzialmente in maniera individuale e autonoma.

1. SPECIFICA DEI REQUISITI

1.1 Requisiti funzionali

Lo scopo dell'app è quello di scannerizzare, attraverso la fotocamera del cellulare, i colori di un cubo di Rubik irrisolto, calcolare la soluzione, ed infine mostrare in modo intuitivo le mosse da eseguire per risolverlo.

L'applicazione fornisce un semplice menù iniziale (fig. 1.1), dove l'utente potrà decidere se leggere le istruzioni, che spiegano in dettaglio come utilizzarla, oppure procedere con la scansione del cubo.

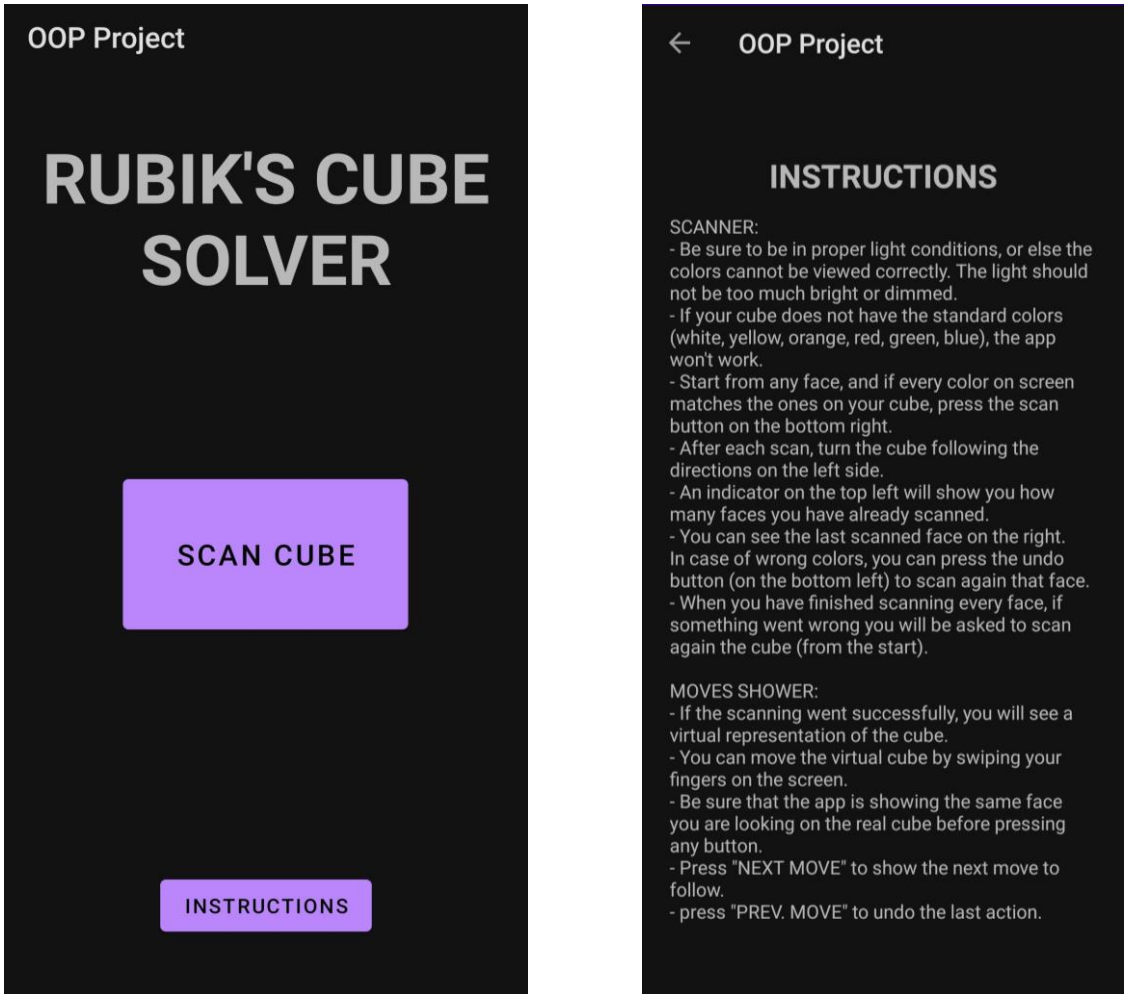


figura 1.1: *menu principale (a sinistra) e istruzioni (a destra)*

Tramite la fotocamera del cellulare (previa richiesta di autorizzazione), l'utente potrà scannerizzare le 6 facce di un cubo standard, con colori rosso, blu, verde, giallo, arancione (è possibile iniziare da una faccia qualsiasi). Verrà chiesto all'utente la dimensione del cubo, che può essere minimo 2*2*2 (ossia 4 quadrati per faccia). Non sono supportate forme diverse dal cubo, come ad esempio le piramidi che hanno 5 facce (fig. 1.2).

figura 1.2: *diversi tipi di cubi di Rubik.*
La piramide non è supportata.

In fig. 1.3 è riportata la schermata di scansione: l'indicatore in alto a sinistra mostra il numero della faccia corrente da scannerizzare, appena sotto c'è una scritta per indicare verso che lato ruotare il cubo, e in basso un bottone "undo" da premere in caso siano stati registrati dei colori errati.
In alto a destra c'è la preview dei colori appena scannerizzati, e sotto di esso il pulsante per registrare i colori della faccia corrente. Infine, in basso al centro, c'è il bottone per rileggere le istruzioni viste in fig. 1.1.

figura 1.3: *schermata di scansione del cubo*

Ogni volta che si preme il tasto di scansione, vengono registrati i colori della faccia che si sta inquadrando, vengono aggiornati in alto a destra i colori appena registrati, viene fornita la direzione in cui bisogna ruotare il cubo per la prossima faccia, e il contatore in alto a sinistra indicherà il numero della prossima faccia da registrare (ossia il numero di quelle già registrate più uno).

Alla fine del processo, dopo la sesta e ultima scansione, verrà effettuato un breve controllo, per verificare se l'utente ha registrato correttamente il cubo (viene per esempio controllato che non ci siano più colori di quelli che ci dovrebbero essere, se la configurazione è risolvibile, se è stato possibile riconoscere tutti i colori, ecc…).
In caso negativo, verrà chiesto di ripetere l'operazione da zero, altrimenti una libreria esterna apposita si occuperà di risolvere il cubo tramite algoritmi di ottimizzazione.

Appena trovata la soluzione, si aprirà una schermata dove verrà mostrato il cubo in 3D (fig. 1.4), che l'utente potrà ruotare a suo piacimento. Esso avrà la configurazione appena inserita dall'utente.
Ci sono due pulsanti: uno per mostrare la mossa successiva, l'altro per tornare a quella precedente. Ogni volta che si premerà un tasto, verrà riprodotta l'animazione che corrisponde alla mossa da eseguire, fino alla completa risoluzione.

