Laurea Magistrale in informatica-Università di Salerno
Corso di *Gestione dei Progetti Software*- Prof.ssa F.Ferrucci

ERASMUSSMART

Object Design Document

Riferimento	
Versione	1.0
Data	16/12/2018
Destinatario	Prof.ssa F. Ferrucci
Presentato da	Andrea Valenti Ruggero Tammaro
Approvato da	

Revision History

Data	Versione	Descrizione	Autori
08/12/2018	0.1	Prima stesura	Tutto il team
16/12/2016	1.0	Revisione e modifiche	Andrea Valenti Ruggero Tammaro

Sommario

1.1 Object Design trade-offs

Efficienza vs. Tempo:

Si preferisce aggiungere tempo per l'acquisizione di competenze necessarie per utilizzare Firebase e Angular con ionic Framework, cosi che il sistema venga progettato in modo efficiente, soprattutto grazie all'ausilio di questi ultimi che forniscono strumenti per facilitare la gestione dei dati e aumentare la qualità dei componenti, in particolare Firebase per l'implementazione della chat.

Comprensibilità vs costi:

Si preferisce aggiungere costi per la documentazione al fine di rendere il codice comprensibile anche alle persone non coinvolte nel progetto o le persone coinvolte che non hanno lavorato a quella parte in particolare. Commenti diffusi nel codice facilitano la comprensione, di conseguenza migliorare la comprensibilità agevola il mantenimento e anche il processo di modifica, si sceglierà invece di risparmiare sugli strumenti per implementare e organizzare il tutto, in quanto saranno tutti open source (balsamiq, visual studio code, visual Paradigm , ionic Framework , Trello, Slack, GitHub...) .

Spazio di memoria vs. Tempo di risposta:

Anche se non ci sono vincoli imposti per quanto riguarda il tempo di risposta, abbiamo voluto comunque tenerlo il più basso possibile, la maggior parte dei dati persistenti del software, in termini di quantità, sono calcolabili a priori ma non si può sapere nulla riguardo al numero di messaggi che gli studenti possono scambiare con il tutor, quindi per facilitare la gestione del database e diminuire il tempo di risposta si è scelto di utilizzare Firebase, che sincronizza i dati in tempo reale con Cloud Firestore (il quale mette a disposizione 10GB/Mese di "Bandwidth" e 1GB di "Stored Data" per la versione free, ampliabile fino a 500GB+ su scelta del cliente),inoltre si preferisce aggiungere dati ridondanti e quindi appesantire lo spazio di memoria per assicurare un tempo di risposta massimo di 3 sec .

1.2 Componenti off-the-shelf

Per il progetto software che si vuole realizzare facciamo largo uso di componenti off-the-shelf, che sono componenti software disponibili sul mercato già testati e funzionanti, per facilitare la creazione del progetto, scelta conseguente anche al tipo di linguaggi utilizzati per l'implementazione che hanno alla base il riutilizzo delle componenti.

Per il sistema che si vuole realizzare ci interessano framework per applicazioni web e per componenti grafici.

I framework che andremo ad utilizzare sono Angular e Ionic.

Sono entrambi framework open source per sviluppo di applicazioni per il Web, Ionic in particolare contiene varie componenti dell'interfaccia, come moduli, bottoni e navigazione, e altri componenti dell'interfaccia.

Per la persistenza dei dati utilizzeremo la piattaforma di sviluppo offerta da Google, "Firebase", che ci consente di accedere via cloud ad una serie di servizi tra cui anche un Database realtime NOSQL, e i servizi di autenticazione per gli utenti.

1.3 Linee guida Documentazione delle Interfacce

Nella fase di implementazione del sistema, gli sviluppatori dovranno attenersi alle seguenti linee guida.

1.3.1 Pagine HTML

Il codice HTML rappresenta la struttura delle pagine dell'applicazione che verrà realizzata, mentre la parte di stile viene realizzata con il codice CSS. La versione di riferimento del linguaggio che verrà utilizzata è la versione 5. Ogni tag di apertura deve essere necessariamente seguito dall'apposito tag di chiusura (eccetto i tag self-closing).

L'indentazione deve essere effettuata con un TAB.

I tag devono essere opportunamente indentati.

Es.

```
<html>
<head>
</head>
<body>
</body>
</html>
```

Deve essere sostituito da:

```
<html>
```

<head>
</head>
<body>
</body>
</html>.

1.3.2 Fogli di stile CSS

La versione di riferimento del linguaggio che andremo ad utilizzare è la versione 3. Gli stili in comune che potranno essere utili in più punti devono essere inseriti in un foglio di stile globale, mentre gli stili definiti per una singola pagina vanno nei fogli di stile CSS presenti nella stessa cartella dove sono contenuti i file HTML a cui si riferiscono. Solo la palette di colori che definisce il tema della nostra applicazione va in un file apposito fornito da Ionic, il file "variables.scss".

