UNIVERSITY OF TRENTO

Dipartimento di Ingegneria e Scienza dell'Informazione

CORSO DI INGEGNERIA DEL SOFTWARE

Documento di progetto: Documento di Architettura

Gruppo: T33

Indice

1. Scopo del documento 3

2. Diagramma delle classi 3

2.1 Tipi di dato 3

2.2 Classi funzionali 6

2.3 Classi individuate dal diagramma di contesto e dei componenti 8

2.4 Diagramma delle classi complessivo 18

3. Codice in Object Constraint Language 19

3.1 Diagramma delle classi con OCL complessivo 28

4. Note ed eventuali 29

1. Scopo del documento

Il presente documento riporta la definizione dell'architettura del progetto *StayBusy* attraverso l'utilizzo del diagramma delle classi in *Unified Modeling Language* (UML) e del codice in *Object Constraint Language* (OCL), al fine di esprimere in modo formale e privo di ambiguità le regole che vengono applicate al diagramma UML e per descrivere anche la logica del software. Di seguito viene dunque rappresentata l'architettura del sistema mediante la descrizione delle classi e delle interfacce che dovranno essere implementate e dei collegamenti sussistenti tra di esse.

2. Diagramma delle classi

Nel presente capitolo vengono elencate e descritte le varie classi previste nel progetto *StayBusy*. In particolare, ogni attore e sistema esterno presenti nel diagramma di contesto, oltre che ad ogni componente presente nel diagramma dei componenti, verranno ora rappresentati attraverso l'utilizzo di una o più classi, eventualmente associate tra loro. In questo caso, se necessario, sono state inserite anche altre informazioni aggiuntive, al fine di rappresentare al meglio le relazioni tra di esse.

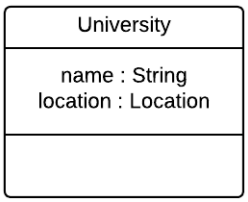
Nello specifico, di seguito vengono riportate le classi individuate dai diagrammi di contesto e delle componenti rappresentati nel documento **D1: Analisi dei requisiti**. Ognuna di queste è accompagnata da una breve descrizione di alcuni attributi e funzioni e, in alcuni casi, come si potrà vedere nel capitolo successivo, dal codice OCL, in modo da poter rappresentare e descrivere i vincoli che sono presenti tra classi descritte.

2.1 Tipi di dato

Di seguito una breve descrizione dei tipi di dato implementati dal sistema.

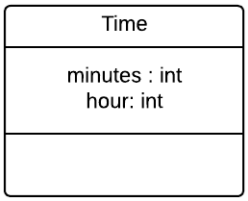
2.1.1 University

Durante la creazione del proprio account personale, l'utente deve inserire il nome della sede universitaria frequentata. Per questo motivo è stato creato un tipo di dato **University** caratterizzato da due attributi, ovvero dal *nome* dell'università di appartenenza e dalla sua *ubicazione*. Questo tipo di dato viene quindi utilizzato dal sistema per verificare l'effettiva esistenza dell'Ateneo indicato e per indirizzare lo studente alla pagina di accesso della propria università.



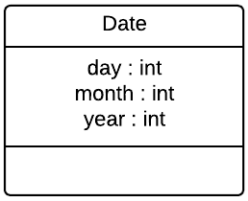
2.1.2 Time

Il tipo di dato **Time** è stato creato per il salvataggio del tempo, espresso dagli attributi *minutes* e *hours*. **Time** viene utilizzato dal sistema per esprimere l'orario di inizio e fine di un servizio. Nello specifico, questo è applicato sia nella classe **Disponibility** – in modo da salvare l'orario di disponibilità che uno studente imposta al momento della creazione dell'account e che può modificare in ogni istante – che nella classe **Announcement** – al fine di rappresentare l'orario di richiesta per un servizio specificato dall'utente offerente al momento di creazione dell'annuncio.



2.1.3 Date

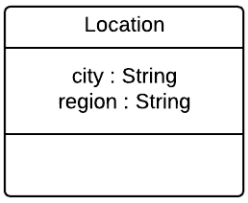
Il tipo di dato **Date** è stato creato per il salvataggio della data, rappresentata attraverso gli attributi *day*, *month* e *year*. Così come per il tipo di dato **Time**, **Date** viene utilizzato nella classe **Disponibility** – in modo da salvare i giorni di disponibilità dell'utente studente per lo svolgimento di un servizio – e nella classe **Announcement** – al fine di rappresentare la data in cui viene richiesto un servizio, specificata dall'utente offerente al momento di creazione dell'annuncio.



È stato deciso di non inserire direttamente un attributo *ora* di tipo **Time** all'interno del tipo di dato **Date** poiché l'orario di disponibilità e di richiesta è composto da due attributi: *StartTime* – ovvero orario di inizio di un servizio – e *EndTime* – ovvero l'orario di conclusione di un servizio - i quali vengono espressi direttamente nelle classi **Disponibility** e **Announcement**.

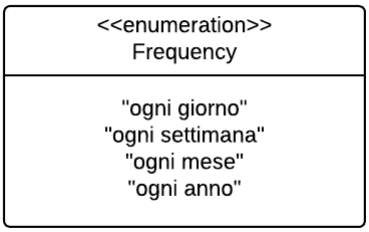
2.1.4 Location

Il tipo di dato **Location** è stato creato per permettere il salvataggio di un luogo ed è costituito da due attributi, ovvero dal nome della città e dalla sua regione di appartenenza. **Location** viene utilizzato sia dal tipo di dato **University** - per poter rappresentare l'ubicazione dell'Ateneo - che dalla classe **Announcement** - in modo da permettere il salvataggio del luogo in cui dovrà essere svolto il servizio richiesto.



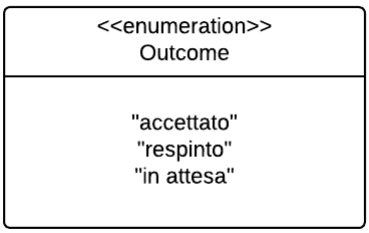
2.1.5 Frequency

Il tipo di dato **Frequency** è stato creato al fine di semplificare il processo di salvataggio delle disponibilità dell'utente studente. Questo tipo è un'enumerazione con quattro valori. In questo caso è stato preferito l'utilizzo di un'enumerazione data la sua chiarezza espositiva e la maggior agevolazione della modifica della classe nel caso si avesse intenzione di modificare/aggiungere attributi. In particolare, **Frequency** verrà utilizzato per permettere all'utente studente di non dover inserire le proprie disponibilità giorno per giorno, bensì di poter impostare questo attributo che gli consente direttamente di selezionare il tipo di frequenza che il sistema deve utilizzare per salvare le sue fasce orarie. Per questo motivo, questo tipo di dato è utilizzato nella classe **Availability**.



2.1.6 Outcome

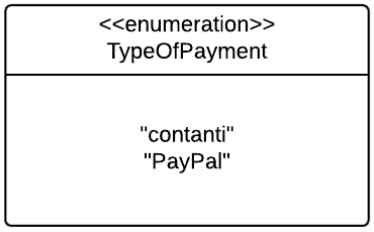
Il tipo di dato **Outcome** è stato creato per gestire lo stato delle richieste degli annunci. Questo tipo è un'enumerazione di tre valori - "Accettato", "Respinto", "In attesa" – attraverso cui l'utente può sapere qual è lo stato di una richiesta da lui effettuata per uno specifico servizio. Per questo motivo, questo tipo di dato è utilizzato nella classe **Candidate**.



2.1.7 TypeOfPayment

Il tipo di dato **TypeOfPayment** è stato creato per salvare la tipologia di pagamento che verrà effettuato al termine di un servizio. Questo tipo è un'enumerazione con due valori che rappresentano i tipi di

pagamento che vengono forniti dall'applicazione. **TypeOfPayment** viene dunque utilizzato dalla classe **Payment** che si occupa della gestione dei pagamenti.



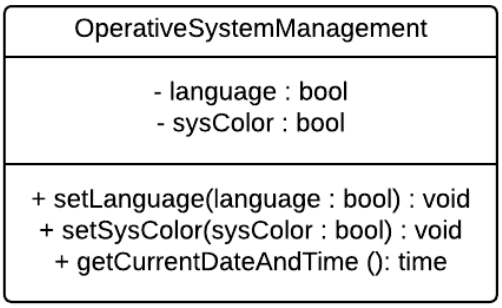
2.2 Classi funzionali

Di seguito vengono presentate le classi che hanno la funzione di raccogliere e raggruppare i metodi riguardanti delle funzionalità cardine utilizzate da più classi. Queste classi sono astratte e quindi tutte le loro funzioni sono statiche dato che una classe astratta non può essere istanziata. Questa decisione ci è sembrata la più corretta in quanto non è necessario creare istanze di queste classi dato che le loro funzioni non agiscono direttamente sulle altre classi ma forniscono solo informazioni o hanno ruoli molto specifici e non legati agli altri oggetti.

2.2.1 OperativeSystemManagement

Dall'analisi delle componenti **5.1.8 Profilo studente** e **5.1.18 Profilo offerente**, si evince la necessità della creazione di una classe **OperativeSystemManagement**. Nello specifico, la classe astratta **OperativeSystemManagement** contiene tutti i metodi che riguardano le modalità di gestione della pagina web.