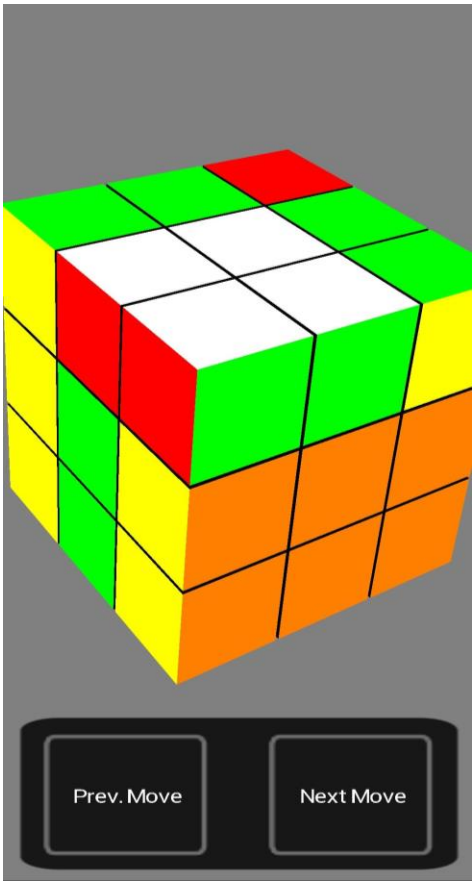


figura 1.4: *visualizzazione mosse*

1.2 Fattori di successo

Lo scopo dell'app è quello di fornire all'utente la soluzione al suo cubo di Rubik fisico, reale.
Quindi, il fattore di successo non è tanto la visualizzazione o la soluzione, ma la virtualizzazione facile e veloce del cubo, ossia la scansione. Un utente dovrebbe scegliere quest'app rispetto ad altre, perché garantisce una velocità di inserimento delle facce molto più veloce, rispetto a inserire manualmente i colori di ogni quadratino.

1.3 Requisiti non funzionali

Il software deve essere abbastanza performante, da permettere una velocità di almeno circa 30 fps della telecamera (altrimenti il feedback mostrato a schermo risulterebbe lento e a scatti).
L'applicazione è realizzata in Android Studio, in linguaggio Java. La scannerizzazione avviene tramite la libreria OpenCv [2], sempre nella sua versione Java.
La telecamera non deve perciò essere quella delle API di Android, ma specifica di OpenCv, ossia è necessario implementare l'interfaccia CvCameraViewListener. Questo mi permette di poter scrivere del testo e disegnare forme sulle immagini, e di effettuare facilmente controlli sui pixel di ogni frame.
Per la soluzione del cubo viene usata la libreria Kociemba [3], scritta in Java, come per la libreria della visualizzazione 3d del cubo (OpenGl [4]).

2. FUNZIONAMENTO GENERALE

Il software è diviso in tre moduli funzionali principali (potenzialmente indipendenti, potrebbero funzionare anche se presi singolarmente):

- **Scansione** della configurazione del cubo, faccia per faccia.
- **Soluzione** del cubo, rappresentata da una sequenza di mosse.
- **Visualizzazione** della suddetta sequenza, tramite simulazione 3D del cubo.

2.1 Use case diagram

L'unico attore che utilizza l'applicazione è l'utente.
La libreria a destra in fig. 2.1 fornisce un servizio, ossia calcola la soluzione del cubo.

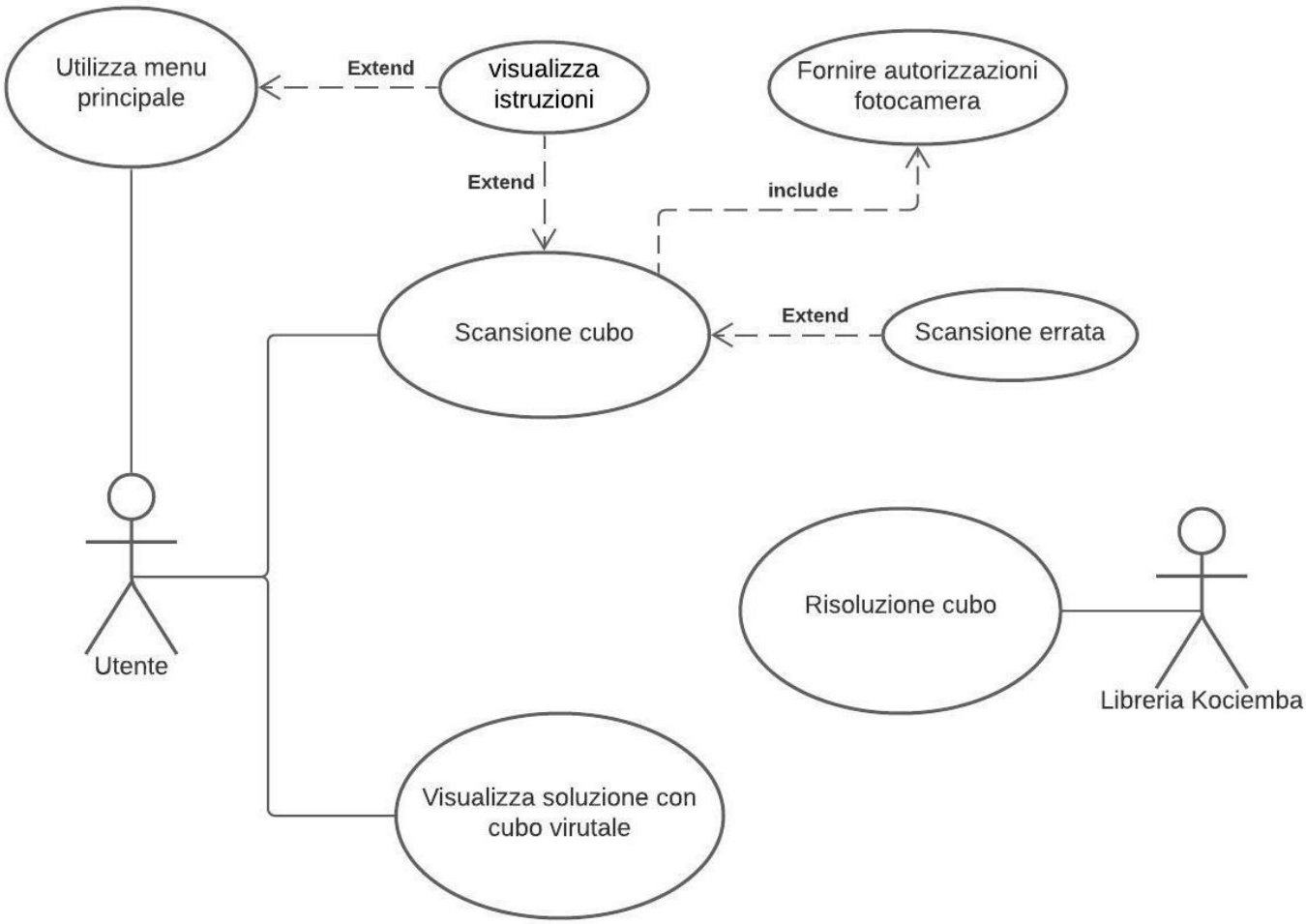


figura 2.1: *use case diagram*

2.2 Package diagram

In fig. 2.2 è riportato il package diagram dei moduli principali del software. Non ci sono tutti, in quanto mancherebbero dei package per classi specifiche di Android (ad esempio gli Intent per passare da un'Activity a un'altra), ma non li ho considerati essenziali per comprendere la struttura del software.

