UNIVERSITÀ
DI TRENTO

TN2night

02/12/2022

D3-T02

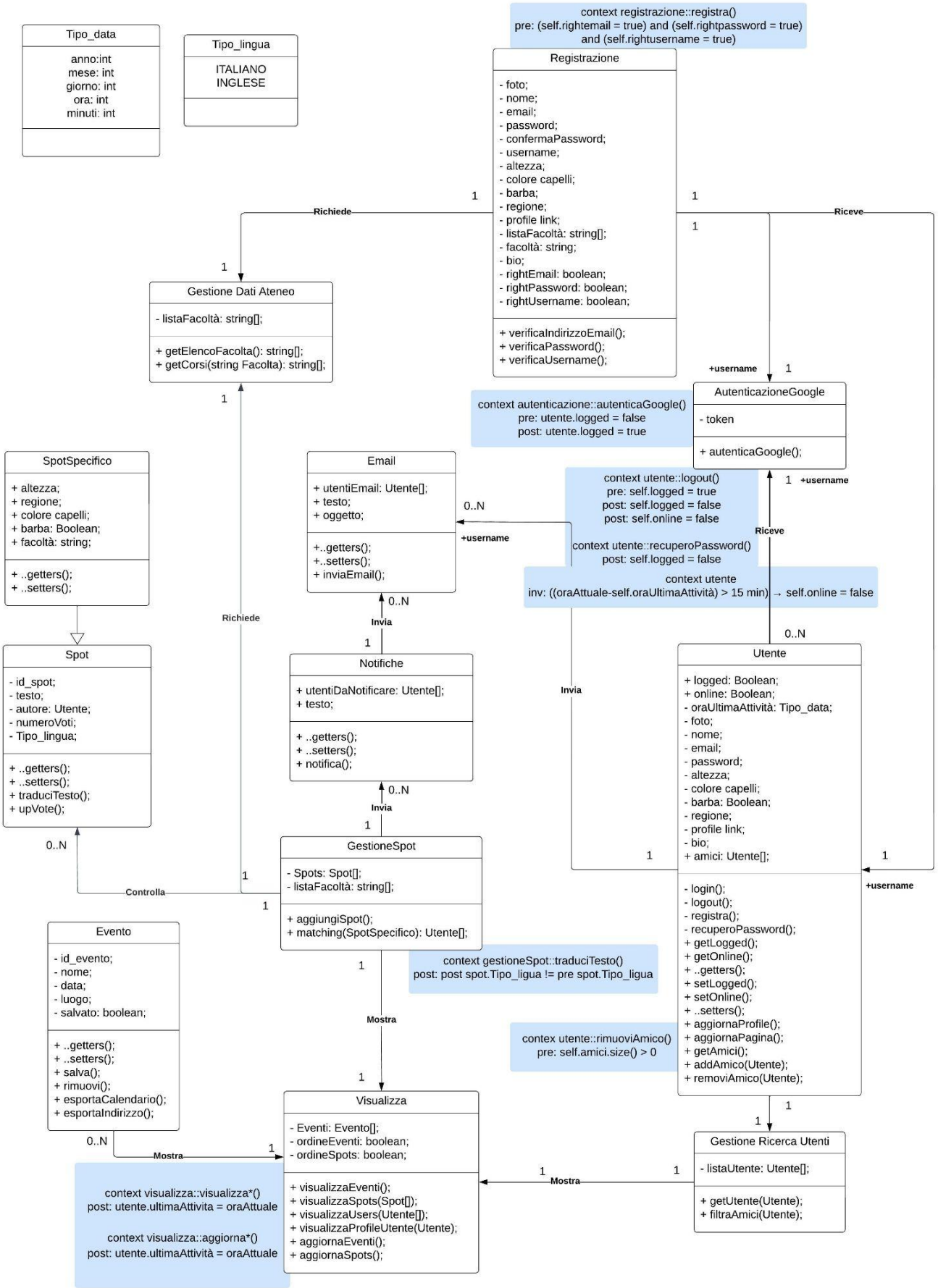
Indice

Scopo del Documento

Il presente documento riporta la definizione dell'architettura del progetto TN2night. Verrà utilizzato un diagramma delle classi in Unified Modeling Language (UML) e del codice in Object Constraint Language (OCL) per esprimere in modo formale e privo di ambiguità le regole che vengono applicate al diagramma in UML. Nel precedente documento era stata progettata l'architettura del progetto mediante diagrammi di contesto e dei componenti. In questo documento, invece, facendo riferimento alla progettazione fatta, viene definita l'architettura del sistema elencando e descrivendo le varie classi e interfacce che dovranno essere implementate, con anche i vari collegamenti tra di esse. Inoltre, verrà descritta la logica che regola il comportamento del software con l'utilizzo di OCL per tali concetti e regole che non possono in alcun modo essere espressi all'interno del Class Diagram.