La classe **OperativeSystemManagement** presenta due attributi: *language* equivale alla lingua impostata dall'utente (italiano oppure inglese mentre *sysColor* rappresenta il colore di visualizzazione delle pagine. Per una gestione più efficiente, si è deciso di impostare gli attributi *language* e *sysColor* a bool in quanto, rispettivamente, uno può assumere solamente valore inglese (true) e italiano (false) mentre l'altro valore chiaro (true) e scuro (false).

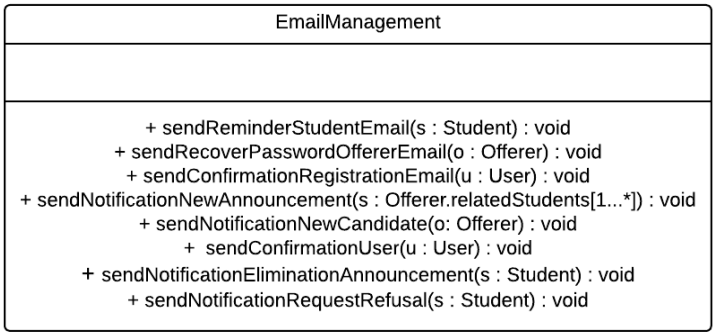


Di seguito una breve descrizione dei metodi implementati dalla classe **OperativeSystemManagement:**

+ `setLanguage(language : bool)` permette di modificare la lingua di sistema, scegliendo tra italiano (default) e inglese.
+ `setSysColor(sysColor : bool)` permette di modificare la modalità di colore di visualizzazione del sistema, optando tra modalità chiara (default) e modalità scura.
+ `getCurrentDateAndTime()` permette di ottenere la data e l'orario attuali restituendo una coppia di valori.

2.2.2 EmailManagement

Dall'analisi della componente **5.2.23 Posta elettronica**, si evince la necessità della creazione di una classe **EmailManagement**. La classe astratta **EmailManagement** si occupa della gestione delle notifiche e-mail il cui invio è fornito attraverso il sistema di Posta Elettronica. Le funzioni contenute da **EmailManagement** vengono dunque chiamate ogni qualvolta si voglia comunicare con l'utente.



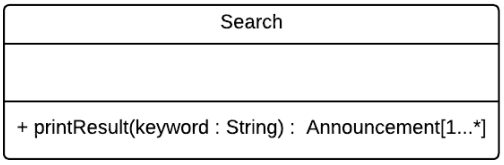
Di seguito una breve descrizione dei metodi implementati dalla classe **EmailManagement:**

+ `sendReminderStudentEmail(s : Student)` permette l'invio di un'e-mail di reminder allo studente il giorno prima dello svolgimento di un servizio a cui si è candidato.
+ `sendRecoverPasswordOffererEmail(o : Offerer)` permette l'invio di e-mail di notifica all'offerente per il ripristino della password di accesso al sito.
+ `sendConfirmationRegistrationEmail(u : User)` permette l'invio di una notifica e-mail per confermare la registrazione dell'utente.
+ `sendNotificationNewAnnouncement(s : Student[1...*])` permette l'invio di un-email di notifica allo studente nel caso in cui un offerente con cui ha già avuto contatto pubblica un nuovo annuncio, solo se lo studente ha precedentemente attivato il flag di notifica. Questa funzione prende in input un'array di studenti *relatedStudents[1...*]* che hanno attivato il flag presente come attributo della classe *Offerer*.
+ `sendNotificationNewCandidate(o : Offerer)` permette l'invio di un'e-mail di notifica all'offerente quando uno studente si candida per un annuncio attivo.
+ `sendConfirmationUser(u : User)` permette l'invio all'utente un'e-mail di conferma per l'identificazione dell'utente in caso di inserimento errato della password per tre volte consecutive.
+ `sendNotificationEliminationAnnouncement(s : Student)` permette l'invio di una notifica e-mail allo studente nel caso in cui l'offerente effettui la cancellazione di un proprio annuncio attivo a cui lo studente si era precedentemente proposto.

+ `sendNotificationRequestRefusal(s : Student)` permette l'invio di una notifica e-mail allo studente nel caso in cui l'offerente rifiuti una sua richiesta per un servizio.

2.2.3 Search

Dall'analisi della componente **5.1.16 Ricerca annuncio**, si evince la necessità della creazione di una classe **Search**. La classe astratta **Search** si occupa della gestione della ricerca degli annunci basata sulle parole chiave utilizzate dallo studente per descrivere il servizio richiesto.



Di seguito una breve descrizione del metodo implementato dalla classe **EmailManagement:**

+ `printResult(keyword : String)` permette di stampare la lista degli annunci che soddisfano la ricerca effettuata dallo studente tramite la barra di ricerca – passata alla funzione tramite la stringa *KeyWord* - restituendo un array di *Announcement*.

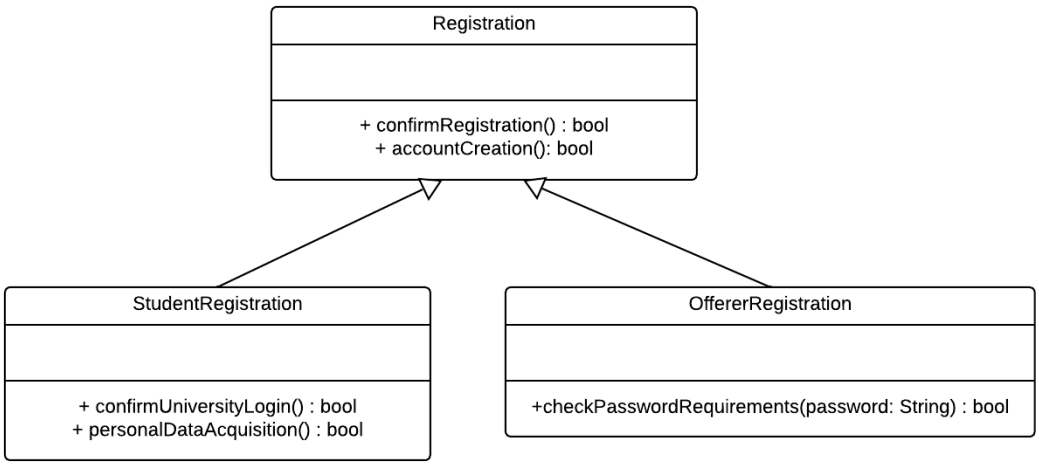
2.3 Classi individuate dal diagramma di contesto e dei componenti

Di seguito vengono presentate le classi derivare dall'analisi del diagramma di contesto e dei componenti descritti nel documento **D2: Specifica dei Requisiti**.

2.3.1 Registration

Dall'analisi delle componenti **5.2.3 Interfaccia registrazione studente** e **5.2.13 Interfaccia registrazione offerente**, si evince la necessità della creazione di una classe **Registration**.

Nel documento **D1: Analisi dei requisiti** sono state distinte due tipologie di account – utente studente e utente offerente. Al fine di implementare nel modo più efficiente la procedura di registrazione si è resa necessaria la creazione di due sottoclassi figlie della classe *Gestione Registrazione* che si distinguono per la tipologia di utente che si registra per utilizzare l'applicazione. In particolare, le due sottoclassi individuate sono *Student*Registration e *Offerer*Registration.



Di seguito una breve descrizione dei metodi implementati dalla classe **Registration:**

+ `confirmRegistration()` controlla, richiamando anche eventualmente gli opportuni metodi, che i dati inseriti dall'utente siano corretti e rispettino gli standard specificati nel documento **D2: Analisi dei requisiti**. Si noti che i controlli effettuati sui dati variano in base alla tipologia di utente su cui viene richiamata (vedi descrizione metodi classe **StudentRegistration** e classe **OffererRegistration**).
+ `accountCreation()` si occupa della creazione vera e propria dell'account nel caso in cui confirmRegistration sia andata a buon fine ritornando come risultato *true*.

Di seguito una breve descrizione dei metodi implementati dalla classe **StudentRegistration:**

+ `confirmUniversityLogin()` gestisce la richiesta della verifica delle credenziali effettuata dal componente **5.2.2 Credenziali universitarie studenti** per confermare la registrazione dell'account.
+ `personalDataAcquisition()` si occupa dell'acquisizione dalla componente **5.2.2 Credenziali universitarie studenti** dei dati personali utente e del loro salvataggio.
+ `confirmRegistration()` effettua l'override del metodo della superclasse e richiama il metodo `confirmUniversityLogin()` per verificare il corretto login nel form dell'università.