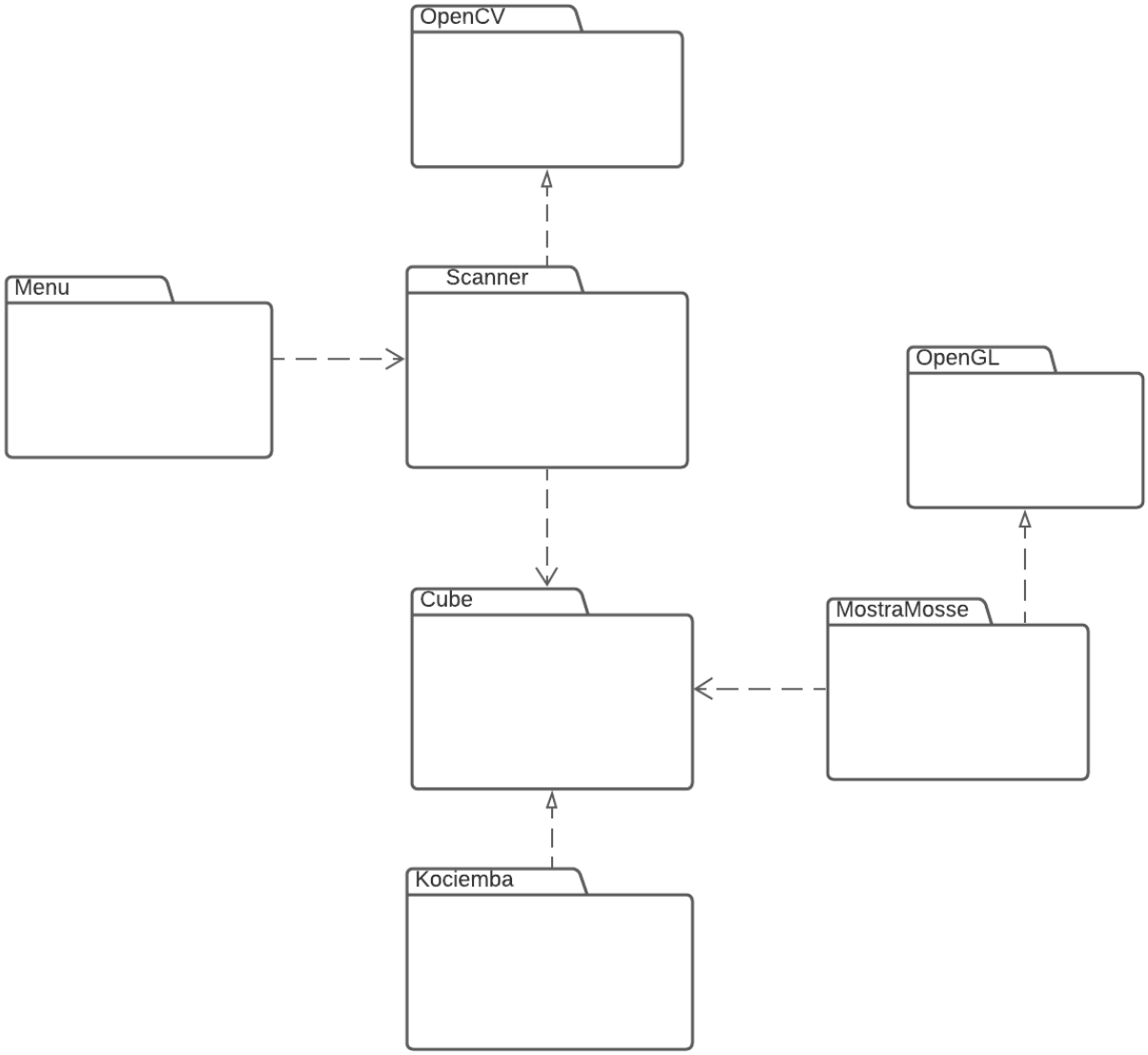


figura 2.2: *package diagram*

2.3 Activity e State diagram

Nello state diagram (fig.2.3), sono riportati gli stati in cui si trova il cubo nel corso dell'esecuzione del software.
Ho deciso di mostrare gli stati per ogni faccia scannerizzata, e non farlo ciclicamente, perché è importante notare che ad ogni scansione deve avvenire la rotazione della matrice dei quadrati, e avviene ogni volta in un senso differente.
Questo perché a ogni registrazione di una faccia, si deve ruotare il cubo in un particolare senso (indicato dal software), per passare alla successiva, pertanto cambia l'orientamento della faccia. Dato che mi serve mantenere lo stesso orientamento (per semplicità), questo passaggio è fondamentale.

Si può notare più esplicitamente nell'activity diagram (fig. 2.4).
Quest'ultimo mostra il processo di scansione di tutte le facce, fino alla sua risoluzione.