1.3.3 Script TypeScript

Il linguaggio che verrà utilizzato per rendere dinamiche le pagine HTML della nostra applicazione è il linguaggio TypeScript, una versione rivisitata del linguaggio JavaScript. Come suggerisce il nome, questo linguaggio aggiunge al linguaggio JavaScript delle feature simile a quelle viste per altri linguaggi OO, come definizioni di classi, interfacce o utilizzo di variabili tipizzate ecc.
Le convenzioni interne da seguire per realizzare gli algoritmi in TypeScript sono le seguenti:

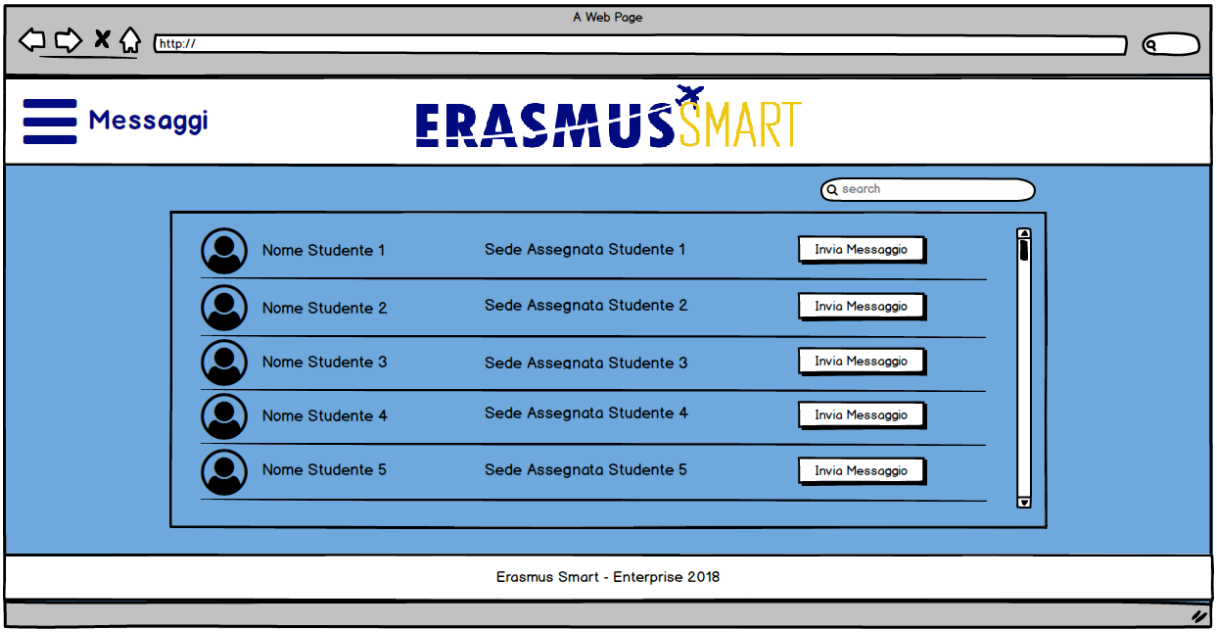
- Ogni metodo e ogni file devono essere preceduti da un commento, o più precisamente da una documentazione che riporti l'obiettivo che si vuole e deve raggiungere con il nome/i dell'autore/i. Inoltre, bisogna commentare, giustificare delle decisioni particolari o dei calcoli.
  /*
  <SCOPO DEL METODO>
  <AUTORE/I>
  <EVENTUALI MOTIVAZIONI>
  */
- La notazione da usare per assegnare nomi alle variabili e ai metodi è la nota Camel Notation.
- I nomi dei file, delle operazioni e delle variabili devono essere in lingua inglese per permettere una lettura del codice non limitata ai programmatori italiani.

- Posizionare le dichiarazioni all'inizio dei blocchi. Non dichiarare le variabili al loro primo uso, può confondere il programmatore inesperto e impedire la portabilità del codice dentro lo scope. L'unica eccezione a questa regola sono gli indici dei cicli for che possono essere dichiarati nell' istruzione stessa. Evitare dichiarazioni locali che nascondono dichiarazioni a più alto livello. Ad esempio, non dichiarare una variabile con lo stesso nome in un blocco interno anche se consentito può causare errori.
- A prescindere dalle istruzioni che seguono un IF, è necessario, laddove ci fosse anche una sola istruzione, riportare il blocco di istruzioni tra parentesi graffe. Non seguire questa regola causa bug difficili da individuare se non si presta attenzione alla convenzione dell'assenza della parentesi.
- Una convenzione importante, per quanto riguarda l'inserimento di numeri o di valori costanti, è quella di non usare una codifica fissa (hard coding), ma inserire il valore in file esterno. In questo modo una modifica durante una revisione è più veloce e sicura, perché se si usa quel valore in più punti non c'è rischio di non aggiornare correttamente ogni riferimento. In particolare, per le stringhe Angular fornisce un modulo per l'internazionalizzazione, che permette di selezionare la lingua del nostro sito e sceglire un file di labels a seconda della scelta effettuata.
- Quando un'espressione supera la lunghezza della linea, occorre spezzarla secondo i seguenti principi generali:
    - Interrompere la linea dopo una virgola;
    - Interrompere la linea prima di un operatore;
    - Preferire interruzioni di alto livello rispetto ad interruzioni di basso livello (interrompere laddove non si interrompe un discorso logico, discorso valido soprattutto per le formule es. (3+4) * 2 interrompere prima della moltiplicazione senza spezzare gli operandi in parentesi);
    - Allineare la nuova linea con l'inizio dell'espressione nella linea precedente;
    - Se le regole precedenti rendono il codice più confuso o il codice è troppo spostato verso il margine destro, utilizzare solo otto spazi di indentazione.
- Indentare opportunamente le istruzioni attraverso l'utilizzo di un TAB. Utilizzare questa pratica soprattutto per le istruzioni FOR e IF.