Di seguito una breve descrizione dei metodi implementati dalla classe **OffererRegistration:**

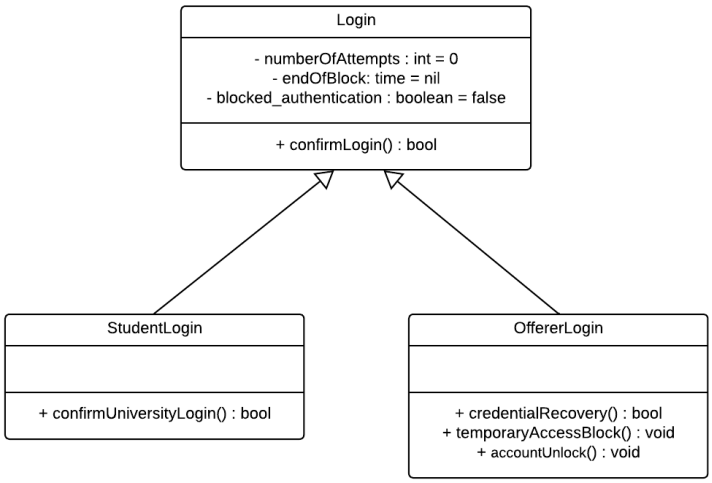
+ `confirmRegistration()` effettua l'override del metodo della superclasse e si occupa di verificare che le credenziali inserite dall'utente offerente siano corrette.
+ `checkPasswordRequirements(password : String)` controlla che la password inserita soddisfi i requisiti Strong Password - definiti nella sezione di sicurezza dei requisiti non funzionali del documento **D1: Analisi dei requisiti** - che deve rispettare per essere accertata.

2.3.2 Login

Dall'analisi delle componenti **5.2.4 Interfaccia login studente** e **5.2.14 Interfaccia login offerente**, si evince la necessità della creazione di una classe **Login**. Questa classe si occupa di gestire tutte le

funzionalità che l'utente può utilizzare una volta effettuati correttamente registrazione e accesso all'applicazione.

La classe **Login** presenta diversi attributi: *numberOfAttempts* è il contatore dei tentativi fatti dall'utente (> 0 se la password è stata sbagliata almeno una volta) in una sessione di autenticazione, mentre *blocked_authentication* rappresenta lo stato dell'account: bloccato oppure sbloccato. Questo attributo è un tipo booleano che di base è impostato a false, ovvero inizialmente l'autenticazione è sempre sbloccata cioè può essere svolta (per una specifica sessione). Nel momento in cui l'attributo *numberOfAttempts* equivale ad un numero maggiore o uguale a 3, *blocked_authentication* assume il valore true e, di conseguenza, l'autenticazione per quella sessione viene bloccata. Infine, *endOfBlock* è l'attributo rappresentante il tempo passato dal momento in cui è stato bloccata la sessione.



Di seguito una breve descrizione dei metodi implementati dalla classe **Login:**

+ `confirmLogin()` varia in basa alla tipologia di utente su cui viene richiamata (vedi descrizione metodi classe **StudentLogin** e classe **OffererLogin**).

Di seguito una breve descrizione dei metodi implementati dalla classe **StudentLogin:**

+ `confirmUniversityLogin()` si occupa di verificare che la verifica delle sue credenziali di accesso affidata al *Sistema Credenziali Universitarie* sia andata a buon fine.
+ `confirmLogin()` richiama il metodo `confirmUniversityLogin()` e controlla che questo restituisca il valore *true*.

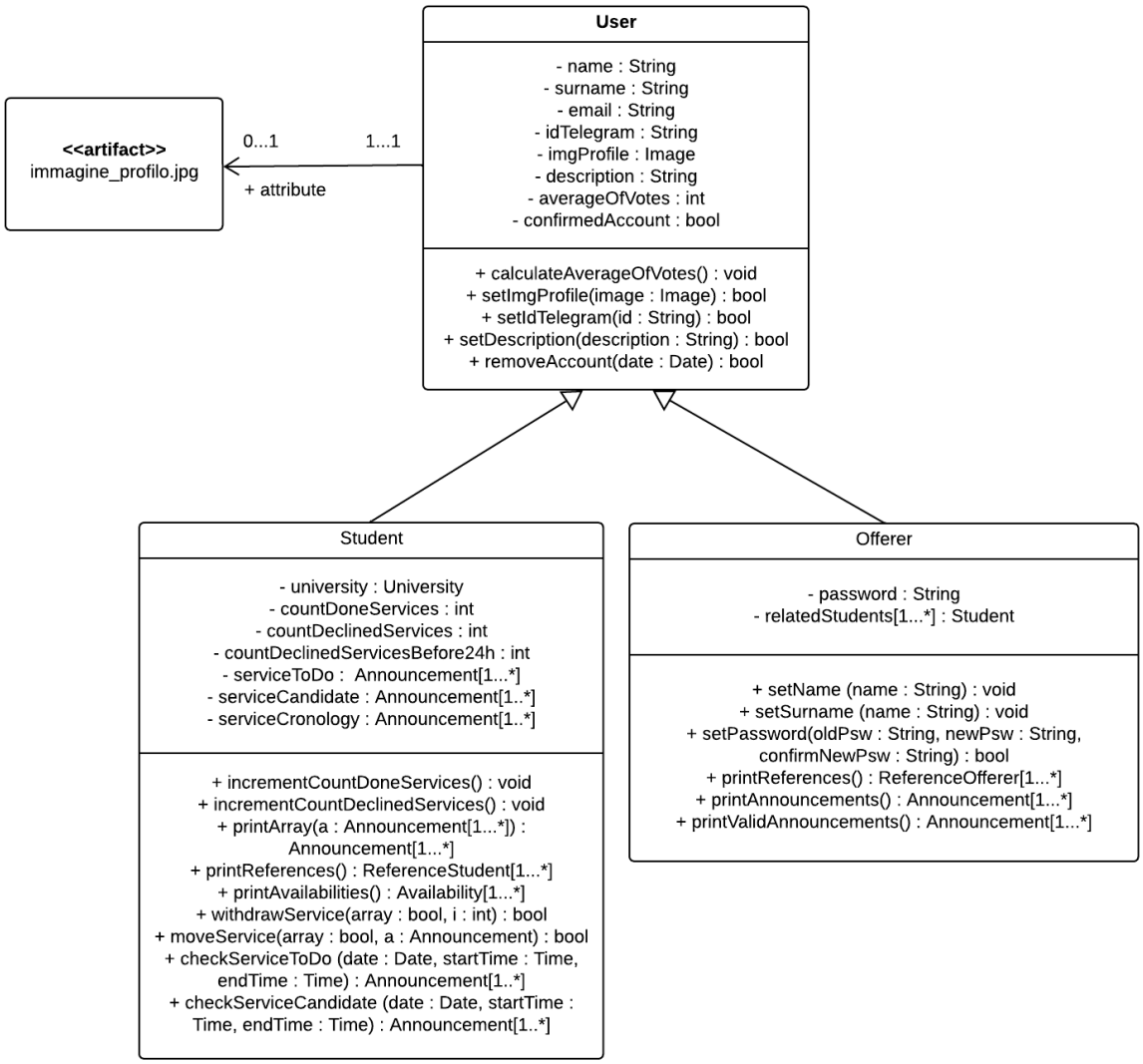
Di seguito una breve descrizione dei metodi implementati dalla classe **OffererLogin:**

+ `confirmLogin ()` verifica che le credenziali inserite dall'utente offerente siano corrette.
+ `credentialRecovery()` gestisce la procedura di recupero password per l'accesso al sito da parte dell'utente offerente.
+ `temporaryAccessBlock()` gestisce il blocco dell'account nel caso di inserimento errato della password per tre volte consecutive. In questo caso la possibilità di login per l'utente viene infatti bloccata per un'ora.

+ `accountUnlock()` permette di sbloccare l'account dopo un'ora.

2.3.3 User

Come si evince dall'analisi dei componenti presentata nel documento **D2: Specifica dei requisiti**, il sistema si basa su due tipologie differenti di utenti: utente studente e utente offerente; lo studente è colui che utilizza il sistema per fornire servizi mentre l'offerente è colui che utilizza il sistema per richiedere servizi. Viene quindi evidenziata la necessità delle classi **Student** e **Offerer** che ereditano gli attributi e metodi dalla super-classe **User**.



La classe **User** presenta diversi attributi: *name* e *surname* sono i dati personali dell'utente, *email* e (opzionale) *idTelegram* sono le informazioni di contatto dell'utente, (opzionale) *imgProfile* e (opzionale) *description* sono informazioni che permettono di fornire un profilo più completo e dettagliato agli utenti del sito, *averageOfVotes* è l'attributo che mantiene la media dei voti dell'utente basati sulle referenze lasciategli dagli

altri utenti e *confirmedAccount* permette al sito la gestione dell'account utente. Si noti che l'immagine del profilo *imgProfile* viene presa in input come file "immagine_profilo.jpg" .

La classe **Student**, oltre ad ereditare gli attributi della classe **User**, possiede: *university* è uno degli attributi su cui si basa la visualizzazione degli annunci nel sito, *countDoneServices*, *countDeclinedServices* e *countDeclinedServicesBefore24h* permettono di mantenere in memoria il numero di servizi rispettivamente svolti e rifiutati più/meno di 24h prima del momento di svolgimento del servizio. Inoltre, *serviceToDo* è l'attributo che salva la lista degli annunci a cui lo studente è stato accettato, *serviceCandidate* mantiene la lista degli annunci ancora attivi e il cui stato è diverso da accettato mentre *serviceCronology* coincide alla lista degli annunci non più attivi, utile per visualizzare la cronologia dei servizi effettuati da uno studente.