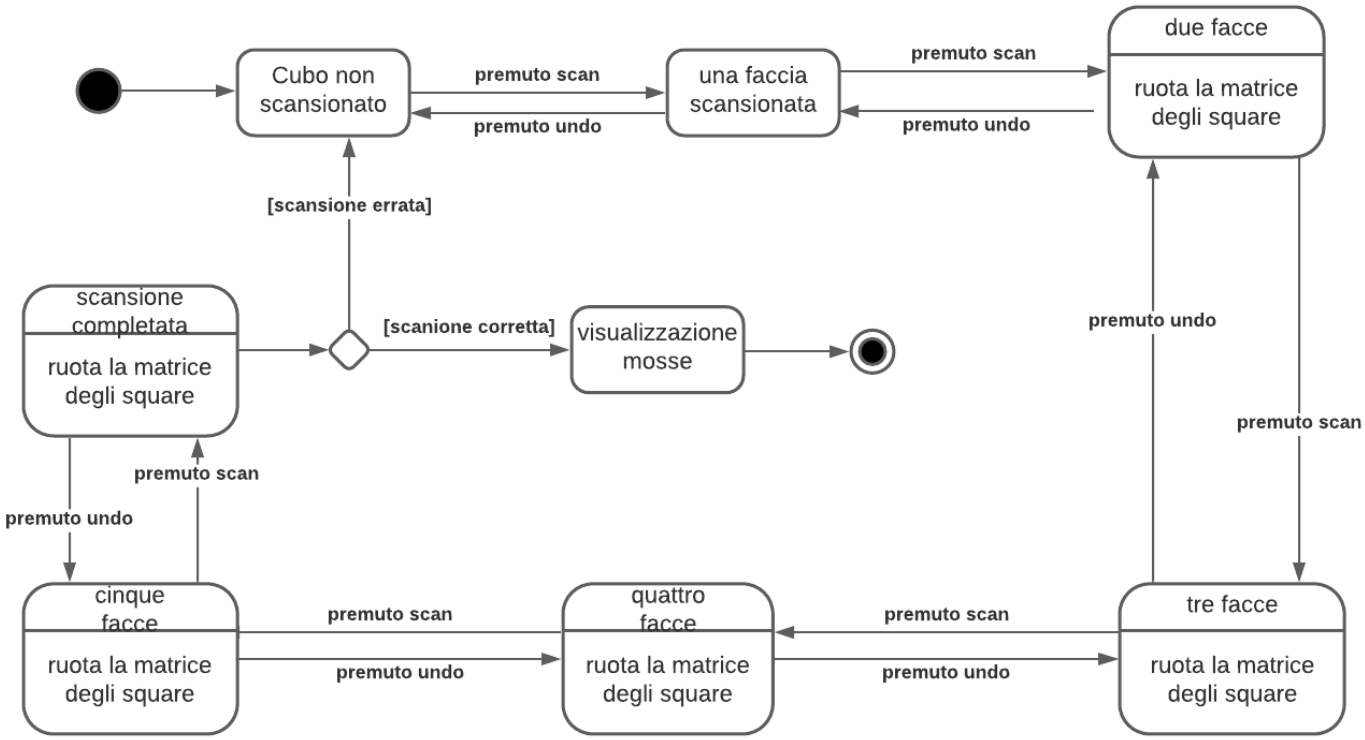


figura 2.3: *state diagram*

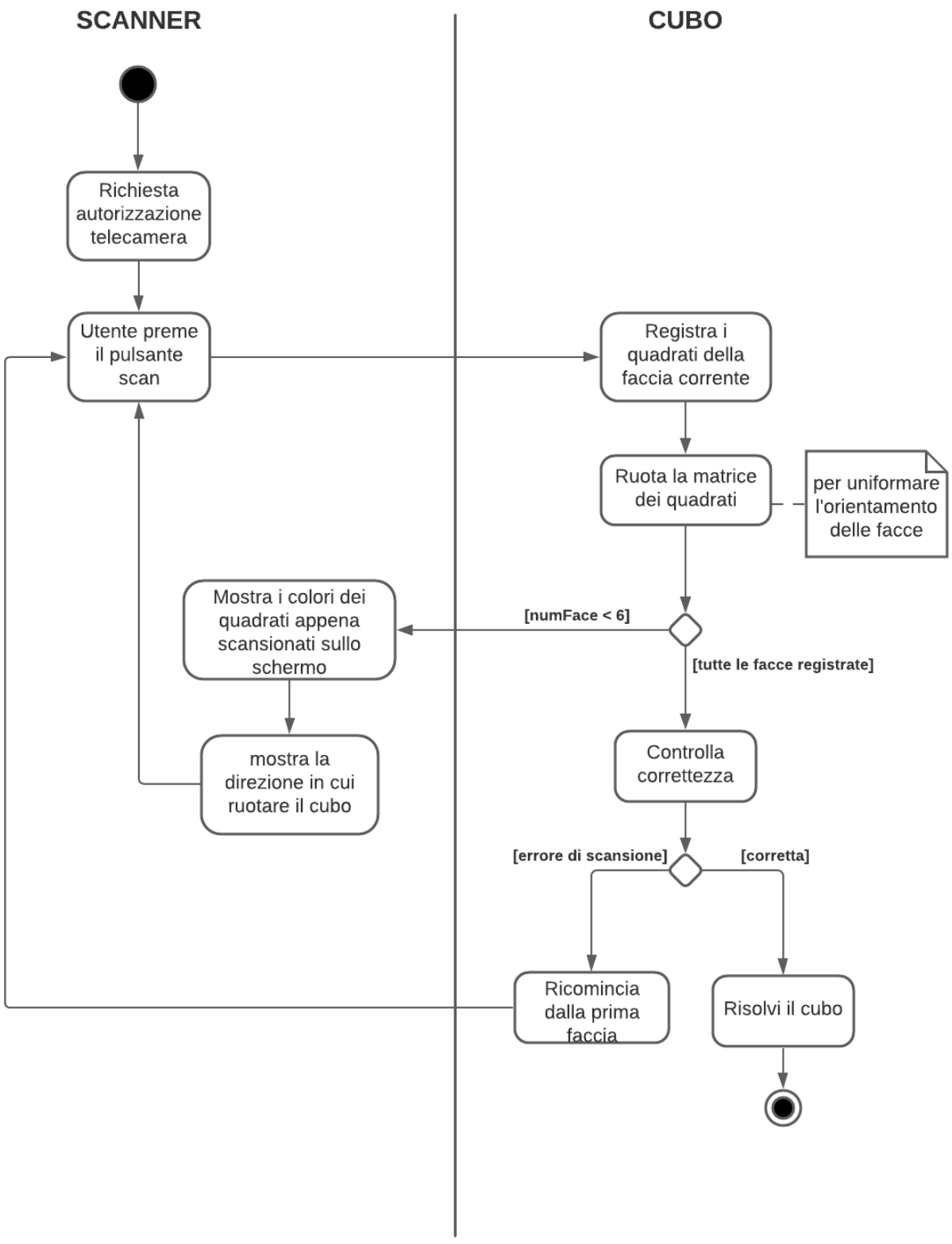
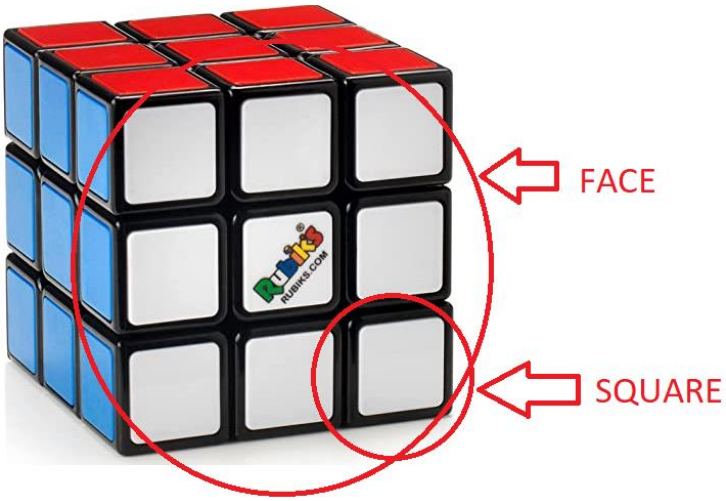


figura 2.4: *activity diagram*

3.DESIGN PATTERN IMPLEMENTATI

3.1 Composite



Un cubo è composto da facce, e ogni faccia ha dei quadrati (fig. 3.1). Questa struttura ad albero potrebbe essere facilmente rappresentata da un Composite Pattern.
Usare un Composite, inoltre, mi permette di impostare facilmente la dimensione del cubo, e facilita la registrazione delle facce col metodo scan() (si veda la fig. 3.3 e 3.2).

