ODD
Object Design Document

English Validation Porting Android

Versione	1.1
Data	14/12/2019
Destinatario	Top Manager: Filomena Ferrucci
Presentato da	Carmine Brancaccio, Antonio Calabrese, Fiorenzo Carino, Michele Costabile, Marco Fresolone, Paolo Schiavo, Stefano Tulino, Simone Veneruso
Approvato da	Alfonso Del Gaizo, Andrea Palladino

Revision History

Data	Versione	Descrizione	Autori
13/12/2019	1.0	Aggiunta capitoli 1,2,3,4 e 5	Carmine Brancaccio, Antonio Calabrese, Paolo Schiavo, Stefano Tulino, Simone Veneruso.
14/12/2019	1.1	Correzioni generali	Simone Veneruso

Contents

1. Introduzione

L'Object Design Document, a differenza dei documenti finora prodotti, ha come scopo quello di fornire una visione più precisa per quanto riguarda l'aspetto implementativo. Infatti, da esso, sarà possibile visionare:

- le interfacce usate nelle classi;
- i parametri utilizzati nelle procedure;
- le principali operazioni supportate e i tipi di dati;
- linee-guida;
- trade-offs.

Inoltre dalla documentazione precedentemente espressa, in particolar modo dal System Design Document, potremo osservare i vari sottosistemi creati e altre informazioni utili per la compilazione del nostro ODD.

1.1. Object design trade-offs

I trade-offs sono prettamente scelte che riguardano la qualità del nostro software. Ovviamente, l'obiettivo è quello di trovare il giusto equilibrio tra le due scelte, in modo tale da non avere perdite sulla qualità in generale. Tra i vari trade-offs, i più comuni sono:

Comprensibilità vs Costi:
Uno degli obiettivi dell'implementazione sarà quello di scrivere del codice chiaro, anche aggiungendo in alcuni casi commenti risolutivi. L'app cercherà di impiegare meno risorse possibili, in modo tale da poter avere alla fine un prodotto completo ma leggero.

Prestazioni vs Costi:
L'app sarà implementata preferendo prestazioni elevate in qualunque caso possibile, pur non impiegando eccessive risorse a livello economico.

Costi vs Manutenzione:
L'app sarà implementata preferendo la manutenzione ad un Sistema molto più articolato del dovuto. Ciò favorisce anche la comprensibilità, agevolando il processo di manutenzione e di modifica del progetto anche per futuri sviluppatori che non hanno lavorato dall'inizio al progetto stesso.

Memoria vs Efficienza:
L'app dovrà permettere l'efficienza a discapito della memoria utilizzata così da dare la possibilità all'utente finale di fare qualunque richiesta possibile al suo interno.

Tempo di Risposta vs Spazio di Memoria:
Se il software non rispetta i requisiti di tempo di risposta e di throughput, è necessario utilizzare più memoria per velocizzare il sistema. Se il software non rispetta i requisiti di memoria, si possono adottare compressioni a discapito della velocità. Il sistema quindi sarà sviluppato in maniera tale da bilanciare i due requisiti non funzionali.

Interfaccia vs Easy-use:
Quest'app nasce per rendere più intuitive varie operazioni collegate alla tematica della validazione dei certificati in lingua straniera. Per questo motivo verrà realizzata un'interfaccia grafica chiara e concisa, usando activity e button i quali avranno lo scopo di rendere semplice l'utilizzo del sistema da parte dell'utente finale.

1.2. Linee guida per la documentazione di interfacce

Gli sviluppatori dovranno seguire precise linee guida per la stesura del codice, il quale sarà implementato utilizzando un linguaggio Android.

Naming Convention:
- Descrittivi;
- Pronunciabili;
- Di lunghezza media;
- Non abbreviati;
- Utilizzando solo caratteri consentiti.

Variabili:
- I nomi delle variabili dovranno iniziare con lettera minuscola.

Metodi:
- I nomi dei metodi avranno come prima lettera iniziale quella
- minuscola e poi, a seconda se il nome sia composto o semplice, potrà avere lettere minuscole (nel caso in cui sia semplice);
- I metodi dovranno essere raggruppati in base alle loro funzionalità;
- Ogni metodo dovrà avere una descrizione delle sue funzionalità, parametri di input e output.

Classi Node.js
- I nomi delle classi dovranno iniziare con lettera maiuscola e così come le parole successive all'interno del nome;
- I nomi delle classi dovranno essere esplicativi e senza abbreviazioni;
- Ogni classe dovrà avere una breve lista dei metodi che implementa.