La classe **Offerer**, oltre ad ereditare gli attributi della classe **User**, possiede: *password* che coincide con la password dell'account e *relatedStudents* che – essendo un array di Student – permette di mantenere per ogni Offerer gli studenti che hanno attivato il flag di notifica per i nuovi annunci da parte dell'offerente stesso; quest'ultimo è necessario per un'efficiente gestione della notificazione degli studenti che, avendo attivato il flag, desiderano essere notificati quando tale offerente pubblica un nuovo annuncio.

Di seguito una breve descrizione dei metodi implementati dalla classe **User**:

+ `calculateAverageOfVotes()` permette di calcolare e settare l'attributo *averageOfVotes* effettuando una media dei voti ricevuti dal offerente/studente per un servizio svolto/richiesto.
+ `setImgProfile(image : Image)` permette di settare/modificare l'immagine del profilo data un'immagine da input.
+ `setIdTelegram(id : String)` permette di inserire/modificare l'id Telegram dell'utente.
+ `setDescription(description : String)` permette di aggiungere/modificare una breve descrizione personale che sarà visibili agli altri utenti del sito.
+ `removeAccount(date : Date)` permette all'utente di eliminare il proprio account dal sistema.

Di seguito una breve descrizione dei metodi implementati dalla classe **Student**:

+ `incrementCountDoneServices()` permette di incrementare la variabile *countDoneServices* che memorizza a sistema il numero di lavori completati dall'utente studente.
+ `incrementCountDeclinedServices()` permette di incrementare la variabile *countDeclinedServices* che memorizza a sistema il numero di lavori rifiutati dall'utente studente.
+ `printArray(a : Announcement[0…*])` permette di stampare l'array di Announcement dato in input.
+ `printReferences()` permette di stampare la lista delle referenze ottenute dall'utente studente restituendo un array di *ReferenceStudent*.
+ `printAvailabilities()` permette di stampare la lista delle disponibilità dell'utente studente restituendo un array di *Availability*.
+ `withdrawService(array : bool, i : int)` permette all'utente studente di ritirarsi da un servizio a cui si era candidato. Questo metodo restituisce un *bool* per confermare l'avvenuta procedura.
+ `moveService(array : bool, a : Announcement)`: permette di spostare l'announcement a dall'array *serviceToDo (array=true)* oppure dall'array *serviceCandidate (array=false)* nell'array *serviceCronology*.

+ `checkServiceToDo(date : Date, startTime : Time, endTime : Time)` serve, dato in input data-startTime-endTime, per controllare se nell'array *serviceToDo* vi sono annunci all'interno di quella fascia oraria.
+ `checkServiceCandidate(date : Date, startTime : Time, endTime : Time)` serve, dato in input data-startTime-endTime, per controllare se nell'array *serviceCandidate* vi sono annunci all'interno di quella fascia oraria.

Di seguito una breve descrizione dei metodi implementati dalla classe **Offerer**:

+ `setName(name : String)` permette di inserire/modificare il nome del profilo.
+ `setSurname(surname : String)` permette di inserire/modificare il cognome del profilo.
+ `setPassword(oldPsw : String, newPsw : String, confirmNewPsw : String)` permette di inserire/modificare la password del profilo.
+ `printReferences()` permette di stampare la lista delle referenze ottenute dall'utente offerente restituendo un array di *ReferenceOfferer*.
+ `printAnnouncements()` permette di stampare la lista dei lavori creati dall'utente offerente, e relative caratteristiche, restituendo un array di *Announcement*.
+ `printValidAnnouncements()` permette di stampare la lista dei lavori ancora attivi – ovvero i lavori i cui attributi *date* e *endTime* sono successivi alla data e ora di verifica – creati dall'utente offerente, e relative caratteristiche, restituendo un array di *Announcement*.

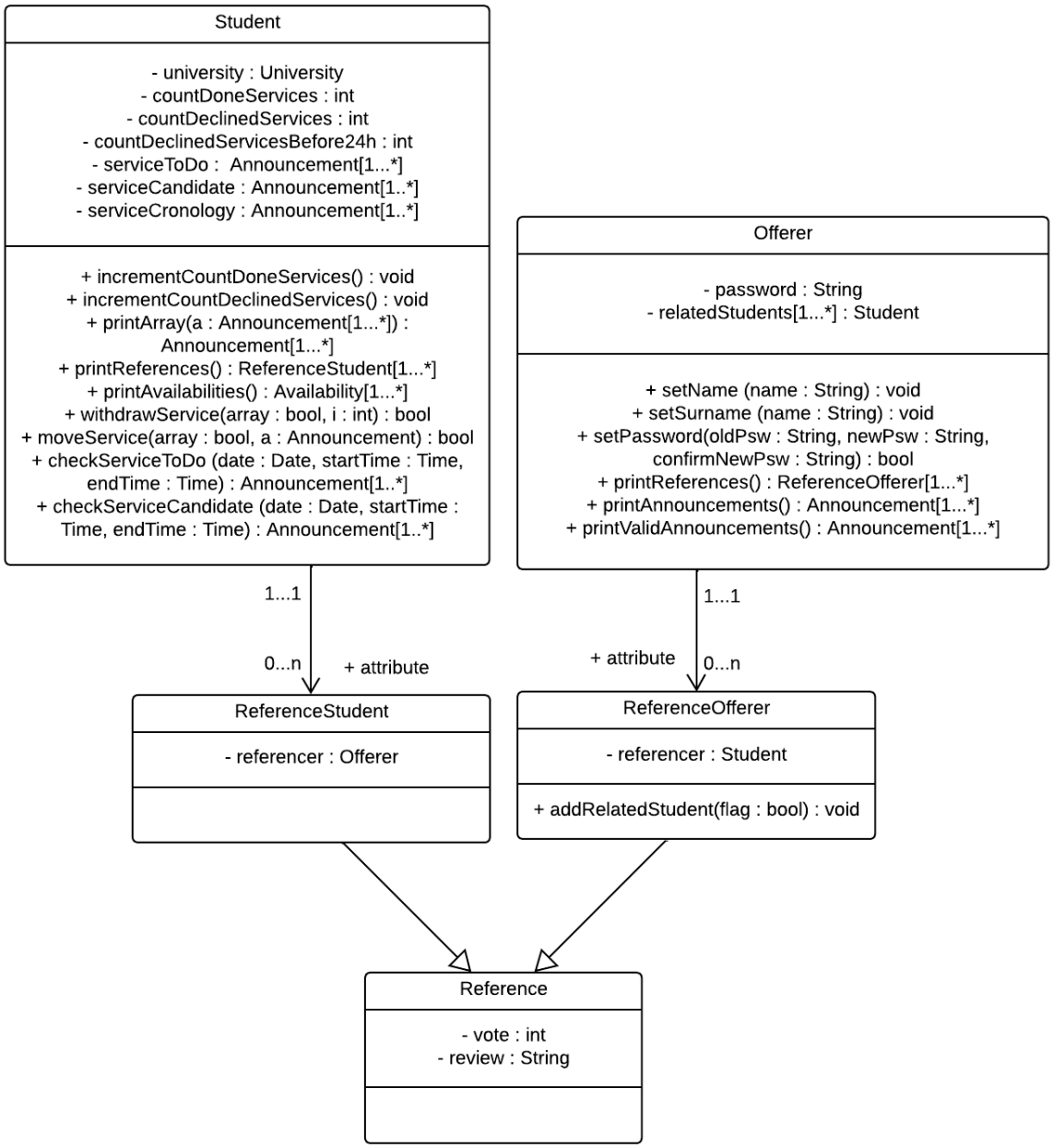
2.3.4 Reference

I componenti **5.2.9 Referenze studente**, **5.2.19 Referenze offerente** e **5.2.22 Crea referenza** necessitano l'individuazione di una classe **Reference** che permetta di creare e gestire le referenze inserite a sistema. Al fine di implementare in modo più efficiente la procedura di creazione delle referenze da parte degli utenti, e gestirne le relative funzionalità specifiche, sono state identificate le due sotto-classi **ReferenceStudent** e **ReferenceOfferer**. Nello specifico, **ReferenceStudent** è la classe che gestisce la creazione di una referenza allo studente da parte di un utente offerente, mentre la classe **ReferenceOfferer** è la classe che gestisce la creazione di una referenza all'offerente da parte di un utente studente.

La classe **Reference** possiede due attributi: *vote* che mantiene il voto aggiunto nella referenza e *review* che corrisponde alla recensione scritta dall'utente.

La classe **ReferenceStudent** possiede, oltre agli attributi ereditati dalla classe **Reference**, il parametro *referencer* di tipo Offerer che permette di mantenere l'utente offerente a cui lo studente sta effettuando la recensione.

La classe **ReferenceOfferer** possiede, oltre agli attributi ereditati dalla classe **Reference**, il parametro *referencer* di tipo Student che permette di mantenere l'utente studente a cui l'offerente sta effettuando la recensione.