Diagramma delle classi

Nel presente capitolo vengono elencate e descritte le varie classi previste nel progetto TN2night. In riferimento al documento di progettazione dell'architettura, ogni sistema esterno e attore presente nel diagramma di contesto, e ogni componente presente nel diagramma dei componenti diverranno una o più classi in questo documento di definizione dell'architettura. Le varie classi potranno anche eventualmente essere associate ad altre descrivendo, se necessario, informazioni aggiuntive per quanto riguarda la relazione tra di esse. Si riportano ora le varie classi individuate a partire dai diagrammi di progettazione dell'architettura presenti nel precedente documento. In questa procedura si è tentato di massimizzare il livello di coesione tra le varie classi anche se a volte però si è dovuto fare qualche sacrificio per l'impossibilità di rappresentare in UML determinati concetti. Si è quindi preferito procedere con un'accurata descrizione delle scelte di definizione dell'architettura per poi descrivere in codice OCL i vari vincoli che vertono nelle e tra le classi descritte. Per le varie classi si è anche deciso di fornire una breve descrizione di alcune funzioni e attributi nel caso in cui il nome non fosse autoesplicativo.


Tipo_data
anno:int
mese: int
giorno: int
ora: int
minuti: int
Tipo_lingua
ITALIANO
INGLESE
context registrazione::registra()
pre: (self.rightemail = true) and (self.rightpassword = true)
and (self.rightusername = true)
Registrazione
- foto;
- nome;
- email;
- password;
- confermaPassword;
- username;
- altezza;
- colore capelli;
- barba;
- regione;
- profile link;
- listaFacoltà: string[];
- facoltà: string;
- bio;
- rightEmail: boolean;
- rightPassword: boolean;
- rightUsername: boolean;
+ verificaIndirizzoEmail();
+ verificaPassword();
+ verificaUsername();
Richiede
Riceve
Gestione Dati Ateneo
- listaFacoltà: string[];
+ getElencoFacolta(): string[];
+ getCorsi(string Facolta): string[];
+username
AutenticazioneGoogle
- token
+ autenticaGoogle();
context autenticazione::autenticaGoogle()
pre: utente.logged = false
post: utente.logged = true
SpotSpecifico
+ altezza;
+ regione;
+ colore capelli;
+ barba: Boolean;
+ facoltà: string;
+ ..getters();
+ ..setters();
Email
+ utentiEmail: Utente[];
+ testo;
+ oggetto;
+..getters();
+..setters();
+ inviaEmail();
context utente::logout()
pre: self.logged = true
post: self.logged = false
post: self.online = false
context utente::recuperoPassword()
post: self.logged = false
context utente
inv: ((oraAttuale-self.oraUltimaAttività) > 15 min) → self.online = false
Richiede
Invia
Spot
- id_spot;
- testo;
- autore: Utente;
- numeroVoti;
- Tipo_lingua;
+ ..getters();
+ ..setters();
+ traduciTesto();
+ upVote();
Notifiche
+ utentiDaNotificare: Utente[];
+ testo;
+ ..getters();
+ ..setters();
+ notifica();
Utente
+ logged: Boolean;
+ online: Boolean;
- oraUltimaAttività: Tipo_data;
- foto;
- nome;
- email;
- password;
- altezza;
- colore capelli;
- barba: Boolean;
- regione;
- profile link;
- bio;
+ amici: Utente[];
- login();
- logout();
- registra();
- recuperoPassword();
+ getLogged();
+ getOnline();
+ ..getters();
+ setLogged();
+ setOnline();
+ ..setters();
+ aggiornaProfile();
+ aggiornaPagina();
+ getAmici();
+ addAmico(Utente);
+ removiAmico(Utente);
Controlla
GestioneSpot
- Spots: Spot[];
- listaFacoltà: string[];
+ aggiungiSpot();
+ matching(SpotSpecifico): Utente[];
context gestioneSpot::traduciTesto()
post: post spot.Tipo_ligua != pre spot.Tipo_ligua
Evento
- id_evento;
- nome;
- data;
- luogo;
- salvato: boolean;
+ ..getters();
+ ..setters();
+ salva();
+ rimuovi();
+ esportaCalendario();
+ esportaIndirizzo();
Mostra
Visualizza
- Eventi: Evento[];
- ordineEventi: boolean;
- ordineSpots: boolean;
+ visualizzaEventi();
+ visualizzaSpots(Spot[]);
+ visualizzaUsers(Utente[]);
+ visualizzaProfileUtente(Utente);
+ aggiornaEventi();
+ aggiornaSpots();
contex utente::rimuoviAmico()
pre: self.amici.size() > 0
Gestione Ricerca Utenti
- listaUtente: Utente[];
+ getUtente(Utente);
+ filtraAmici(Utente);
context visualizza::visualizza*()
post: utente.ultimaAttivita = oraAttuale
context visualizza::aggiorna*()
post: utente.ultimaAttività = oraAttuale