1.3.4 Struttura progetto Ionic-Angular

Alle convenzioni precedenti sugli algoritmi vanno aggiunte quelle sulla struttura del progetto che sono definite dal framework Ionic. In particolare, di questo framework utilizziamo la versione 3. C'è da dire che Ionic per la parte di logica e comunicazione tra i componenti fa uso di Angular, poiché esso in sé fornisce solo delle feature per realizzare uno stile grafico responsive e portabile anche su dispositivi mobile. Nello specifico la versione 3 di Ionic usa la versione 5 di Angular.
Le convenzioni da seguire per creare un progetto che rispetta lo standard Ionic 3 e Angular 5 sono:

- Pensare le pagine a componenti, ossia prima di realizzare una pagina HTML cercare di individuare di quali componenti quella pagina è formata. Quindi ad esempio, di seguito consideriamo il mock-up della pagina di News della nostra applicazione:



  In questa pagina si possono individuare i seguenti componenti:
  - Form di ricerca
  - Header della pagina
  - Footer della pagina
  - La singola news
  - La tabella che contiene la lista di news

  Questi componenti devono essere realizzati nell'apposita cartella "components" e utilizzando questo approccio ad esempio il component HeaderES e FooterES si può riutilizzare in ogni pagina che lo prevede andando ad avere un riutilizzo del codice.

- I metodi per gestire il Database, la login, le news, etc. vanno inseriti in dei particolari file chiamati providers. Quindi ogni qualvolta che pensiamo che un metodo possa offrire servizi globali a tutta l'applicazione va in un opportuno providers. Come per i componenti anche i providers vanno nell'apposita cartella "providers"
- Ogni pagina che è stata individuata nei mock-up deve essere realizzata nell'apposita cartella "pages" e deve avere al suo interno:
    - L'import di tutti i componenti che la compongono
    - L'import di tutti i providers che gli offrono servizi
    - Il path del file html che definisce la sua struttura
    - Il path del file CSS che definisce il suo stile

  Tutte queste informazioni vanno nel file TypeScript.
- Tutti gli asset statici della nostra applicazione (Immagini, logo, file di labels) vanno nella cartella "assets"
- I modelli che rispecchiano la struttura degli oggi presenti nel Database vanno nella cartella "model" e sono delle semplici classi TypeScript (Simili al concetto di Bean Java).

1.3.5 Organizzazione del repository

Alle convenzioni precedenti vanno aggiunte delle convenzioni per condividere il proprio lavoro attraverso un repository Git. Ogni membro del team deve realizzare il proprio codice in un branch personale in modo da non causare problemi agli altri membri del team, utilizzare il tool Git Flow che automatizza questo processo. Una volta che si è sicuri del proprio artefatto contattare i manager per far revisionare e testare il codice prodotto. Quando viene dato il via libera si può effettuare un merge del proprio branch sul branch comune di sviluppo chiamato develop. Quando deve essere realizzata una versione stabile del codice per una demo si fa il merge del develop nel master e si crea un tag con il numero della versione. Infine, cercare di effettuare dei commit con un commento anche minimo che spiega cosa viene aggiunto al codice.

1.4 Definizioni, acronimi e abbreviazioni

Off-The-Shelf: Servizi esterni di cui viene fatto utilizzo da terzi.

Framework: Software di supporto allo sviluppo web.

Plugin: Un programma non autonomo che interagisce con un altro programma per ampliarne le funzioni.

HTML: Linguaggio di Mark-up per pagine web.

CSS: Linguaggio usato per definire la formattazione di pagine web.

Bootstrap: Framework che contiene librerie utili per lo sviluppo responsive di pagine web.

Angular: è un framework per applicazioni web open source principalmente sviluppato da Google e dalla comunità di sviluppatori individuali nato per affrontare le molte difficoltà incontrate nello sviluppo di applicazioni su singola pagina.

Ionic: Framework che consente di sfruttare al meglio le tecnologie Web per creare applicazioni mobile con look and feelsimile a quelle native e in questo ambito si rivela tra le soluzioni di maggior successo. Ma per capire meglio dove si colloca Ionic Framework nel panorama dello sviluppo mobile, dobbiamo fare un breve riepilogo dell'attuale stato dell'arte.