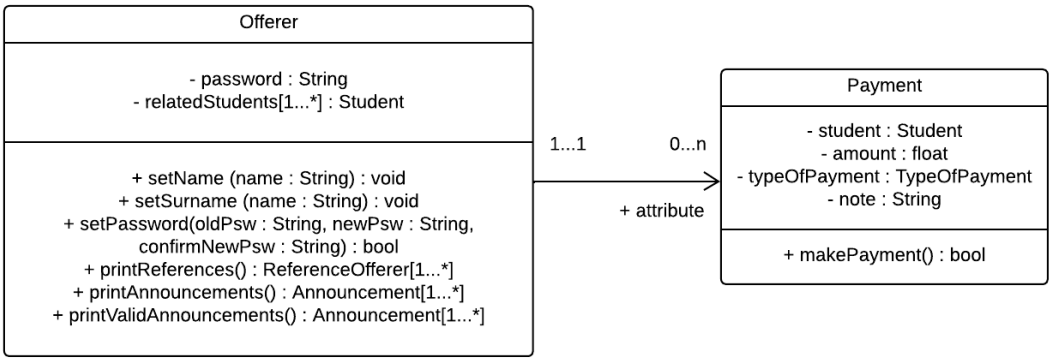
Di seguito una breve descrizione dei metodi implementati dalla classe **ReferenceOfferer**:

+ `addRelatedStudent(flag : bool)` permette, in caso di valore true del booleano *flag* richiesto dal metodo, di aggiungere tale studente nell'array *relatedStudents*. Questo serve per avvisare della pubblicazione di un nuovo annuncio da parte di tale offerente a tutti gli studenti che hanno deciso, attivando il flag al momento della creazione di una referenza, di essere notificati.

2.3.5 Payment

Il componente **5.2.20 Pagamento** esplicita la necessità di una classe **Payment** al fine di gestire i pagamenti – da effettuare ed effettuati – da parte dell'offerente verso uno studente. Tale classe è infatti in relazione con la classe **Offerer**.

La classe **Payment** presenta diversi attributi: *student* mantiene lo studente a cui viene effettuato il pagamento, *amount* mantiene l'importo da pagare a servizio effettuato, *typeOfPayment* mantiene la modalità di pagamento selezionata dall'offerente e *note* corrisponde ad eventuali note inserite dall'offerente.



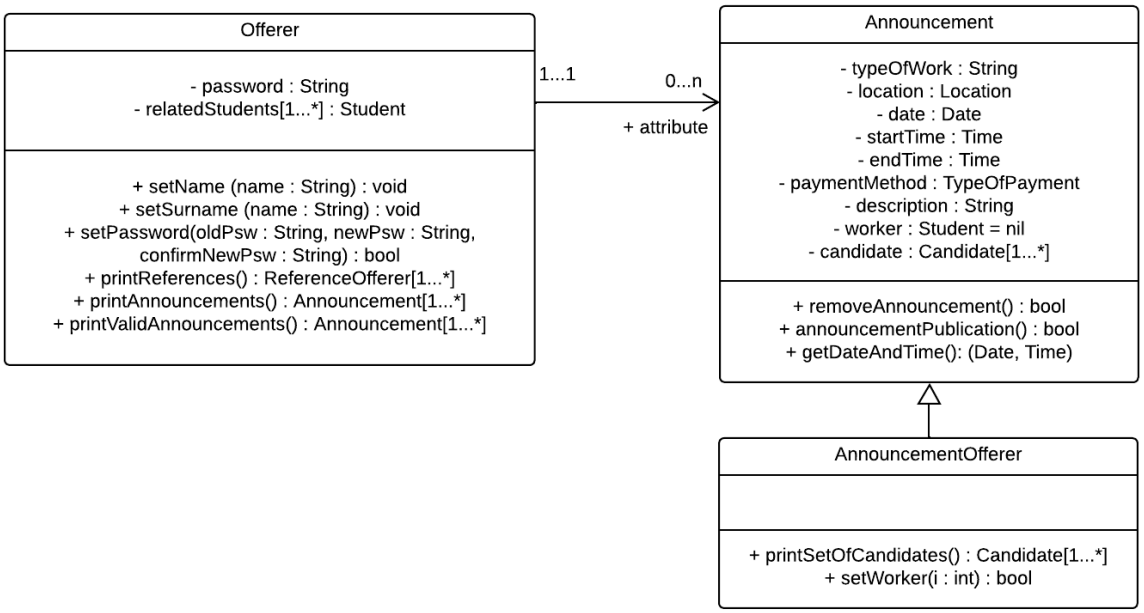
Di seguito una breve descrizione dei metodi implementati dalla classe **Payment**:

+ `makePayment()` si occupa, nel caso il valore assunto dall'attributo *typeOfPayment* corrisponda a "PayPal", di interfacciarsi con il sistema esterno PayPal e verificare l'avvenuta transazione. Questo metodo restituisce un valore *bool* equivalente al risultato della transazione (true se è avvenuta correttamente, false altrimenti).

2.3.6 Announcement

Dall'analisi del componente **5.1.17 Annuncio singolo**, si evince la necessità della creazione di una classe **Announcement**. Questa si occupa della gestione degli annunci. Per la creazione di questo vengono richiesti all'utente dei dati specifici, quali il tipo di lavoro – ovvero un insieme di parole chiave per descrivere il servizio richiesto , (eventuale) luogo dove verrà svolto il lavoro, la data e l'ora in cui deve essere svolto il lavoro, la tipologia di pagamento prevista e una breve descrizione del lavoro che lo studente dovrà effettuare. Viene inoltre evidenziata la necessità della creazione della sottoclasse **AnnouncementOfferer** che eredita gli attributi e metodi dalla super-classe **Announcement** e rappresenta una vista più dettagliata dell'annuncio lato offerente.

La classe **Announcement** presenta diversi attributi: typeOf*Work* è la tipologia di servizio proposta dall'offerente, *location* rappresenta la locazione in cui il lavoro deve essere svolto, *date* equivale al giorno in cui tale attività deve essere svolta, *startTime* e *endTime* permettono di memorizzare rispettivamente l'ora di inizio e l'ora di fine della disponibilità (evidenziano e definiscono quindi la fascia oraria di disponibilità), *paymentMethod* esplicita la tipologia di pagamento richiesto, *description* è la breve descrizione del servizio inserita dall'offerente, *worker* – inizialmente settato a `nil` – assume il valore di puntatore allo studente scelto dall'offerente tra la lista dei candidati per effettuare il servizio mentre *candidate* è la lista di studenti che si sono candidati per tale annuncio.



Di seguito una breve descrizione del metodo implementato dalla classe **Announcement:**

- `removeAnnouncement()` gestisce l'eliminazione di un annuncio attivo.
- `announcecementPublication()` segnala se la pubblicazione di un nuovo annuncio è andata a buon fine.
- `getDateAndTime()` restituisce la coppia di valori (*date*,*startTime*) della classe.

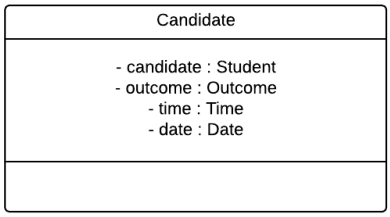
Di seguito una breve descrizione del metodo implementato dalla sottoclasse **AnnouncementOfferer:**

- `printSetOfCandidates()` permette all'utente offerente la visualizzazione della lista dei candidati ad un suo specifico annuncio.
- `setWorker(i : int) permette di` settare uno studente come *worker* tra quelli presenti nell'array *candidate*. Nello specifico, tale metodo possiede in input un intero che corrisponde all'indice del vettore *candidate* relativo allo studente selezionato.

2.3.7 Candidate

Dall'analisi del componente **5.1.17 Annuncio singolo**, si evince la necessità della creazione di una classe **Candidate**. Questa classe si occupa della gestione della lista candidati visibile dall'offerente. Come specificato nel documento **D1: Analisi dei requisiti** il sito offre a tutti gli studenti autenticati la possibilità di candidarsi ad un annuncio.

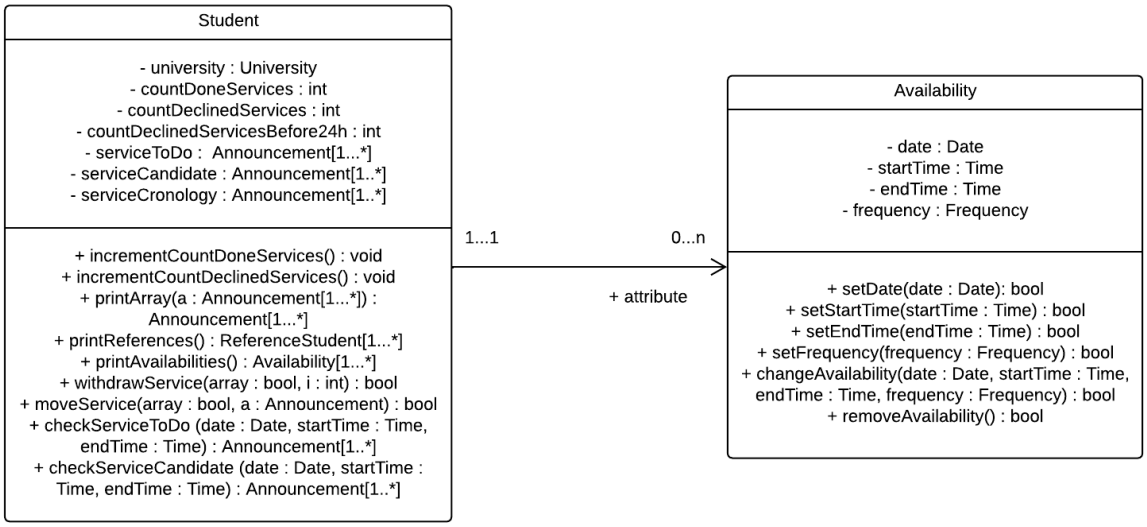
La classe **Candidate** presenta dunque due attributi: l'attributo *candidate* può essere solamente uno studente iscritto all'applicazione. Per ogni studente che si propone per un annuncio il sito offre la visualizzazione di una della pagina *Stato richieste attive* che mostra tutti gli annunci per cui lo studente si è reso disponibile. Attraverso l'attributo *outcome* è invece possibile visionare lo stato di tali annunci. Vi sono inoltre l'attributo *time* e *date* che mantengono rispettivamente il tempo e la data ultima in cui si è registrato un cambiamento del valore dell'attributo *outcome*.