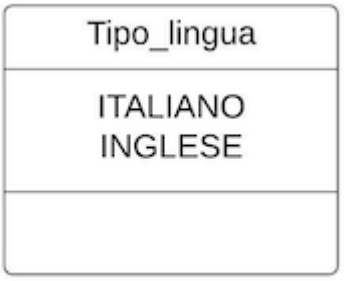
Descrizione del diagramma

Tipi di dato

Data

Abbiamo individuato un tipo di dato che va a salvare la data. Questa classe contiene gli attributi giorno, mese, anno, ora, minuti (GG/MM/AAAA hh:mm). La suddetta classe viene utilizzata per mantenere in ordine cronologico gli eventi mostrati nella gli spot nella *Spot Page.*

Lingua



Abbiamo individuato un tipo di dato che va a salvare la lingua degli spot. In questo caso si è deciso di optare per una semplice enumerazione con solo la lingua italiana e inglese. Questo tipo di dato verrà utilizzato dall'applicazione per decidere gli spot

Classi

Spot

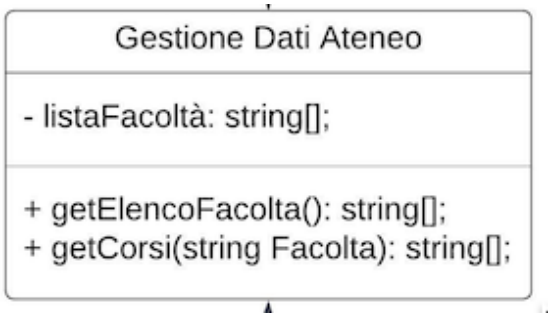
Le funzioni *getters()* vengono utilizzate dalla classe *CreazioneSpot*. In questo caso le informazioni recuperate servervono per ricordurre lo spot all'autore e quindi mettere in contatto quest'ultimo con la persona cercata. Negli altri casi il *testo* sarà utilizzato come body nella *Spot Page* e nelle *Notifiche*. L'attributo *Tipo_lingua* viene utilizzato per sapere la lingua originale dello spot *(ITA/EN)* e in quale lingua eventualmente tradurlo. la classe ha un attributo autore che viene utilizzato per eventualmente contattare l'autore dello spot. Infine, il *numeroVoti* è necessario per sapere l'ordine di visualizzazione degli spots. All'intero di questo componente sarà anche possibile invocare la funzione *traduciTesto()* che in base alla lingua di partenza di uno *Spot* ritornerà la stringa tradotta nel *Tipo_lingua* opposto. *upVote()* permette all'utente registrato di esprimere le proprie preferenze per gli *Spot* più popolari.

Spot Specifico

A livello gerarchico, questa classe si trova ad un livello inferiore rispetto alla precedente *(Spot)*. Ciò detto, la classe *Spot Specifico* eredita i metodi della classe padre. Oltre il testo, tale classe permette di reperire le informazioni specifiche dello spot.