Packages:
- I nomi dei packages dovranno iniziare con lettera minuscola;
- Non saranno ammessi caratteri speciali.

1.3. Definizioni acronimi e abbreviazioni

1.3.1. Definizioni

- Design Pattern: descrizione o modello logico da applicare per la risoluzione di un'problema incontrato durante le fasi di progettazione e sviluppo del software.
- Package: collezione di classi e interfacce correlate.
- Package model: pacchetto che include i model del progetto, ossia la componente del modello MVC che si occupa dell'interazione tra l'applicazione e il database.
- MVC: modello architetturale che si basa sull'utilizzo di 3 componenti fondamentali (Model, View, Controller) per lo sviluppo di app.
- Class diagram: Tipo di diagramma che descrive la struttura di un sistema mostrando le sue classi, gli attributi di tali classi, le operazioni (o metodi) e le relazioni tra gli oggetti.

1.3.2. Acronimi e abbreviazioni

PEV_RAD_Vers.1.7: Utilizzata per indicare il Requirements Analysis Document.
PEV_SDD_Vers.1.5: Utilizzata per indicare il System Design Document.
PEV_ODD_Vers.1.1: Utilizzata per indicare l'Object Design Document.

1.4. Riferimenti

Libri:

- Kathy Schwalbe, “Information Technology Project Management”, International Edition 7E, Cengage Learning, 2014;
- Bernd Bruegge, Allen H. Dutoit, “Object-Oriented Software Engineering Using UML, Patterns and Java”, Third Ed., Pearson, 2010.

Guide:

- Tutorial per l’utilizzo di Firebase;
- Documentazione degli strumenti sviluppatore per Android
- (https://developer.android.com/docs) ;
- Tutorial per l’utilizzo di Node.js;
- Slide delle lezioni della Prof.ssa F. Ferrucci e del Prof. C. Gravino;
- Slide delle lezioni del corso di Android.

2. Design Pattern

2.1. Design pattern Singleton

Il singleton è un design pattern che permette di garantire che di una determinata classe venga create una ed una sola istanza in modo da fornire un punto di accesso globale alla classe.
Una delle caratteristiche di questo pattern è la possibilità di avere più tipi di implementazione in modo da permettere un’implementazione più semplice e conforme alla situazione.
Nel nostro caso, utilizzeremo il principio della lazy initialization (letteralmente "inizializzazione pigra") in quanto la creazione dell'istanza della classe viene rimandata nel tempo e messa in atto solo quando ciò diventa strettamente necessario (di solito, al primo tentativo di uso).

Il design pattern Singleton, di tipo lazy, ha come scopo:

- La presenza di un accesso controllato all’unica istanza della classe.
- Ridurre il numero di oggetti condivisi.
- Centralizzare informazioni e comportamenti in un'unica entità condivisa dagli utilizzatori.

Il principale vantaggio offerto è: Mutua esclusione.

Utilizzo:
Il design pattern verrà utilizzato per mantenere la sessione, permette di memorizzare informazioni e accedere a Firebase e infine permette di mantenere informazioni sempre vive durante la navigazione

2.2. Design pattern DAO

DAO (Data Access Object) è un pattern architetturale usato per la gestione della persistenza:
si tratta di una classe con relativi metodi che rappresenta un'entità tabellare il cui uso è fornito per applicazioni Android.

Il design pattern DAO (Data Access Object) ha come scopo:

- L’utilizzo di un meccanismo di accesso richiesto per lavorare con la sorgente dei dati, che potrebbe essere un servizio esterno o un repository di file.

- L'utilizzo del modulo client (che contiene la business-logic) che utilizza l'interfaccia esposta dal DAO. Il Data Access Object nasconde completamente i dettagli dell'interazione con la sorgente dati. L'interfaccia esposta dal DAO al client non cambia quando l'implementazione dell'origine dati sottostante cambia, in modo da consentire al DAO di adattarsi a diversi schemi di archiviazione senza dover modificare nulla.

Il principale vantaggio offerto è nella sua caratteristica :
funge da adapter tra il componente della business logic e l'origine dati.

Utilizzo:
Il design pattern verrà utilizzato per la gestione della classe centrale di Firebase tramite le quali accediamo a tutti i servizi disponibili della nostra applicazione.