2.3.8 Availability

Dall'analisi del componente **5.2.10 Calendario** si evince la necessità di creare una classe **Availability** per la creazione e la gestione delle disponibilità inserite nel calendario dallo studente. Tale classe è infatti in relazione con la classe **Student**.

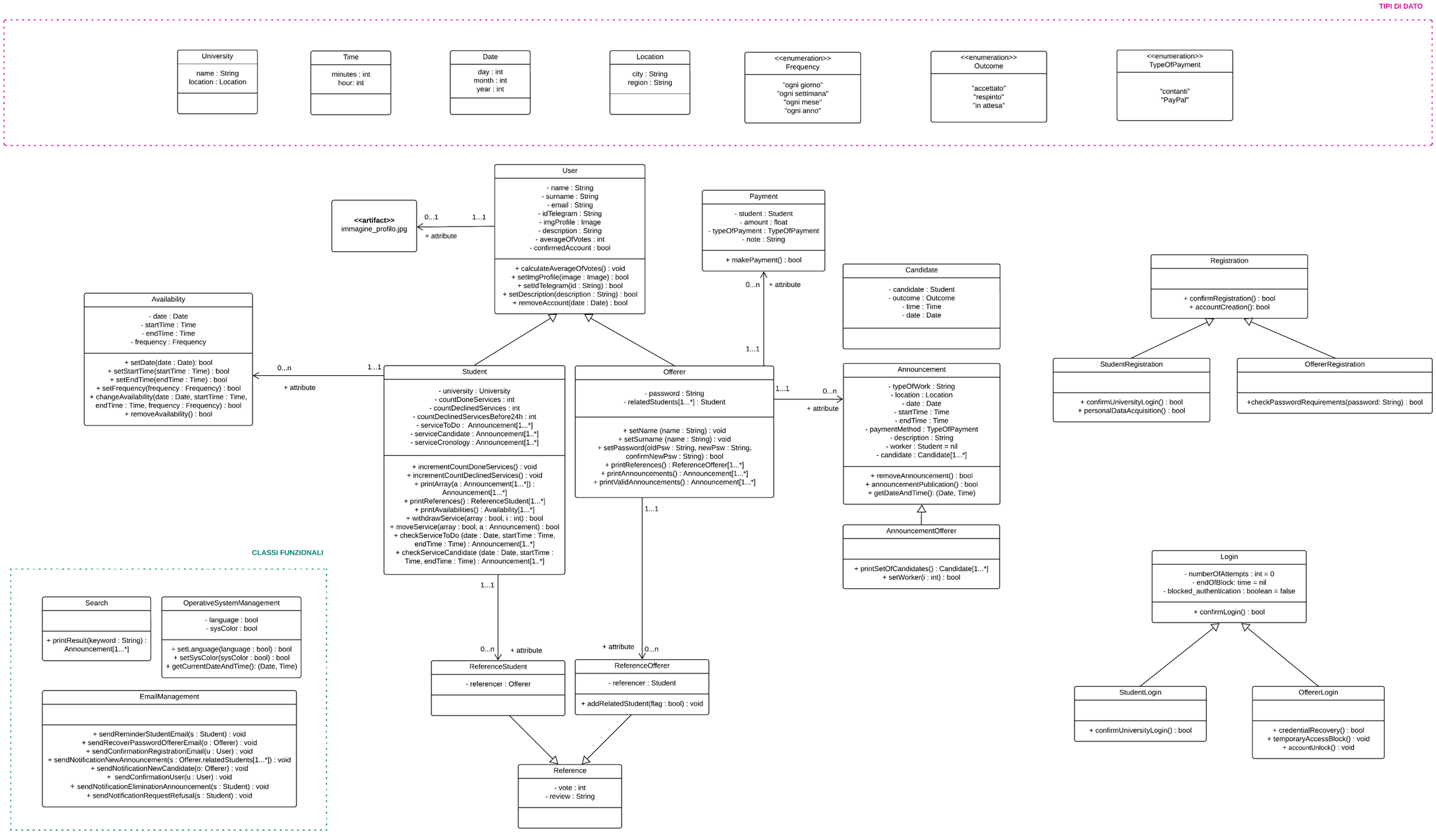
La classe **Availability** presenta diversi attributi: *date* equivale al giorno a cui si riferisce tale disponibilità, *startTime* e *endTime* permettono di memorizzare rispettivamente l'ora di inizio e l'ora di fine della disponibilità (evidenziano e definiscono quindi la fascia oraria di disponibilità) mentre *frequency* esplicita la frequenza con cui tale disponibilità viene ripetuta.



Di seguito una breve descrizione dei metodi implementati dalla classe **Availability**:

+ `setDate(date : Date)` permette di settare/aggiornare il campo *date*.
+ `setStartTime(startTime : Time)` permette di settare/aggiornare il campo *startTime*.
+ `setEndTime(endTime : Time)` permette di settare/aggiornare il campo *endTime*.
+ `setFrequency(frequency : Frequency)` permette di settare/aggiornare il campo *frequency*.
+ `changeAvailability(date : Date, startTime : Time, endTime : Time, frequency : Frequency)` permette, richiamando le rispettive funzioni `set*`, di modificare gli attributi della classe.
+ `removeAvailability()` permette di eliminare tale disponibilità dal sistema.

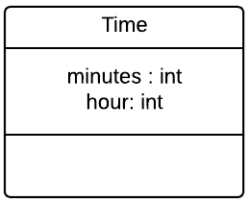
2.4 Diagramma delle classi complessivo



3. Codice in Object Constraint Language

In questa sezione viene descritta la logica prevista in alcune operazioni di certe classi. Nello specifico, questa è descritta mediante l'utilizzo di *Object Constraint Language* (OCL), strumento che viene usato dal momento che non tutti i concetti possono essere espressi in modo formale in UML.

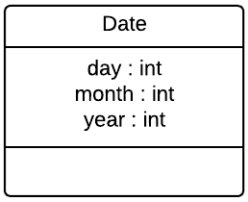
3.1 Time



L'orario deve essere caratterizzato sempre da un range di minuti compreso tra 0 e 59 e di ore tra 0 e 23. Queste condizioni sono espresse in OCL attraverso un invariante nel seguente modo:

```
context Time inv:
minutes >= 0 AND minutes <= 59
hour >= 0 AND hour <= 23
```

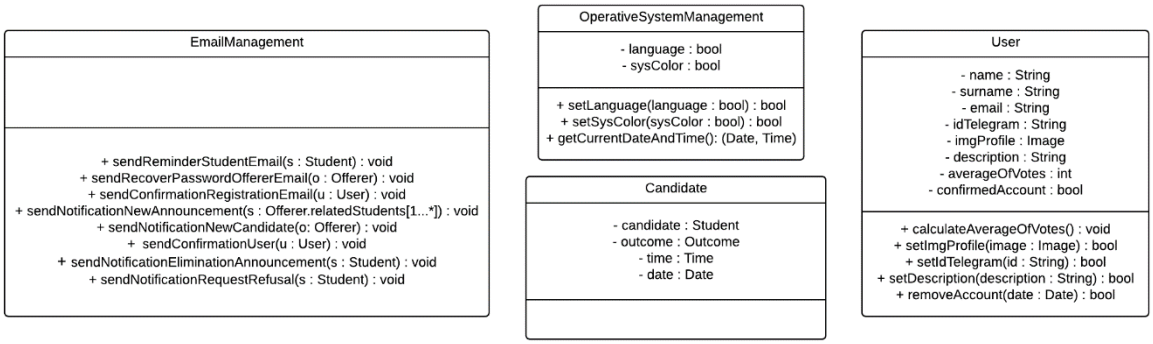
3.2 Date



La data deve essere caratterizzata sempre da un giorno compreso tra 1 e 31, un mese compreso tra 1 e 12 e un anno maggiore o uguale a 1850. Queste condizioni sono espresse in OCL attraverso un invariante nel seguente modo:

```
context Date inv:
day >= 1 AND day <= 31
month >= 1 AND month <= 12
year >= 1850
```