Figura 3.1: *definizione di faccia (Face) e quadrato (Square)*

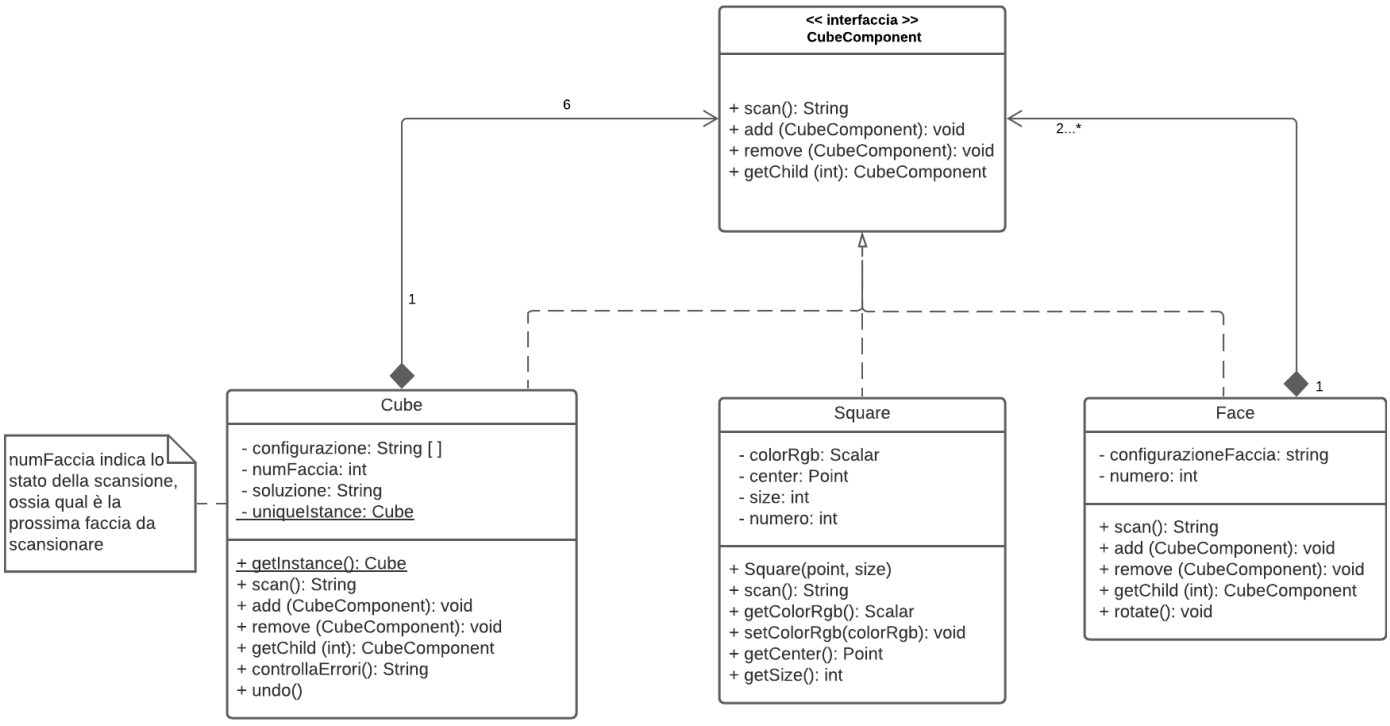


figura 3.2: *Class diagram che mostra la struttura del cubo*

3.2 Iterator

Gli iterator mi permettono di scorrere le diverse Face e Square in modo immediato in un unico ciclo, grazie alla precedente implementazione del Composite Pattern.
L'interfaccia Iterator è già presente in Java.

In figura 3.2 è riportato il class diagram che mostra l'implementazione del cubo con i design pattern sopracitati, e in fig. 3.3 il Sequence diagram che descrive il processo di scansione di una singola faccia.
Il metodo scan() iniziale nel Sequence viene chiamato dalla classe ScanCommand, tramite il suo metodo execute() (si veda più avanti l'implementazione del Command Pattern).

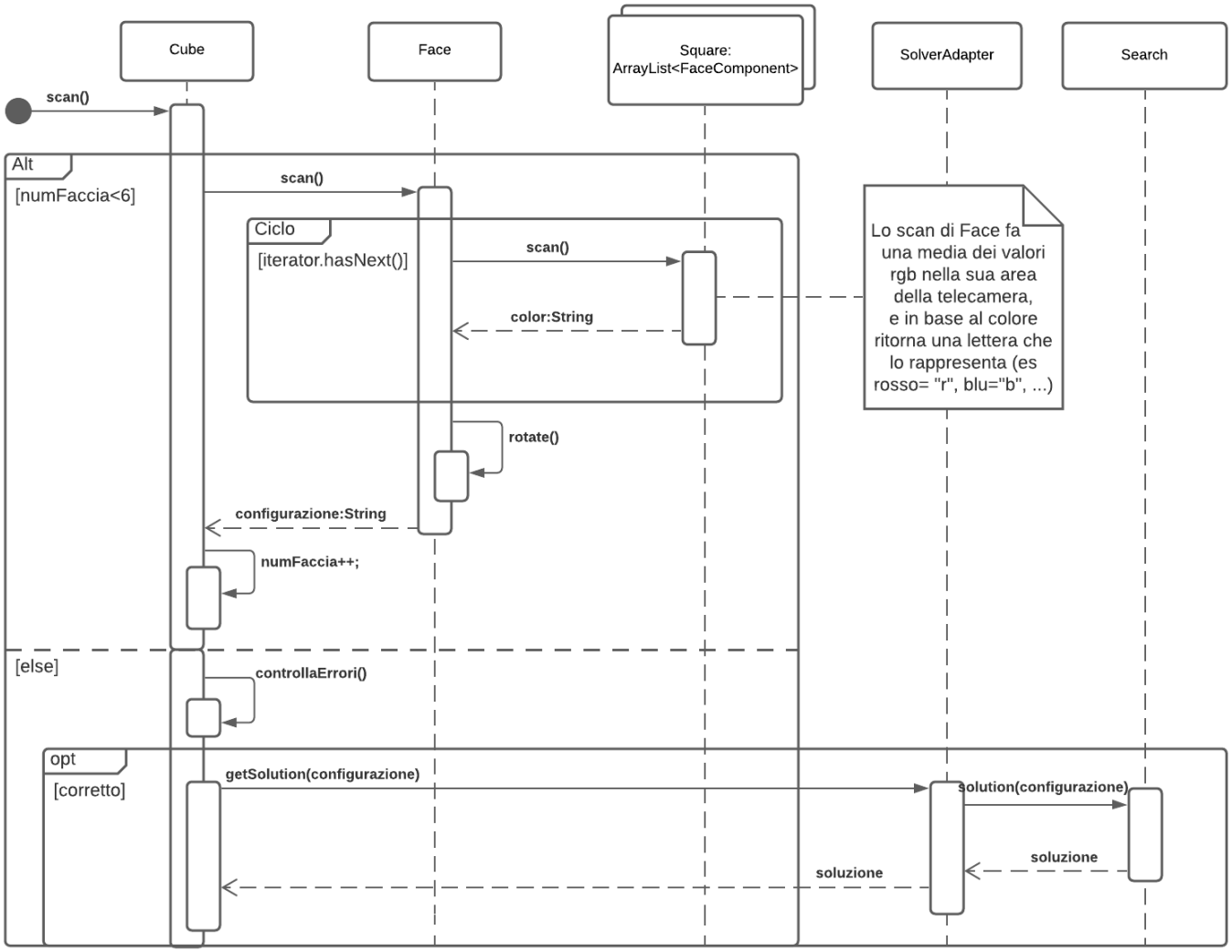


figura 3.3: *Sequence diagram che descrive il processo di scansione della faccia corrente.*