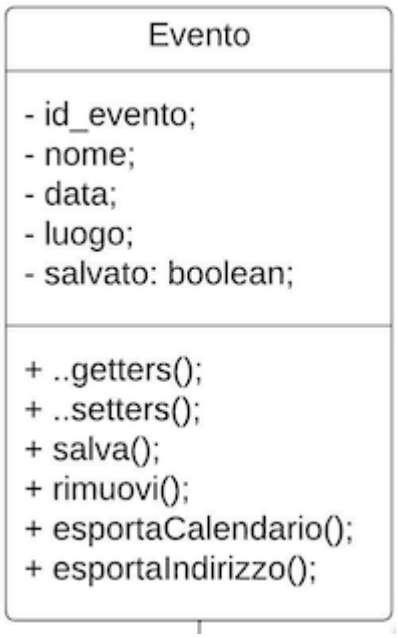
Gestione Dati Ateneo

Questa classe è utilizzata sia dalla classe *Registrazione* che dalla *Gestione Spot*. I metodi presenti in questa classe hanno il fine di reperire i dati corretti inerenti alle facoltà di studio dell'Università di Trento. Un utente potrà quindi scegliere da un menu dropdown le informazioni corrette ed aggiornate.



Evento

Come suggerisce il nome, la classe *Evento* racchiude le informazioni riguardanti tali fenomeni. Le funzioni *getters* verranno utilizzate sia dalla classe *Gestione Eventi* e, di nuovo, dalla classe *Spot Specifico*. Questa componente fornisce delle funzioni per esportare l'indirizzo dove avverrà un dato fenomeno e la possibilità di creare un evento nel proprio calendario personale, otre che fornire la possibilità ad un utente di *salvare* o *rimuovere* da una lista personale determinati eventi considerati come *preferiti* (*attribute: - salvato: Boolean*). Tutti gli eventi gestiti da questa classe verranno poi utilizzati all'interno di *Visualizza*, ed in particolare dal metodo *visualizzaEventi*.

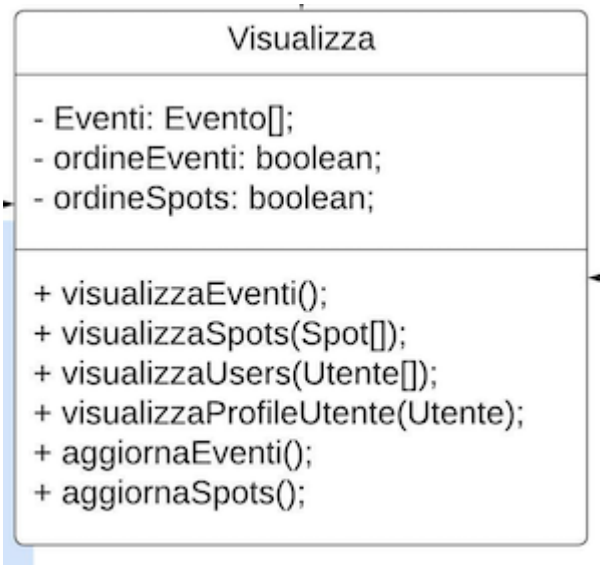


Gestione Ricerca Utenti

Questa classe ha il compito di gestire gli altri utenti: permette all'utente loggato di aggiungere ai propri amici le persone mostrate nella schermata *Users Page*. Il metodo *getUtente* è responsabile della ricerca tramite apposita barra, ritornerà una lista di utenti che corrispondono al nome cercato. La funzione *filtraAmici* restituisce sempre una lista di utenti che sono stati precedentemente salvati come

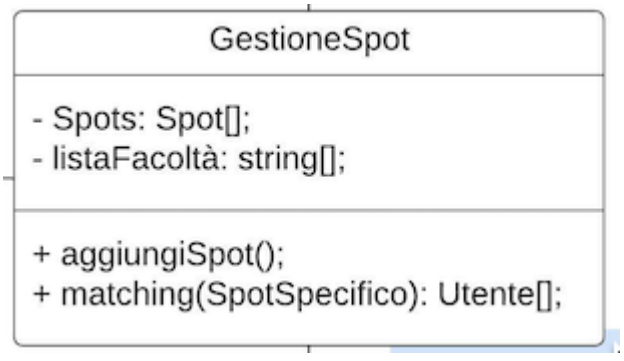
amici e che quindi verranno mostrati grazie al metodo *visualizzaUsers* nella classe *Visualizza*.