2.3. Design pattern Observer

Il pattern Observer (noto anche col nome Publish-Subscribe) permette di definire una relazione uno a molti tra oggetti, in modo che un oggetto cambi stato e tutte le sue dipendenze vengano notificate e aggiornate automaticamente. Nasce dall'esigenza di mantenere un alto livello di consistenza fra classi correlate, senza produrre situazioni di forte dipendenza e di accoppiamento elevato. Si può considerare l'utilizzo di questo modello nell'applicazione quando più oggetti dipendono dallo stato di un oggetto in quanto fornisce un design pulito e ben testato per lo stesso. Il pattern Observer è strutturato nel seguente modo:

- Il Subject rappresenta l'astrazione principale (indipendente), è un oggetto con un uno stato rilevante, poiché notificherà anche ad altri oggetti le modifiche spesso viene chiamato Publisher.
- Il Publisher notifica eventi d'interesse per altri oggetti. Questi eventi si verificano quando il Publisher cambia stato o esegue alcuni comportamenti. I Publisher contengono una infrastruttura di Subscriber che inserisce i nuovi Subscribers e toglie i vecchi. Quando si verifica l'evento, il Publisher esamina l'elenco dei Subscriber e chiama il metodo di notifica dichiarato nell'interfaccia del Subscriber su ogni suo oggetto.
- L'interfaccia del Subscriber dichiara di notifica. Nella maggior parte dei casi, è costituito da un singolo update metodo. Il metodo può avere diversi parametriche consentono al Publisher di passare dettagli dell'evento insieme all'aggiornamento.
- I Subscriber eseguono alcune azioni in risposta alle notifiche emesse dal Publisher. Tutte queste classi devono implementare la stessa interfaccia in modo che l'editore non sia associato a classi concrete. Di solito i Subscriber hanno bisogno di alcune informazioni contestuali per gestire correttamente l'aggiornamento. Per questo motivo, i Publishers spesso trasmettono alcuni dati di contesto come argomenti del metodo di notifica. Il Publisher può passare se stesso come argomento, consentendo all'utente di recuperare direttamente tutti i dati richiesti.
- Il Client crea oggetti Publisher separatamente e quindi registra gli abbonati per gli argomenti publisher.

Il pattern Observer quindi verrà utilizzato per i listener degli oggetti grafici e le chiamate http.

2.4. Design pattern Adapter

Il pattern Adapter (noto anche come decoration pattern) viene usato quando è necessario inglobare componenti esistenti nel Sistema. Questo tipo di soluzione viene adottata spesso nei sistemi interattivi, dove si usano finestre, dialog e bottoni.

Un Adapter avvolge uno degli oggetti per nascondere la complessità della conversione che sta avvenendo dietro le quinte. L'Adapter è un componente che si occupa della rappresentazione grafica dei dati e dell'interazione con essi.
Gli Adapter possono non solo convertire i dati in vari formati, ma possono anche aiutare gli oggetti con interfacce diverse a collaborare.

1. L'Adapter ottiene un' interfaccia, compatibile con uno degli oggetti esistenti.
2. Utilizzando questa interfaccia, l'oggetto esistente può chiamare in modo sicuro i metodi dell'Adapter.
3. Alla ricezione di una chiamata, l'Adapter passa la richiesta al secondo oggetto, ma in formato e in ordine previsti dal secondo oggetto.

A volte è possibile creare un Adapter a due vie in grado di convertire le chiamate in entrambi le direzioni.

Il codice viene regolato per comunicare con la libreria solo tramite gli Adapter, quando questo riceve una chiamata, traduce i dati in entrata e passa la chiamata ai metodi appropriate di un oggetto analitico spostato.

Il pattern Adapter è strutturato nel seguente modo:

- Il Client è una classe che continua la logica aziendale esistente nel programma.
- Il Client Interface un protocollo che le altre classi devono seguire per poter collaborare con il codice client.
- Il Service è una classe utile (di solito di terze parti o legacy). Il client non può utilizzare questa classe direttamente perché ha un' interfaccia incompatibile.
- L'Adapter è una classe che in grado di funzionare sia con il Client che con il servizio: Implementa l'interfaccia client, mentre incorpora l'oggetto del servizio. L'Adapter riceve le chiamate dal client tramite l'interfaccia dell'adattatore e le traduce in chiamate verso l'oggetto di servizio convertito in un formato comprensibile.

È possibile introdurre nuovi tipi di Adapter nel programma senza rompere il codice client esistente. Questo può essere utile quando l'interfaccia della classe di servizio viene cambiata o sostituita: si può semplicemente creare una classe Adapter senza cambiare il codice client.

Il pattern Adapter quindi verrà utilizzato spesso per oggetti di tipo ListView.

3. Packages

Cartelle

- adapter: Contiene tutti gli adapter
- firebaseArchive: Contiene tutti i bean.