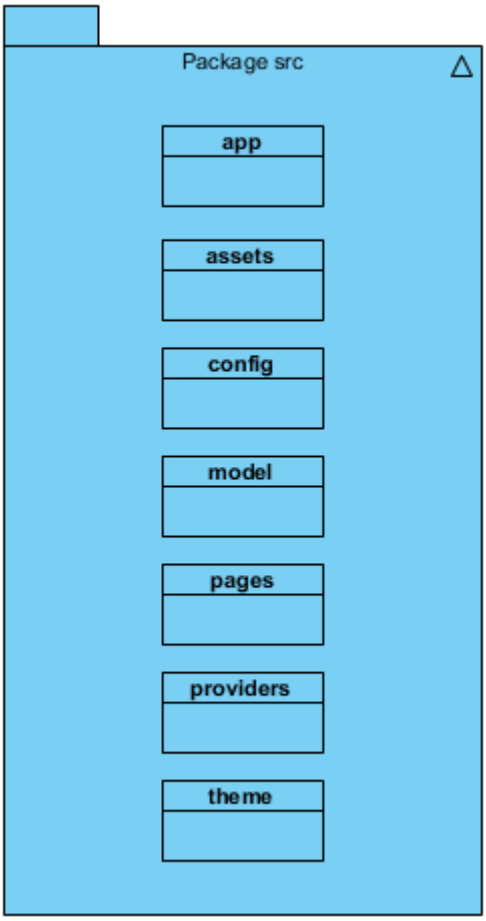
TypeScript: è un linguaggio di programmazione libero ed Open source sviluppato da Microsoft. Si tratta di un Super-set di JavaScript che basa le sue caratteristiche su ECMAScript 6

Firebase: è un potente servizio on line che permette di salvare e sincronizzare i dati elaborati da applicazioni web e mobile.

1.5 Riferimenti

- Bernd Bruegge & Allen H. Dutoit, Object-Oriented Software Engineering: Using UML, Patterns and Java, (2nd edition), Prentice-Hall, 2003
- Ian Sommerville, Software Engineering, Addison Wesely
- DDI_ODD_2.0
- https://ionicframework.com/

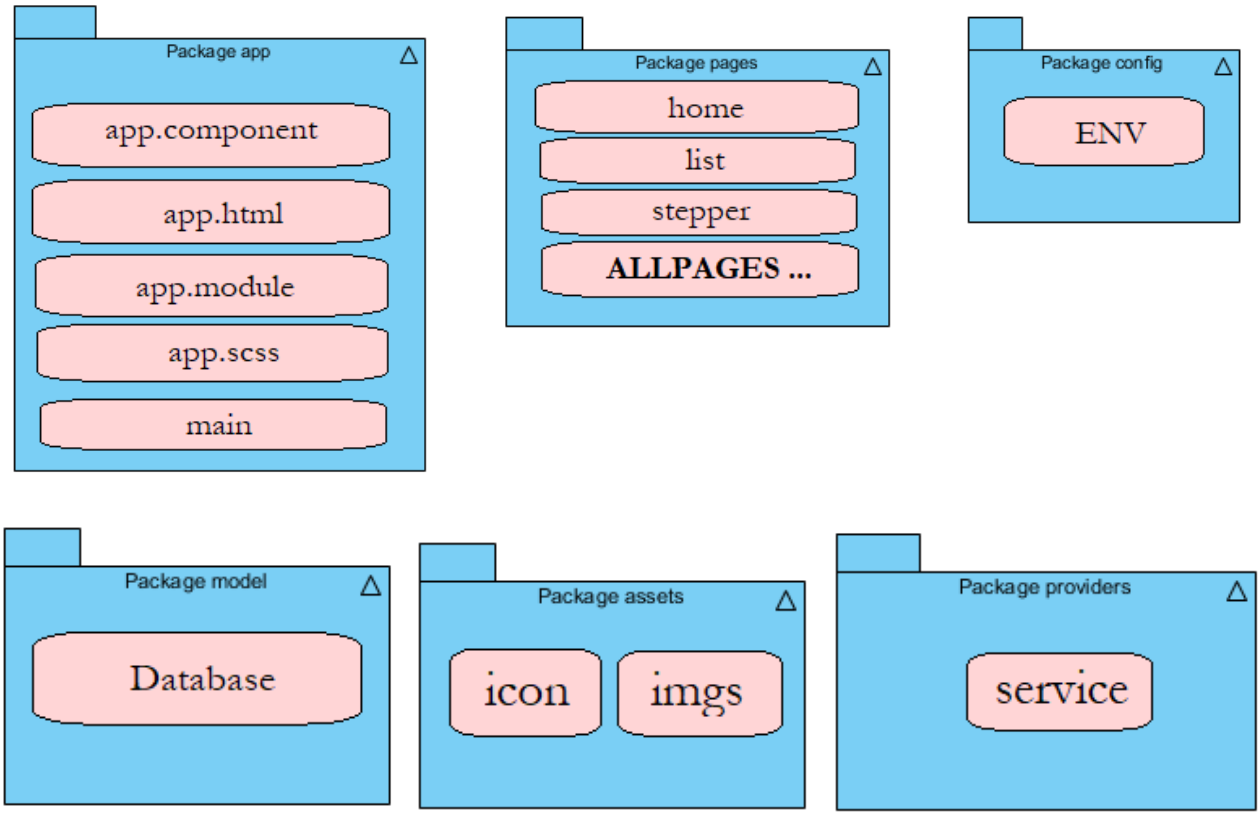
2. Packages



Come è possibile osservare dalla figura Immagine 2.2.4.1, il sistema conterrà diversi packages:

- **App:** dove il codice dell'intera applicazione viene definito. Contiene il codice di avvio dell'applicazione e il CSS globale.
- **Assets:** contiene tutti i file multimediali dell'applicazione.
- **Config:** contiene l'environment, ovvero le informazioni necessarie per il collegamento a Firestore.
- **Model:** contiene tutte le classi che permettono la manipolazione dei dati persistenti dell'applicazione.
- **Pages:** contiene le pagine dell'applicazione e ognuna di essa ha una cartella dove vi è definita la pagina (.html), lo stile (.css) e lo script utilizzato (.ts).
- **Provider:** contiene le classi dove vi è il decoratore **@Injectable**. Queste classi sono **iniettate** in altre classi per **fornire** un **servizio.**

- **Themes:** contiene i file in cui è possibile modificare il tema di default di Ionic.



3 Class Interfaces

3.1 AccountService

Nome classe	AccountService
Descrizione	Questa classe gestisce la logica applicativa degli account. Permette la registrazione e il login di un account. Permette l'attivazione della messaggistica tra studente e tutor.
Pre-condizione	**context** AccountService:: registration(String fiscalCode, String name, String surname, String dateOfBirth, String birthPlace, String email, String password, String confPassword, String userType, String sex, Boolean privacy) **pre:** (fCode != null && name != null && surname != null && dateOfBirth != null && birthPlace != null && email != null && password != null && confPassword != null && userType != null && sex != null) && privacy == true && email non presente nel DB. **context** AccountService:: login(String email, String password) **pre:** email != null && password != null && email e password corrispondono. **context** AccountService:: acceptRequestMessaging(String student, String tutor, String university) **pre:** student != null && tutor != null && university != null **context** AccountService:: sendRequestMessaging(String student, String tutor, String university) **pre:** student != null && tutor != null && university != null
Post-condizione	**context** AccountService:: acceptRequestMessaging() **post:** (tutor.studentList = tutor.studentList+1 && student.tutor = tutor && tutor.pendingList = tutor.pendingList-1) | | (tutor.pendingList = tutor.studentList-1) **context** AccountService:: sendRequestMessaging(String student, String tutor, String university) **post:** tutor.pendingList = tutor.pendingList+1
Invarianti	

Tutti i campi di registrazione hanno come precondizione di avere un corretto formato.

PendingList rappresenta la lista del tutor delle richieste di abilitazione messaggistica in sospeso.

3.2 MessaggingService

Nome classe	MessagingService
Descrizione	Questa classe gestisce la logica applicativa della messaggistica. Permette l'invio e la visualizzazione di un messaggio. Permette l'invio e la visualizzazione di un file.
Pre-condizione	**context** MessagingService:: startChat(String receveir) **pre:** receveir != null **context** MessagingService:: sendMessage(String message) **pre:** message != null && message.text > 0 && message.text <= ? **context** MessagingService:: List<Message> getAllMessages(String chatID) **pre:** chatID != null **context** MessagingService:: Message getMessage(String chatID) **pre:** chatID != null **context** MessagingService:: uploadFile(String file) **pre:** file.type = ".pdf" && file.size = 50MB **context** MessagingService:: List<File> getFiles(String sender, String receveir) **pre:** sender != null && receveir != null **context** MessagingService:: endMessage(String chatID) **pre:** chatID != null && exists(chatID) **context** MessagingService:: addNotificationMessage()
Post-condizione	
Invarianti	

CFile contiene le informazioni relative al file caricato.

Files rappresenta una collezione di file.

3.3 ReviewService

Nome classe	ReviewService
Descrizione	Questa classe gestisce la logica applicativa delle recensioni. Permette la visualizzazione, l'inserimento, la valutazione di una recensione e la ricerca tramite filtri di esse.
Pre-condizione	**context** ReviewService:: getReview(String university) **pre:** university != null **context** ReviewService:: insertReview(String text, String university) **pre:** text != null && text > 0 && text <= 1000 && university != null **context** ReviewService:: List<Review> filterReview(int star) **pre:** star != null **context** ReviewService:: rateReview(Review review, int star) **pre:** review != null && star != null && l'utente non ha ancora valutato review
Post-condizione	**context** ReviewService:: insertReview(String text, String university) **post:** reviews.university = reviews.university + 1 **context** ReviewService:: reviewsStar filterReview(int star) **post:** reviewsStar <= reviews
Invarianti	

Star è la valutazione di una review.

Reviews è una collezione di review.

ReviewsStar contiene le review che hanno come valutazione Star.