Visualizza



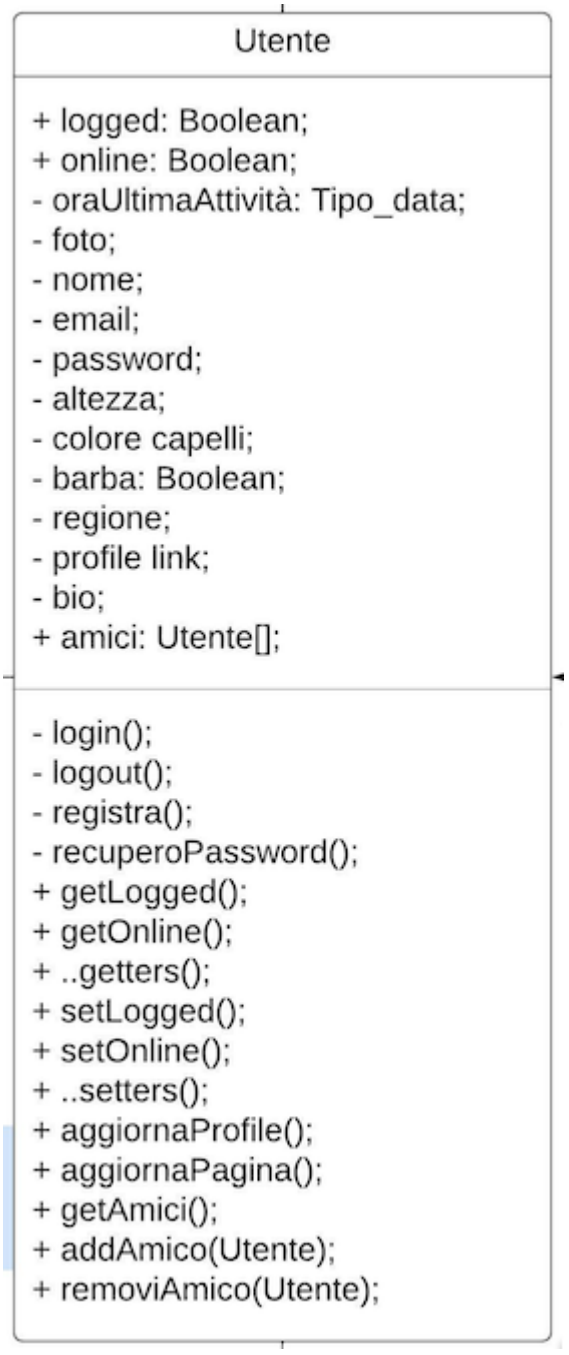
All'interno di questa classe sono contenuti tutti i metodi responsabili della visualizzazione effettiva dei dati: ogni metodo si occuperà di mostrare a video le informazioni inerenti alla pagina sulla quale la funzione viene chiamata. Per *visualizzaEventi* e *visualizzaSpots*, i metodi accettano un valore booleano per impostare l'ordine di visualizzazione in base alle preferenze espresse dall'utente. Le funzioni *aggiorna**, come suggerisce il nome, servono ad aggiornare la schermata rispettiva o per la creazione di un nuovo oggetto o sotto richiesta dell'user.

Gestione Spot



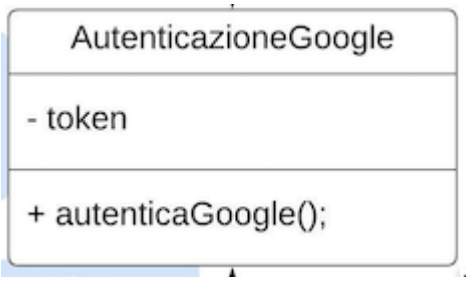
Questa classe permette di aggiungere lo spot alla lista già presente e mostrata a schermo, oltre che salvata nel database, tramite il metodo *aggiungiSpot()*. Nel caso in cui venga creato uno spot specifico verrà invocata la funzione *matching()* che restituirà una lista di *utente* che hanno un certo valore di corrispondenza con lo spot creato. La lista di questi utenti viene salvata in *Notifiche.utentiDaNotificare*.