3.3 Command

Implemento il Command Pattern per gestire al meglio le chiamate dei pulsanti.
In questo modo, la classe Scanner dipende solo da interfacce, ed è completamente isolato (cosa che facilita la successiva fase di testing), inoltre non ha dipendenze dirette verso Cube (rispettando il dependency inversion principle). Inoltre, se in futuro saranno cambiati dei metodi in Cube, Scanner non sarà modificato (rispettando così l'open-close principle).
In fig. 3.4 è riportato il class diagram che mostra le principali classi coinvolte.
Nel grafico, non ho collegato InstructionCommand ad altre classi, in quanto il suo metodo execute() chiama l'activity delle istruzioni tramite il metodo startActivity() di Android. Per la superflua difficoltà nel rappresentare tale collegamento, ho preferito non riportarlo, ma il codice è nella nota.

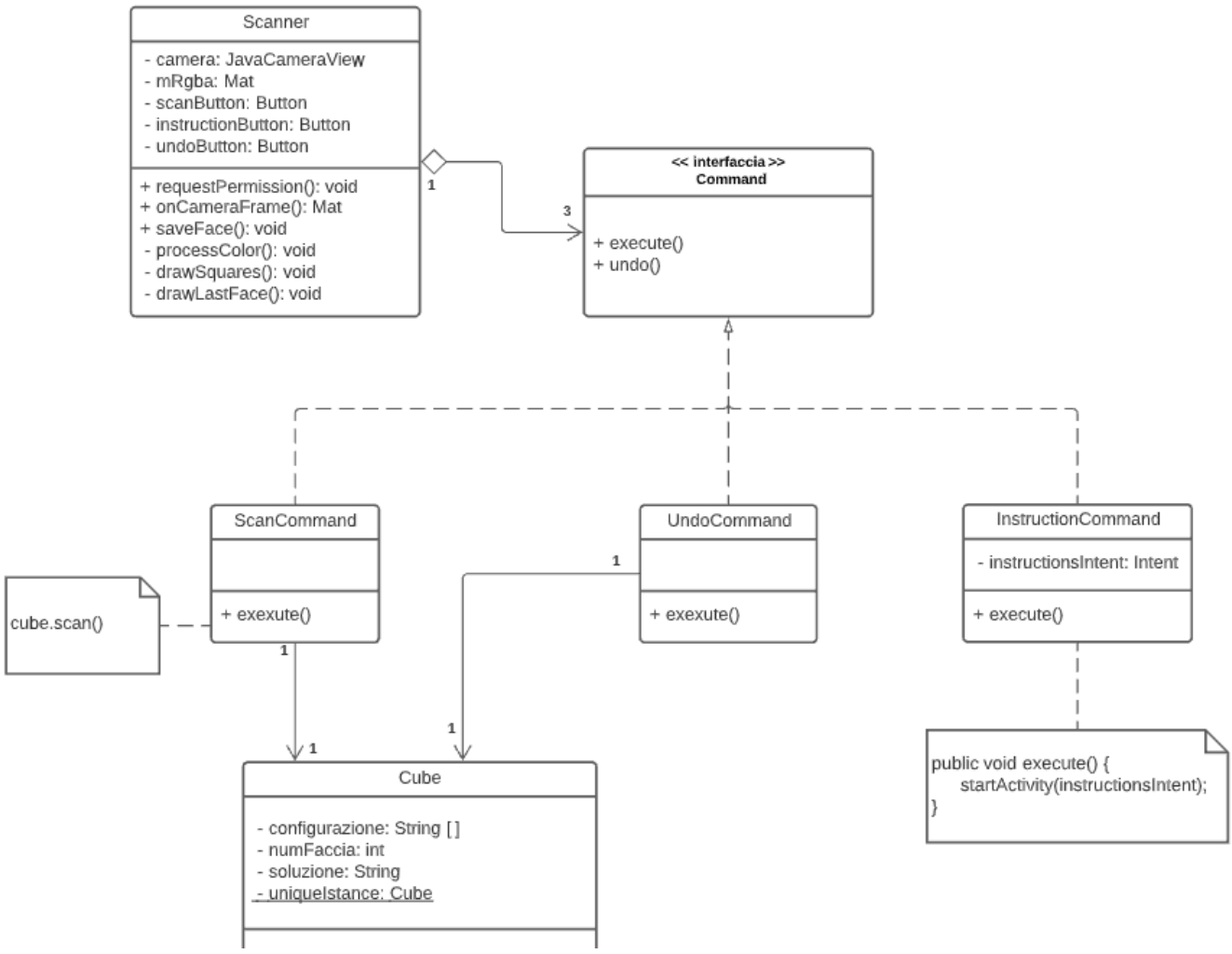


figura 3.4: *Class diagram dello scanner, che tramite il Command pattern si interfaccia col cubo*

3.4 Singleton

Per assicurarmi che la classe Cube sia istanziata solo una volta, la implemento usando il Singleton Pattern.
Utilizzo anche il metodo synchronized di java, per assicurarmi che anche in caso di threads paralleli, non possano verificarsi conflitti, i quali potrebbero portare alla doppia istanziazione della classe. Qui sotto è riportata la parte di codice che ne chiarisce il funzionamento.

```
public class Cube implements CubeComponent {
    private static Cube uniqueInstance;

    private Cube() {}

    public static Cube getInstance() {
        if (uniqueInstance == null)
            synchronized (Cube.class) {
                if (uniqueInstance == null)
                    uniqueInstance = new Cube();
            }
        return uniqueInstance;
    }

    //resto del codice
}
```

3.5 Adapter

Kociemba è una libreria, pertanto ho bisogno di isolarla dalla classe Cube.
Infatti, in futuro potrò sempre decidere di cambiare il modo in cui risolvo il cubo, o magari verrà cambiata dai suoi sviluppatori, e non posso permettermi di dipendere da un modulo così instabile. Potrebbe poi essere necessario dover adattare la stringa della configurazione per l'input specifico che richiede la libreria. Per questi motivi ho ritenuto opportuno implementare il pattern Adapter che possa fungere da intermediario tra Cube e la classe Search del package Kociemba (ho così applicato il dependency inversion principle).