- gestioneAdmin: Contiene la Main Activity dell'Admin.
- gestioneSegreteria: Contiene la Main Activity della Segreteria.
- gestioneStudente: Contiene la Main Activity dello Studente.
- gestioneUtente: Contiene la Main Activity dell'Utente.
- network: Contiene il collegamento con Node.js.
- utility: Contiene il Singleton.

3.1. Package Adapter

Classe	Descrizione
RequestAdapter.java	Classe che interfaccia le richieste con le ListView.

3.2. Package Firebase

Classe	Descrizione
FirebaseArchive	Classe che si interfaccia con Firebase.

Classe	Descrizione
RequestBean.java	Classe che consente di salvare localmente i dati Segreteria prelevati dal database.
UtenteBean.java	Classe che consente di salvare localmente i dati Utente prelevati dal database.

3.3. Package MainActivityAdmin

Classe	Descrizione
MainActivityAdmin	Classe che rappresenta la schermata principale dell'Admin.

3.4. Package MainActivitySegreteria

Classe	Descrizione

MainActivitySegreteria	Classe che rappresenta la schermata principale della Segreteria.

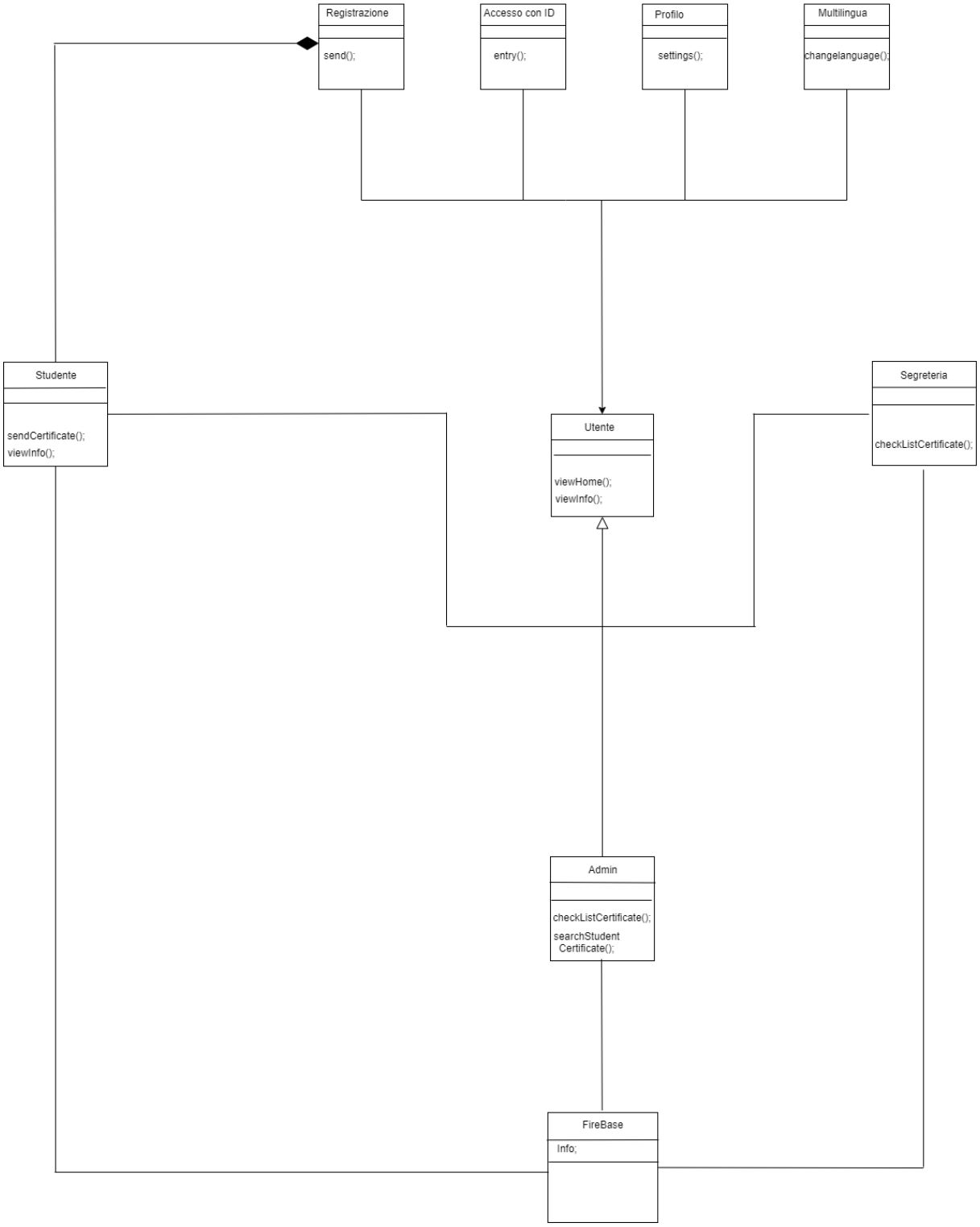
3.5. Package MainActivityStudente

Classe	Descrizione
MainActivityStudente	Classe che rappresenta la schermata principale dello Studente.
Register.java	Classe che rappresenta la registrazione di uno studente.