Utente



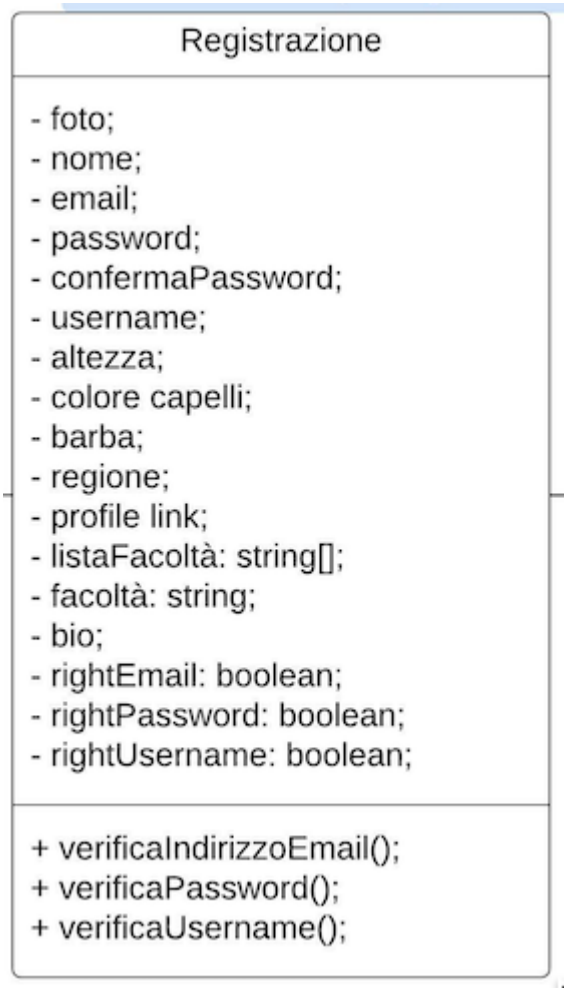
Visionando l'analisi del contesto dello scorso documento (*Deliverable 2*) è possibile identificare tre attori che si interfacceranno con il sistema: *Utente registrato*, *Utente Anonimo.* È stato deciso di raggruppare il tutto in un'unica classe *Utente* che rappresenta i due attori. Infatti tale componente ha la possibilità di effettuare il *login()* e *logout()* dall'applicazione. Sono disponibili i metodi *getters()* e *setters()* specificate nella classe per ottenere le informazioni necessarie. Tra i vari attributi sono state inseriti *logged* e *online,* entrambi booleani, per ottenere lo stato dell'utente all'interno dell' app. Il metodo *getAmici()* modifica l'attributo delle lista di amici dell'utente registrato. Questa lista viene utilizzato nella visualizzazione di un evento nella *home page* per segnalare gli amici che parteciperanno ad un determinato fenomeno. *addAmico()* e *removeAmico()* servono per gestire le relazione tra utenti all'interno dell'applicazione. Il metodo *recuperoPassword()* ha lo scopo di generare e fornire una nuova password all'utente. Una volta che l'utente effettuerà l'accesso verrà settato online lo stato di attività dell'utente. Il metodo *registra()* serve all'applicazione per salvare l'utente nel database.

Autenticazione



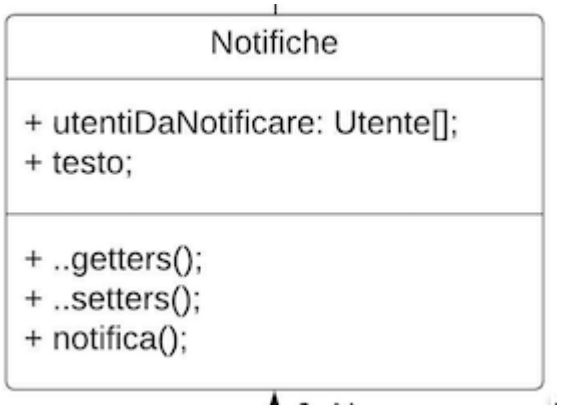
Analizzando il diagramma dei componenti abbiamo ritenuto necessario la creazione di una classe che si interfacciasse con gli altri componenti che si occupavano della registrazione e dell'accesso dell'utente. La classe *Autenticazione* ha un metodo che serve ad effettuare l'autenticazione con il sistema fornito da Google. È presente l'attributo *token* in quanto viene utilizzato dall'api di *Google Sign In*.