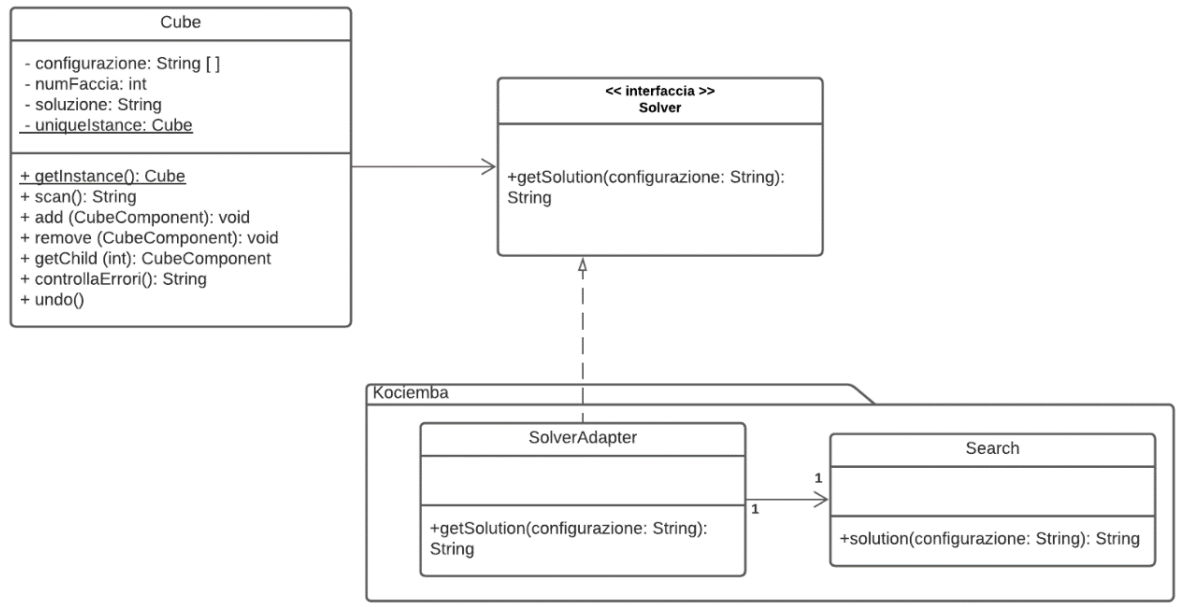


figura 3.5: *class diagram del pattern Adapter*

3.6 Class diagram

Dopo aver implementato tutti i vari design pattern, è riportato in fig. 3.6 il class diagram completo del package Scanner e Cube, per fornire una visione d'insieme.
Mancherebbe la classe ScannerActivity, ossia la classe main, responsabile delle istanziazioni, e che le gestisce il tutto. Ho deciso di non rappresentarla, in quanto non utile a comprendere le scelte di design e la struttura del modulo.

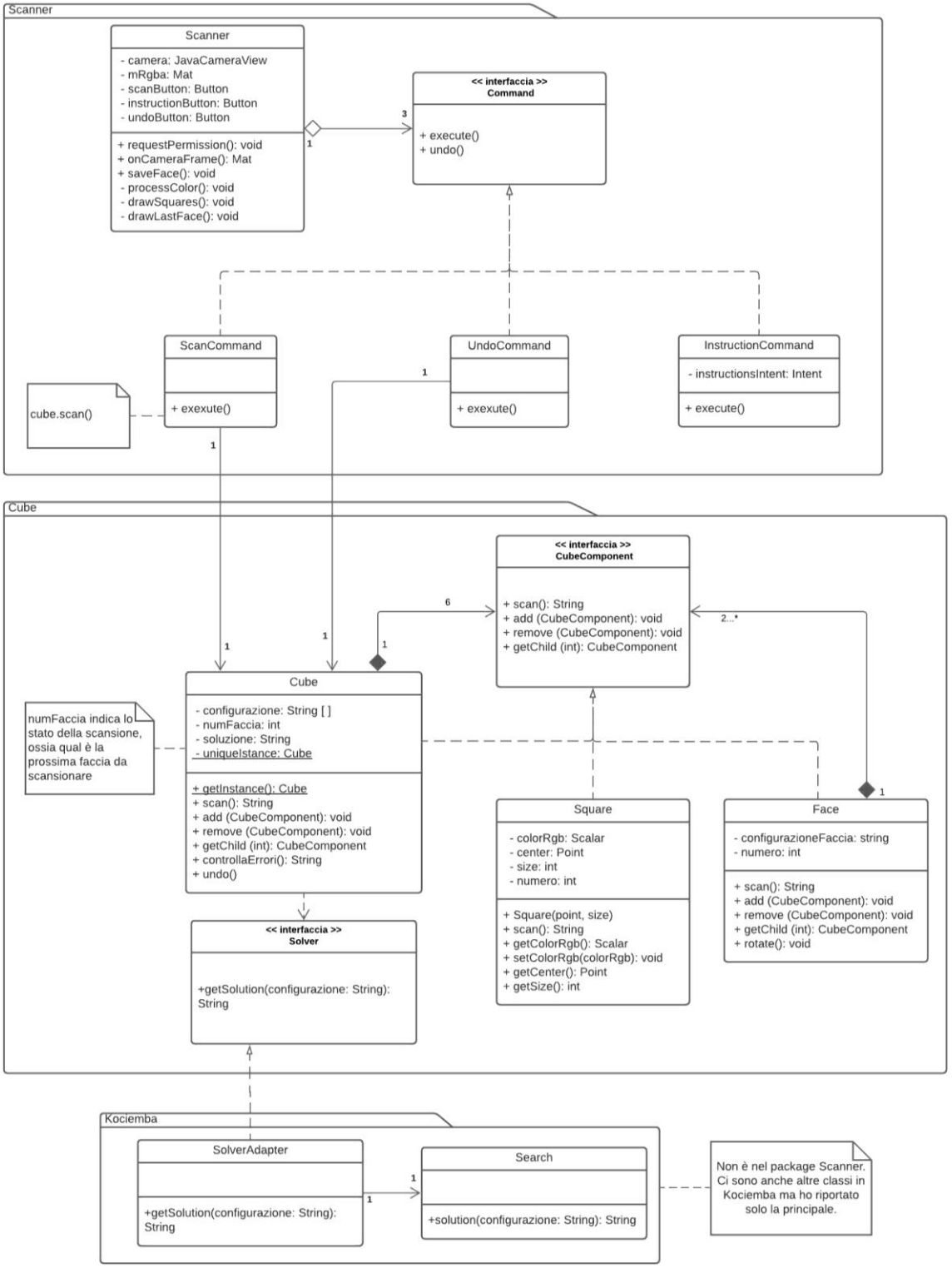


figura 3.6: *class diagram completo* [5]

4. DESIGN PRINCIPLES

4.1 Principi SOLID

Esaminando il class diagram in figura 3.6, e dopo le considerazioni sulle implementazioni dei design pattern, ne risulta che sono stati seguiti quanto più possibile i seguenti principi di design:

- **Single Responsibility**: Ogni classe del modulo Scanner ha solo una ragione di cambiamento, facilitato dal fatto che esiste solo un tipo di attore (l'utente) che può usare il software.
- **Open-Close**: con l'implementazione dei pattern Command e Composite, garantisco che le classi principali (Scanner e Cube) cambino il meno possibile, mentre posso facilmente modificare le loro estensioni.
- **Liskov Substitution**: ogni superclasse può essere sostituibile da uno qualsiasi dei suoi figli.
- **Interface Segregation**: Tutte le sottoclassi usano tutti i metodi ereditati dalle loro superclassi. Ad esempio, tutti i figli di CubeComponent usano il metodo scan().
- **Dependency Inversion**: Non sono presenti dipendenze cicliche, e in genere seguono la direzione della stabilità. L'applicazione del pattern Adapter, ad esempio, rispetta questo principio, disaccoppiando la classe Cube da quelle della libreria Kociemba.