Review ha come campi testo, università, mediaValutazioni, totValutazioni, qntValutazioni. mediaValutazioni = totValutazioni / qntValutazioni.

3.4 Q&AService

Nome classe	Q&AService
Descrizione	Questa classe gestisce la logica applicativa delle Question&Answer. Permette l'inserimento di domande e di rispondere a una determinata domanda. Inoltre, ne permette la visualizzazione.
Pre-condizione	**context** Q&AManger:: insertQuestion(String title, String text) **pre:** text != null && text > 0 && text <= 200 && title != null && title > 0 && title <= 40 **context** Q&AManger:: insertAnswer(question, String answer) **pre:** question != null && answer > 0 && answer <= 500 **context** Q&AManger:: List<Question> getQuestions() **pre:** questions != 0 **context** Q&AManger:: List<Answer> getAnswers(Question question) **pre:** question.answers != 0
Post-condizione	**context** Q&AManger:: insertQuestion(String title, String text) **post:** questions = questions + 1 **context** Q&AManger:: insertAnswer(Question question, String answer) **post:** question.answers = question.answers +1
Invarianti	

Question rappresenta una domanda.

Answer rappresenta una risposta relativa a una domanda.

Questions rappresenta una collezione di question.

Answers rappresenta una collezione di answer.

3.5 StepService

Nome classe	StepService
Descrizione	Questa classe gestisce la logica applicativa della guida. Permette il caricamento dello step di interesse.
Pre-condizione	**context** StepService:: Step getStep(int stepNumber) **pre:** numeroStep > 1 && numeroStep <= 6
Post-condizione	
Invarianti	

Step rappresenta le informazioni relative allo step indicato da stepNumber

3.6 NewsService

Nome classe	NewsService
Descrizione	Questa classe gestisce la logica applicativa delle news. Permette l'inserimento e la visualizzazione delle news.
Pre-condizione	**context** NewsService:: insertNews(String titolo, String descrizione) **pre:** titolo != null && descrizione != null && titolo > 0 && titolo <= 80 && descrizione > 0 && descrizione <= 1000 **context** NewsService:: List<News> getNews() **pre:** **context** NewsService:: addNotificationNews()
Post-condizione	**context** NewsService:: List<News> insertNews(String titolo, String descrizione) **post:** newsList = newsList+1
Invarianti	

NewsList rappresenta una lista di news.

4 Design Pattern con class diagram

4.1 Bridge pattern

Nome:

Bridge (Handle/Body)

Descrizione del problema:

Erasmus Smart fa uso di un database realtime NOSQL (Firebase), che in futuro potrebbe diventare obsoleto o poco manutenibile e richiedere quindi il cambiamento del database.

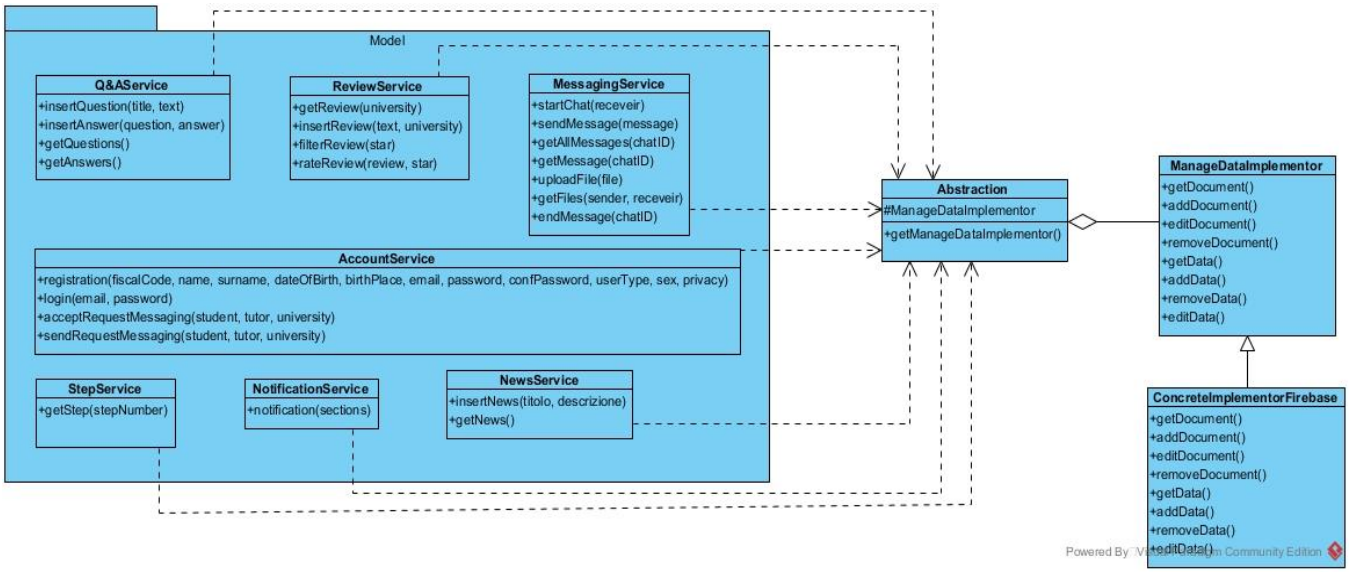
Separare quindi l'astrazione dall'implementazione così che una diversa implementazione possa essere sostituita in futuro, è il motivo che ci spinge ad utilizzare questo pattern.