Registrazione



Analizzando il documento precedente (*Deliverable 2*) e il diagramma dei componenti è stato deciso di realizzare la classe *Registrazione* la quale ha come attributi le informazioni richieste all'utente in fase di, appunto, registrazione. Tali dati verranno richiesti in maniera pratica attraverso una form. Il metodo *verificaIndirizzoEmail()* è utilizzato per verificare che l'indirizzo email sia valido, *verificaPassword()* controlla che l'utente abbia inserito le password correttamente seguenedo i requisiti specificati in fase di progettazione, e infine *verificaUsername()* controlla che l'username inserito sia unico. Le ultime tre funzioni appena citate andranno a salvare l'esito della verifica in rispettive variabili booleane. Le informazioni prelevate dalla classe *Gestione Dati Ateneo* sono salvate nella lista di stringhe *listaFacoltà,* verranno mostrate all'utente sotto forma di *dropdown* menù.

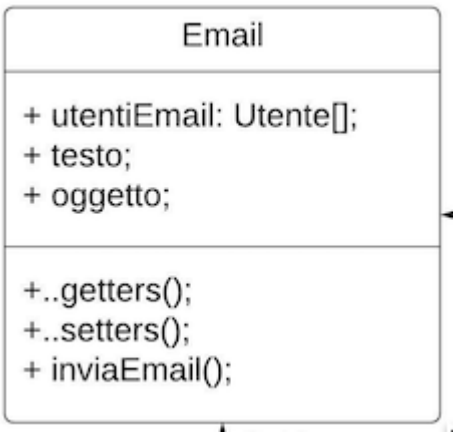
Notifiche



Analizzando il diagramma dei componenti, risulta che l'applicazione comunica all'utente diverse informazioni tramite delle notifiche. Gli attributi presenti in questa classe sono necessari ad individuare chi e cosa deve notificare, i metodi servono a modificare i valori degli attributi e ad inviare l'effettiva notifica. La lista di utenti da notificare (*self.utentiDaNotificare*) è aggiornata dalla metodo *GestioneSpot.matching()*.

Email

Vista la natura dell'applicazione e come quest'ultima fornisce informazioni all'utente, è stato deciso di rappresentare una classe dedicata per l'invio di email all'utente. Questo componente contiene gli attributi necessari per poter comporre un email, con i vari metodi *setters()* per poterli modificare ed un metodo dedicato all'invio dell'email. Come nel caso precedente, la lista di utenti da notificare (*self.utentiEmail*) è aggiornata dalla metodo *GestioneSpot.matching()*.



OCL: Object Constraint Language

In questa sezione descriveremo la logica prevista in alcune operazioni di certe classi. Questa logica verrà descritta in Object
Constraint Language (OCL) e ricorriamo a questo strumento dal momento che non tutti i concetti sono esprimibili in modo
formale in UML.

Registrazione

Un utente per potersi registrare è necessario che abbia passato i controlli sull'email, password ed username:

```
context registrazione::registra()
pre: (self.rightemail = true) and (self.rightpassword = true)
and (self.rightusername = true)
```

Autenticazione

Ogni qualvolta un utente decide di effettuare il login con google è necessario che lui non risulti già loggato all'interno dell'applicazione

```
context autenticazione::autenticaGoogle()
pre: utente.logged = false
post: utente.logged = true
```

Utente

Ogni volta che un utente effettua l'azione di logout viene modificato lo status logged e lo status online diventa offline.

```
context utente::logout()
pre: self.logged = true
post: self.logged = false
post: self.online = false
```

Ogni volta che l'utente desidera recuperare la password è necessario assicurarsi che l'utente non sia loggato nell'applicazione

```
context utente::recuperoPassword()
post: self.logged = false
```

Ogni volta che l'utente risulterà inattivo per più di 15 minuti, il suo stato verrà modificato da online ad offline

```
context utente
inv: ((oraattuale-self.oraUltimaAttività) > 15 min) →
self.online = false
```

Prima che un utente elimini un amico è necessario assicurarsi che quest'ultimo abbia almeno un amico

```
contex utente::rimuoviAmico()
pre: self.amici.size() > 0
```

Spot

Quando avviene la traduzione di uno spot la sua lingua viene cambiata da italiano ad inglese o viceversa

```
context spot::traduciTesto()
post: post self.Tipo_ligua != pre self.Tipo_ligua
```

Visualizza

Quando viene effettuata una una qualsiasi operazione di visualizzazione o di aggiornamento, lo status online dell' utente viene riportato a true e viene aggiornata l'ultima oraUltimaAttivita

```
context visualizza::visualizza*()
post: utente.ultimaAttivita= oraAttuale

context visualizza::aggiorna*()
post: utente.ultimaAttivita= oraAttuale
```