3.3 EmailManagement



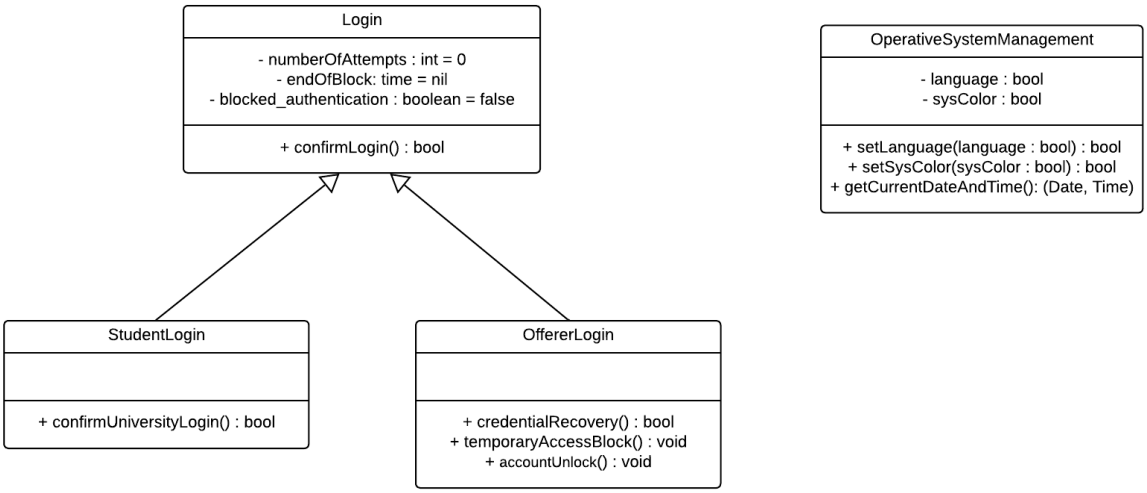
Quando si effettua la registrazione al sito, sia per l'utente studente sia per l'utente offerente, è necessario inserire determinate informazioni (es: *nome*, *cognome*) e rispettare determinati standard imposti dal sito (es: la *password* di accesso al sito deve rispettare i requisiti di "Strong Password" esplicitati nel documento **D2: Specifica dei requisiti**). Al fine di effettuare la registrazione, quindi, il sistema effettua un controllo sui dati inseriti. . Ciò può essere espresso in OCL attraverso l'utilizzo di una pre-condizione sul metodo `sendConfirmationRegistrationEmail(u : User)` nel seguente modo:

context EmailManagement :: sendConfirmationRegistrationEmail(u : User)
pre: User.confirmRegistration() = true

Quando viene accettata la richiesta ad un servizio di uno studente, il giorno antecedente all'inizio del lavoro, il sistema invia un'e-mail automatica allo studente per ricordargli del servizio che ha concordato con l'offerente. Ciò può essere espresso in OCL attraverso l'utilizzo di una pre-condizione nel seguente modo:

context EmailManagement :: sendReminderStudentEmail()
pre: OperativeSystemManagement.getCurrentDateAndTime().second - Announcement.startTime <= 24:00
AND Candidate.Outcome = "accettato"

3.4 OffererLogin



Nell'eventualità in cui l'utente offerente provi ad accedere all'applicazione inserendo la password errata, si verifica un incremento della variabile *numberOfAttempts* di un'unità. Ciò può essere espresso in OCL attraverso l'utilizzo di una post-condizione nel seguente modo:

context OffererLogin :: confirmLogin()

post: self.numberOfAttempts = self.numberOfAttempts + 1

Quando la variabile *numberOfAttempts* raggiunge o supera il valore 3, il sito si blocca per un'ora. Ciò può essere espresso in OCL attraverso l'utilizzo di una pre-condizione e una post-condizione nel seguente modo:

context OffererLogin :: temporaryAccessBlock()

pre: self.numberOfAttempts ≥ 3

post: self.endOfBlock = OperativeSystemManagement.getCurrentDateAndTime().second + 24:00

AND self.blocked_authentication = true

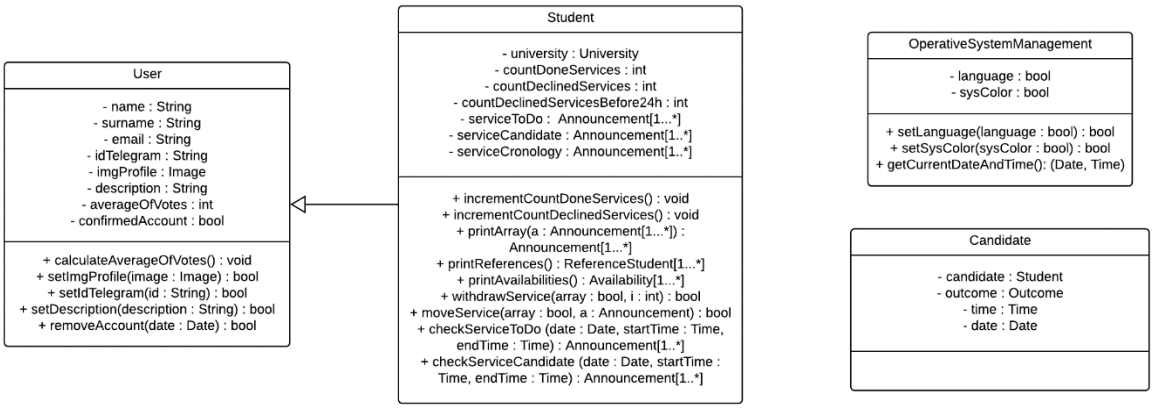
Un ulteriore decisione presa è stata quella di creare un metodo che permette di sbloccare l'accesso al sito solo dopo un'ora dal momento del blocco. Ciò può essere espresso in OCL attraverso l'utilizzo di una pre-condizione e una post-condizione nel seguente modo:

context OffererLogin :: accountUnlock()

pre: OperativeSystemManagement. getCurrentDateAndTime() > self.endOfBlock

post: self.blocked_authentication = false AND self.endOfBlock = nil

3.5 Student



Il numero di effettuati da uno studente e il numero di servizi respinti (oltre e prima delle 24 ore) è sempre pari o maggiore a 0. Queste condizioni sono espresse in OCL attraverso un invariante nel seguente modo:

```
context Student inv:
countDoneServices >= 0
countDeclinedServices >= 0
countDeclinedServicesBefore24h>=0
```

Per effettuare correttamente la rimozione dell'account studente, l'applicazione impone diverse condizioni che devono essere rispettate. Innanzitutto, nel caso in cui la sezione di visualizzazione dello stato di richiesta degli annunci sia vuota, lo studente può effettuare immediatamente l'eliminazione dell'account. Se invece tale sezione contiene uno o più annunci a cui lo studente ha fatto domanda che presentano lo stato "*accettato*", lo studente ha la possibilità di ritirarsi da uno specifico annuncio solo nell'eventualità in cui il tempo mancante dallo svolgimento del servizio richiesto è maggiore di 72 ore. Altrimenti, se l'annuncio presenta stato "*in attesa*" o *"respinto"*, lo studente può ritirarsi in qualsiasi momento. Ciò può essere espresso in OCL attraverso l'utilizzo di una pre-condizione nel seguente modo:

```
context Student :: removeAccount(date : Date)
pre: (self.printArray(self.serviceToDo) = nil) OR (self.printArray(self.serviceToDo) <> nil AND Candidate.outcome = "accettato"
AND OperativeSystemManagement.getCurrentDateAndTime().first - date > 3) OR (self.printArray(self.serviceToDo) <> nil
AND (Candidate.outcome = "non accettato" OR Candidate.outcome = "in attesa"))
```

Ogni studente possiede tre array di Announcement distinti: *serviceToDo*, *serviceCandidates* e *serviceCronology*. Come descritto precedentemente, *serviceToDo* contiene solamente Announcement ancora attivi per cui lo studente si è candidato ed è stato accettato, *serviceCandidates* contiene solamente Announcement ancora attivi per cui lo studente si è candidato e non è ancora stato accettato o è stato rifiutato. Come specificato nel **D1: Analisi dei requisiti** e nel **D2: Specifica dei requisiti**, un annuncio lato studente è definito attivo quando:

1. è in stato "in attesa" e la data dell'annuncio è successiva dalla data di analisi dell'annuncio;
2. è in stato "accettato" e la data dell'annuncio è precedente alla data di analisi più 48h;

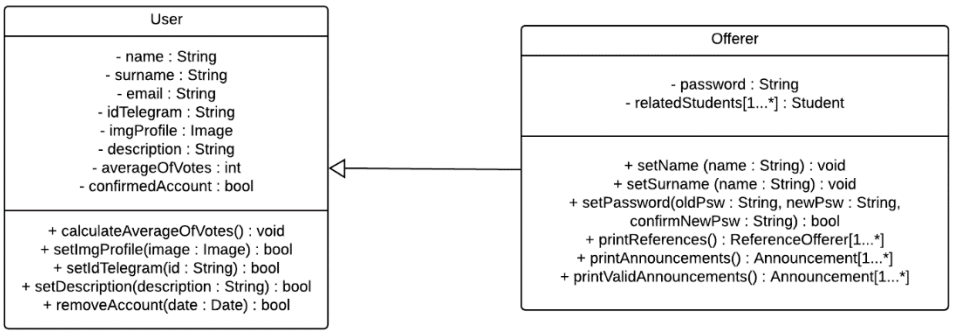
3. è in stato "respinto" e la data dell'annuncio è precedente alla data di analisi più 72h.