Conseguenze:

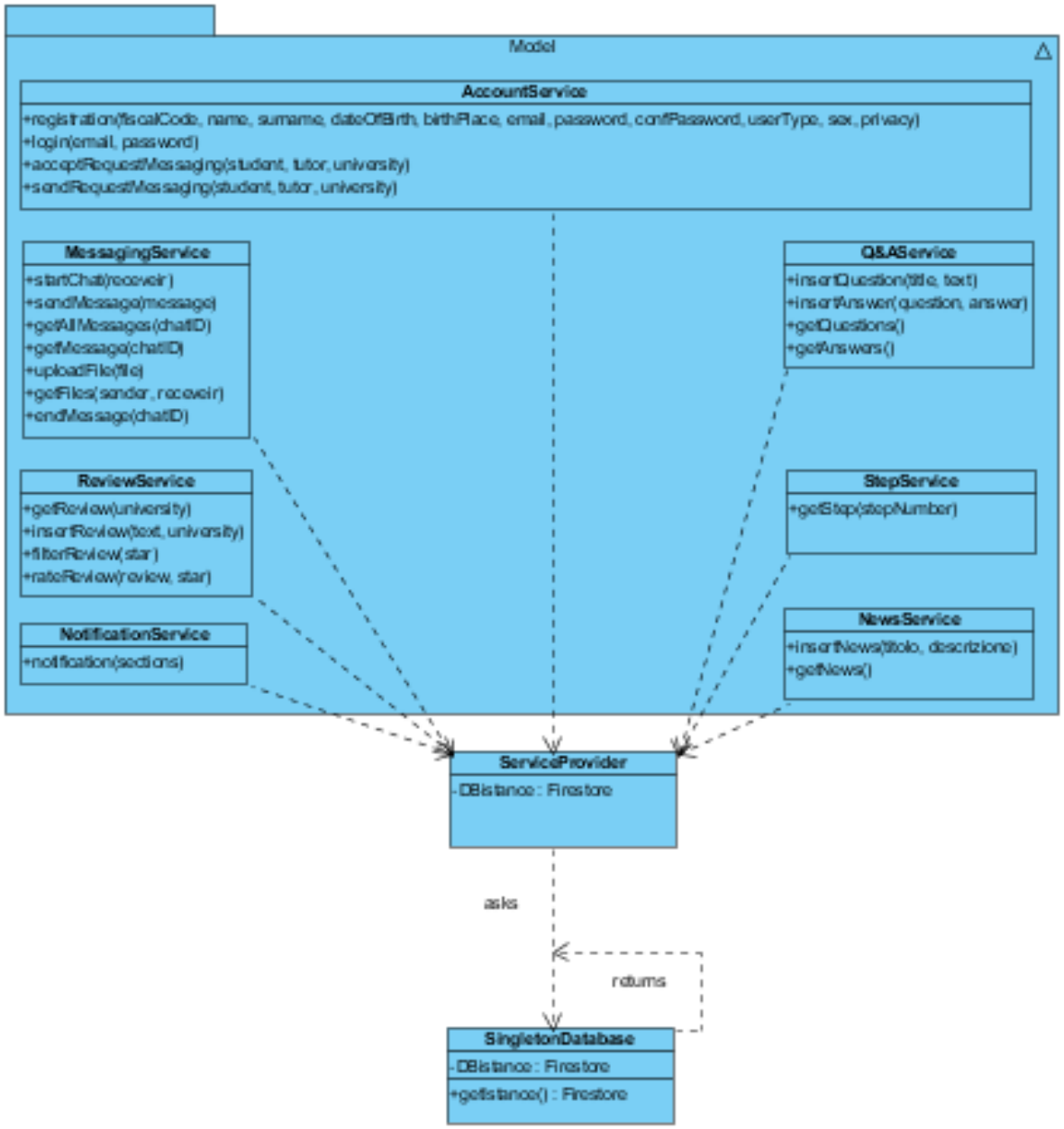
Disaccoppiamento tra interfaccia ed implementazione per rendere veloci e poco laboriosi i futuri cambiamenti al sistema.

L'implementazione non è più legata in modo permanente ad un'interfaccia.

La parte di alto livello del sistema conosce soltanto le classi Abstraction e ManageDataImplementor (Implementor).



4.2 Singleton Pattern



ES fa uso del singleton pattern per implementare la connessione a Firestore.