4.2 Coesione e Accoppiamento dei moduli

Si è cercato di mettere negli stessi moduli le classi che riguardano lo stesso aspetto funzionale del software, che poi sono proprio le stesse che cambiano e che si rilasciano insieme. In questo modo sono rispettati i principi Reuse/Release e Common Closure (ad esempio, tutto ciò che riguarda la scansione è nel modulo Scanner).
Ho poi deciso di separare la gestione del cubo dal package di scanner, seguendo il Common Reuse Principle. Questo perché il componente di Cube viene usato sia per la scansione, che per la visualizzazione virtuale del cubo.

Per quanto riguarda l'accoppiamento, facendo riferimento al package diagram (fig. 2.2), sono state evitate dipendenze cicliche, pertanto è soddisfatto l'Acyclic Dependencies Principle.
Il componente più stabile e con grado di astrazione maggiore è Cube; infatti, le dipendenze vanno nella sua direzione (Stable Dependency Principle).
Calcolando l'indice di abstractness del package Cube, ci sono cinque classi totali, due delle quali sono interfacce.
$\frac{5}{2} = 0.4$, ossia quasi la metà delle classi sono astratte. Possiamo pertanto affermare che tale componente sia il più stabile.
Facendo la comparazione con altre classi meno stabili, ad esempio, la libreria Kociemba che ha solo classi concrete (quindi un abstractness uguale a zero), si può concludere che in genere i componenti siano tanto astratti quanto stabili (Stable Attraction Principle).

4.3 Clean architecture

Tutti i package e le classi vanno in direzione della classe Cube, o meglio dell'interfaccia CubeComponent (che rappresenta il Cubo in tutti i suoi aspetti, quindi anche facce e quadrati).
Tale interfaccia è senza dubbio la più stabile dell'intero software: in qualsiasi momento potrò decidere di cambiare il modo in cui visualizzo le mosse, o la libreria per risolvere il cubo, oppure potrò aggiungere funzionalità allo scanner, mentre il modo in cui rappresento il cubo cambierà molto più difficilmente, a meno che non decida di cambiare totalmente l'applicazione.
CubeComponent è infatti la business rule più critica, e rappresenta la Entity principale del software.
Le classi di Scanner, MostraMosse e Menu rappresentano gli Use Case principali (fig. 2.1), mentre Kociemba, OpenCV e OpenGL sono librerie esterne.
Kociemba si collega a Cube tramite classi Adapter, come mostrato precedentemente nel paragrafo 3.5.

In figura 4.1, è riportato il modello a strati che rappresenta graficamente quanto appena affermato.

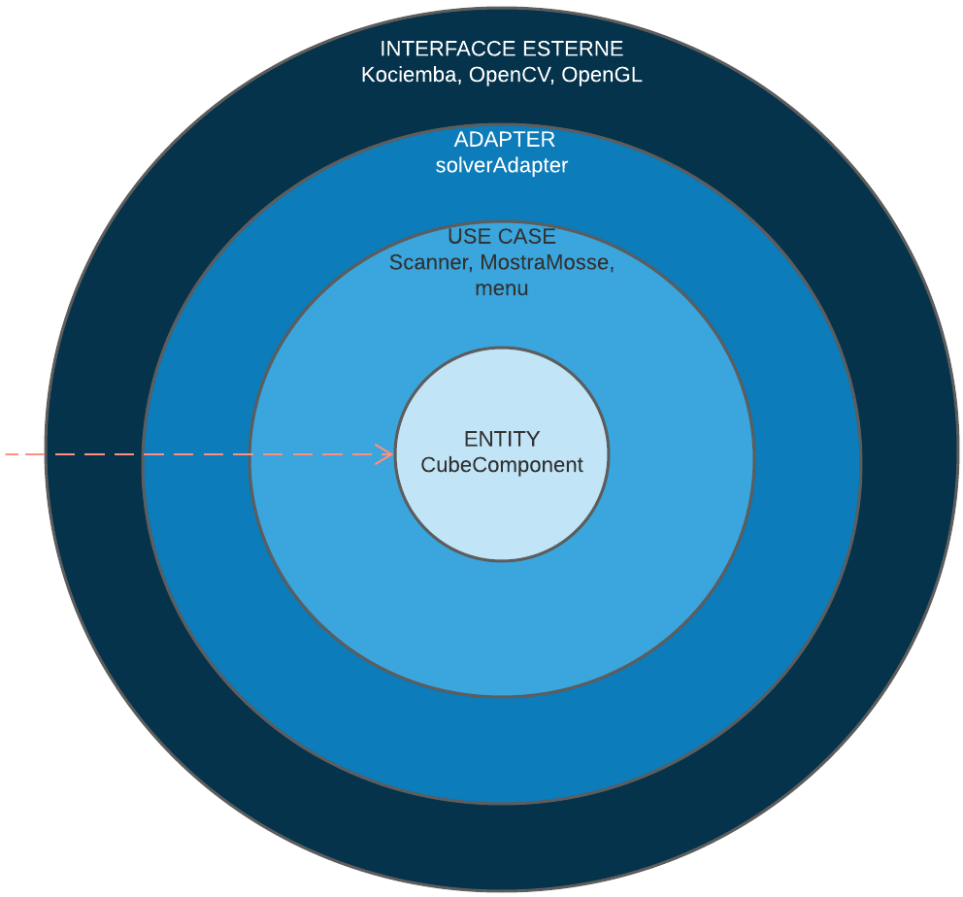


figura 4.1: *modello a strati dell'architettura clean.*
La freccia indica la direzione delle dipendenze.

CONCLUSIONI

Per realizzare questo documento, ho dovuto ripensare completamente la struttura del software, in quanto in realtà la classe di Scanner comprendeva anche Cube e tutta la gestione del cubo, ed era lunga quasi 300 linee di codice. Era perciò molto difficile da gestire (e conteneva anche il main), ma soprattutto sarebbe stato impensabile dover leggere e modificare il progetto per una persona esterna. In pratica, avevo sviluppato tutta la parte di scan e gestione del cubo usando il paradigma procedurale, invece che object oriented.
Dopo aver riprogettato il tutto, applicando i principi studiati nel corso, mi sono effettivamente reso conto degli errori commessi, e dei vantaggi di sfruttare a pieno, e in modo consapevole, la programmazione ad oggetti, i principi di design, e l'effettiva utilità dei design patterns.

NOTE

1) *Link GitHub dell'applicazione: https://github.com/danielenapo/progetto_oop*
2) *Documentazione di OpenCv: https://opencv-java-tutorials.readthedocs.io/en/latest/*
3) *Download di Kociemba: http://kociemba.org/download.htm (la documentazione è inclusa nel file scaricato)*
4) *Documentazione di OpenGl: https://developer.android.com/guide/topics/graphics/opengl*
5) *Link per scaricare l'immagine del class diagram completo dei moduli Scanner e Cube: https://github.com/danielenapo/progetto_oop/blob/master/Class%20diagram.jpeg*