3.6. Package MainActivityUtente

Classe	Descrizione
EditActivityUtente.java	Classe che consente di modificare il profilo Utente.
LoginActivity.java	Classe che rappresenta il login dell'Utente.
ViewActivityUtente.java	Classe che consente di visualizzare il profilo Utente.

4. Interfacce delle classi



Nome file	Profilo.java
Descrizione	Permette all'utente di eseguire funzionalità riguardante il proprio profilo, come Modifica e Visualizzazione del profilo.

Pre-condizione	-
Post-condizione	-
Invarianti	-

Nome file	Multilingua.java
Descrizione	Permette all'utente di selezionare la lingua dell'applicazione fra tre possibili scelte : Italiano, Inglese e Spagnolo.
Pre-condizione	-
Post-condizione	-
Invarianti	-

Nome file	Admin.java
Descrizione	Contiene tutte le possibili funzionalità dell'Admin, come: Visualizzazione lista richieste. Verifica certificato tramite codice. Approvare o rifiutare certificazioni e organizzarle in due liste separate. Correzione errori certificato. Invio email di verifica certificazione. Visualizza numero di documenti non ancora revisionati.
Pre-condizione	-
Post-condizione	-
Invarianti	-

Nome file	Utente.java

Descrizione	Contiene tutte le possibili funzionalità di un qualunque utente generico, quali: Supporto multilingue; Autenticazione tramite impronta digitale; Profilo, il quale contiene visualizzazione e modifica del suddetto.
Pre-condizione	-
Post-condizione	-
Invarianti	-

Nome file	Studente.java
Descrizione	Contiene tutte le possibili funzionalità dello Studente, quali: Registrazione; Compilazione form richiesta; Caricamento allegati; Accesso area utente; Notifica stato della pratica; Visualizza QR code della pratica.
Pre-condizione	-
Post-condizione	-
Invarianti	-

Nome file	Registrazione.java
Descrizione	Consente ad uno studente di potersi registrare, purché sia in possesso di tutti i suoi relativi dati, compresa email istituzionale.
Pre-condizione	L'utente dev'essere uno studente.
Post-condizione	Registrazione avvenuta, con i suddetti dati caricati su Firebase.
Invarianti	-

Nome file	AccessoImpronta.java

Descrizione	È una classe che consente a tutti gli utenti di poter accedere alla propria area personale tramite la scansione dell'impronta digitale, ovvero: Area Utente; Area Studente; Area Segreteria Area Admin.
Pre-condizione	L'utente deve essere registrato.
Post-condizione	-
Invarianti	-

Nome file	Segreteria.java
Descrizione	È una classe che consente tutte le possibili funzionalità della Segreteria, quali: Visualizzazione lista con le richieste; Aggiunta numero di CFU alla richiesta; Correzione eventuali errori commessi; Invio automatico mail con esito; Lettura QR Code; Notifica arrivo nuovi protocolli.
Pre-condizione	-
Post-condizione	-
Invarianti	-

Nome file	FirebaseArchive.java
Descrizione	È una classe che ci permette di connetterci con Firebase e di seguire tutte le operazioni di persistenza dei dati e dei servizi.
Pre-condizione	-
Post-condizione	-
Invarianti	-

5. Glossario

- **Utente:** rappresenta l'utilizzatore del sistema.
- **Studente:** rappresenta un utente autenticato che può effettuare diverse operazioni nel sistema come ad esempio una richiesta per la convalida dei certificati.
- **Segreteria:** rappresenta un utente autenticato che può effettuare diverse operazioni nel sistema interagendo con lo studente e con l'admin.
- **Admin:** rappresenta un utente autenticato che può effettuare diverse operazioni nel sistema interagendo con lo studente e con la segreteria.
- **Dialog:** è una piccola finestra che richiede all'utente di prendere una decisione o inserire informazioni aggiuntive.
- **Bottoni**: sono elementi dell'interfaccia utente che l'utente può cliccare per eseguire un'azione.
- **Profilo Utente:** pagina che mostra tutte le informazioni relative ad un utente;