Singleton Pattern

- **Descrizione:** Definisce una singola classe che è responsabile di creare un oggetto assicurandosi che questa operazione non venga ripetuta più di una volta nel ciclo di vita dell'applicazione.
- **Problema:** Si ha l'esigenza di non istanziare la variabile di tipo Firestore per ogni altra classe che la utilizza, ma di crearne solo una che venga utilizzata in tutta l'applicazione.
- **Soluzione:** Per ottenere un siffatto comportamento è necessario avvalersi dello specificatore di accesso «private» anche per il costruttore della classe (cosa che generalmente non viene mai praticata in una classe "standard") ed utilizzare un metodo statico che consenta di accedere all'unica istanza della classe, quindi basterà implementare la seguente porzione di codice e far si che tutte le classi del programma utilizzino SingletonDatabase.getIstance(); per accedere alla variabile DBinstance e quindi a Firestore.

Esempio:

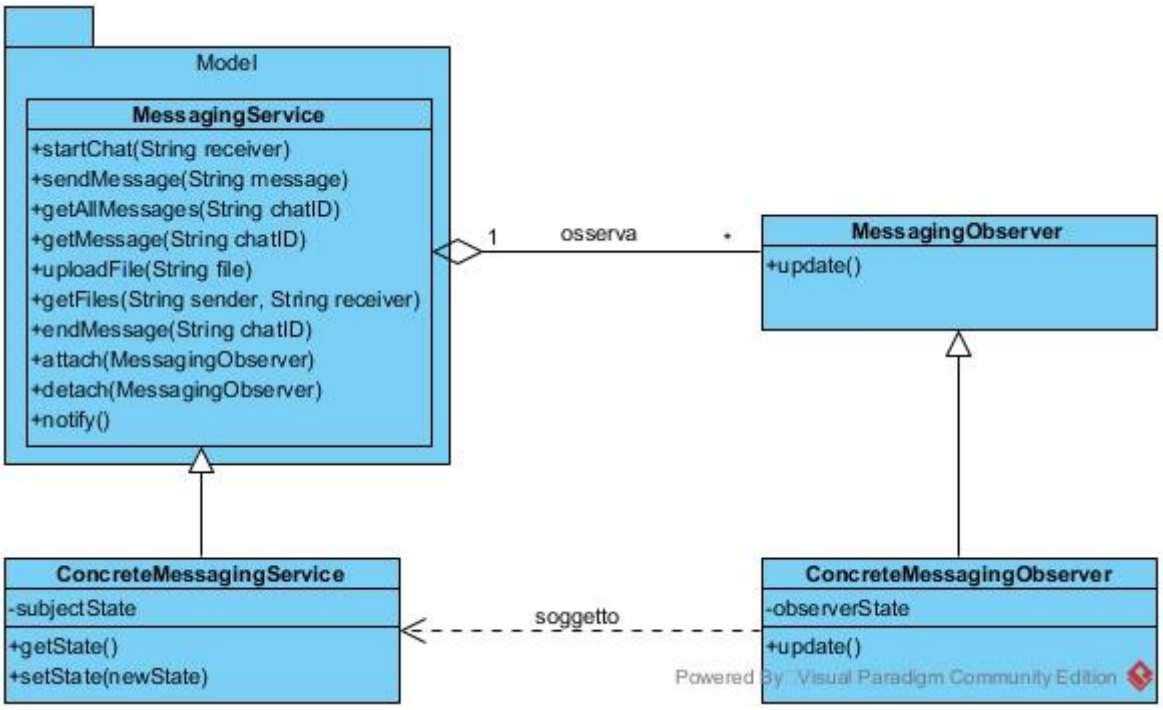
```
export class SingletonDatabase {
    private static DBistance: firebase.firestore.Firestore;
    private constructor() {
        SingletonDatabase.getInstance();
    }

    static getInstance(): firebase.firestore.Firestore {

        if (this.DBistance == undefined) {
            return firebase.firestore()
        } else {
            return this.DBistance
        }

    }
}
```

4.3 Observer Pattern:



ES fa uso dell'Observer Pattern per mantenere consistenza tra i diversi stati ridondanti della messaggistica tra tutor-studente, come un vero e proprio "sincronizzatore", notificando gli interessati qualora dovesse cambiare lo stato della chat.

Observer Pattern

- **Descrizione:** Definisce una dipendenza 1-N tra oggetti in modo tale che quando cambia lo stato di un oggetto, tutte le sue dipendenze sono notificate ed aggiornate automaticamente.
- **Problema:** Si ha l'esigenza di mantenere un alto livello di consistenza fra classi correlate, senza produrre situazioni di forte dipendenza e di accoppiamento elevato.
- **Soluzione:** La soluzione si compone di quattro elementi:
  - *Subject:* ha conoscenza dei propri Observer e fornisce operazioni per la notifica agli Observer;
  - *Observer:* specifica un'interfaccia per la notifica di eventi agli oggetti interessati in un Subject;
  - *ConcreteSubject:* mantiene lo stato del soggetto osservato e notifica gli observer in caso di un cambio di stato ed invoca le operazioni di notifica ereditate dal Subject, quano devono essere informati i ConcreteObserver;
  - *ConcreteObserver:* implementa l'interfaccia dell'Observer definendo il comportamento in caso di cambio di stato del soggetto osservato.

5 Glossario

Termine	Spiegazione
Design Pattern	È una soluzione progettuale generale ad una problematica ricorrente.