Quando un annuncio – sempre vista studente – non è più considerato "attivo", è necessario spostare tale annuncio dall'array in cui è memorizzato all'array *serviceCronology*. Come precedentemente spiegato (vedi qui), il metodo `moveService(array : bool, a: Announcement)` permette lo spostamento di un Announcement *a* nell'array *serviceCronology*. Nello specifico, in base al valore booleano assunto da *array*, l'Announcement *a* proviene da (0) *serviceCandidates* o (1) *serviceToDo*.

Quindi, per effettuare lo spostamento di un annuncio quando questo non è più attivo, è possibile applicare la seguente pre-condizione sul metodo `moveService(array : bool, a: Announcement)`:

context Student :: moveService(array : bool, a: Announcement)
pre: array=0 implies OperativeSystemManagement.getCurrentDateAndTime() - a.getDateAndTime() >= 0000/00/00 72:00
array=1 implies OperativeSystemManagement.getCurrentDateAndTime() - a.getDateAndTime() >= 0000/00/00 48:00

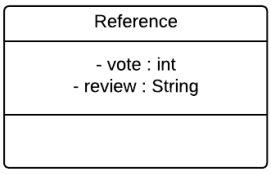
3.6 Offerer



Per effettuare l'eliminazione dell'account offerente è necessario che tale utente non abbia annunci pubblicati ancora attivi, ovvero non deve avere annunci la cui data di svolgimento sia successiva alla data in cui viene richiesta l'eliminazione dell'account. Questa condizione viene espressa tramite una pre-condizione sul metodo `removeAccount(date : Date)` che, come si evince dal nome, permette l'eliminazione dell'account dal sito *StayBusy*.

context Offerer :: removeAccount(date: Date)
pre: self.printValidAnnouncements() = nil

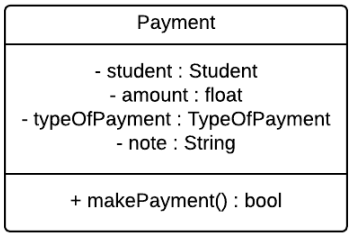
3.7 Reference



Una referenza è espressa attraverso un voto che è compreso tra 0 e 5. Questa condizione è espressa in OCL attraverso un invariante nel seguente modo:

```
context Reference inv:
vote >= 0 AND vote <= 5
```

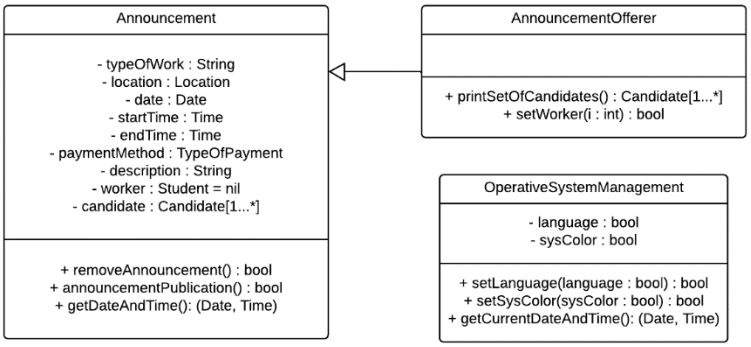
3.8 Payment



L'applicazione fornisce due diverse modalità di pagamenti: in contanti oppure pagamento elettronico attraverso l'utilizzo del sistema esterno *PayPal*. In entrambi i casi è però presente il vincolo che l'amount del pagamento deve essere sempre non nullo. Per questo motivo, il metodo `makePayment()` della classe **Payment** deve restituire come valore booleano *true*, ovvero permette di conseguire correttamente il pagamento solo nel caso in cui il valore del pagamento è maggiore di zero. Ciò può essere espresso in OCL attraverso l'utilizzo di una pre-condizione nel seguente modo:

```
context Payment :: makePayment()
pre: self.amount > 0
```

3.9 Announcement



L'orario di inizio di un servizio esplicitato all'interno di un annuncio deve sempre essere inferiore all'orario di fine. Questa condizione è espressa in OCL attraverso un invariante nel seguente modo:

context Announcement inv:
startTime < endTime

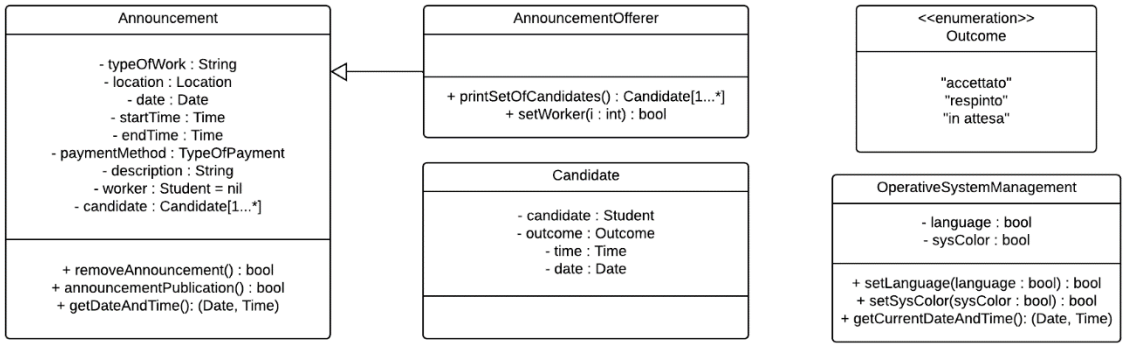
Al momento della creazione di un nuovo annuncio, l'offerente deve inserire opportunamente diverse informazioni, tra cui è presente la fascia oraria del lavoro (ora inizio – ora fine). Su questo l'applicazione imposta il vincolo che l'orario di inizio servizio segnato dall'offerente sia sempre maggiore di quello attuale. Ciò può essere espresso in OCL attraverso l'utilizzo di una pre-condizione nel seguente modo:

context Announcement :: announcementPublication()
pre: (self.date, self.startTime) > OperativeSystemManagement.getCurrentDateAndTime()

Per quanto riguarda l'eliminazione degli annunci, è imposta la condizione che, nelle ventiquattro ore prima dello svolgimento del servizio, un offerente non ha la possibilità eliminare un annuncio se a questo è associato uno studente che si era precedentemente candidato e che era stato accettato dallo stesso offerente. Queste condizioni sono espresse in OCL attraverso un invariante in questo codice:

context Announcement :: removeAnnouncement()
pre: self.candidates = nil
AND OperativeSystemManagement.getCurrentDateAndTime().second - Announcement.startTime >= 24:00

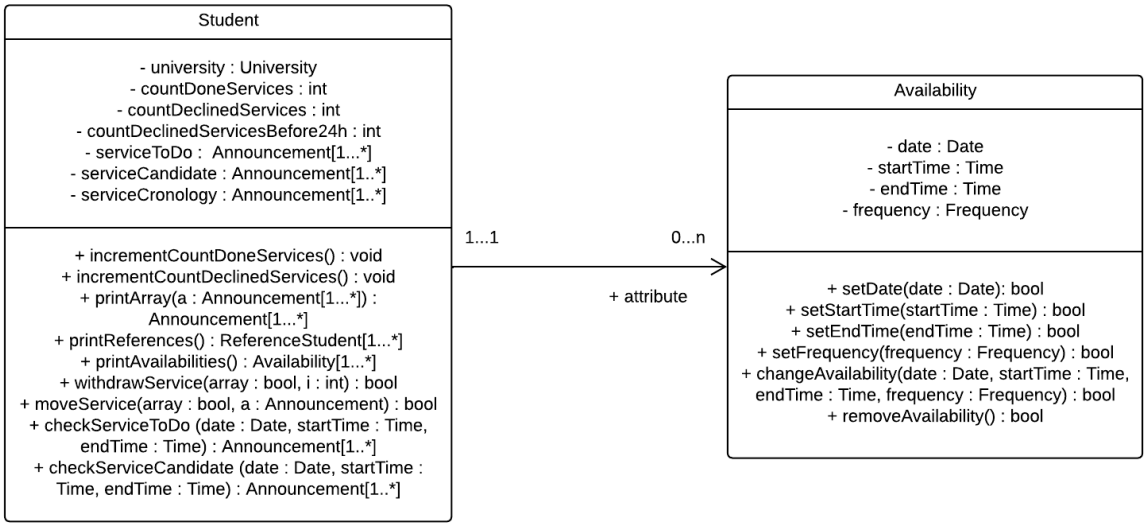
3.10 AnnouncementOfferer



L'offerente può accettare un solo studente per Announcement; questo vincolo viene espresso tramite una pre-condizione (vedi OCL di seguito). Inoltre, quando viene selezionato un candidato tra la lista di candidati *candidate,* il sistema deve settare gli *outcome* (classe **Candidate**) come "respinto" (tipo di dato **Outcome**) degli altri studenti che si sono candidati all'annuncio ed effettuare l'invio della relativa e-mail di "notifica rifiuto da un annuncio". Queste azioni possono essere espresse in OCL come post-condizioni del metodo `setWorker(i : int)`.

context AnnouncementOfferer :: setWorker(i : int)
pre: worker = nil
post: self.Candidate[0..*] -> forAll(c<>self.Candidate[i] | c.outcome="respinto"
AND EmailManagement.sendNotificationRequestRefusal(c))
AND self.Candidate[0..*] -> forAll(c | c.time = OperativeSystemManagement.getCurrentDateAndTime().second)

3.11 Availability



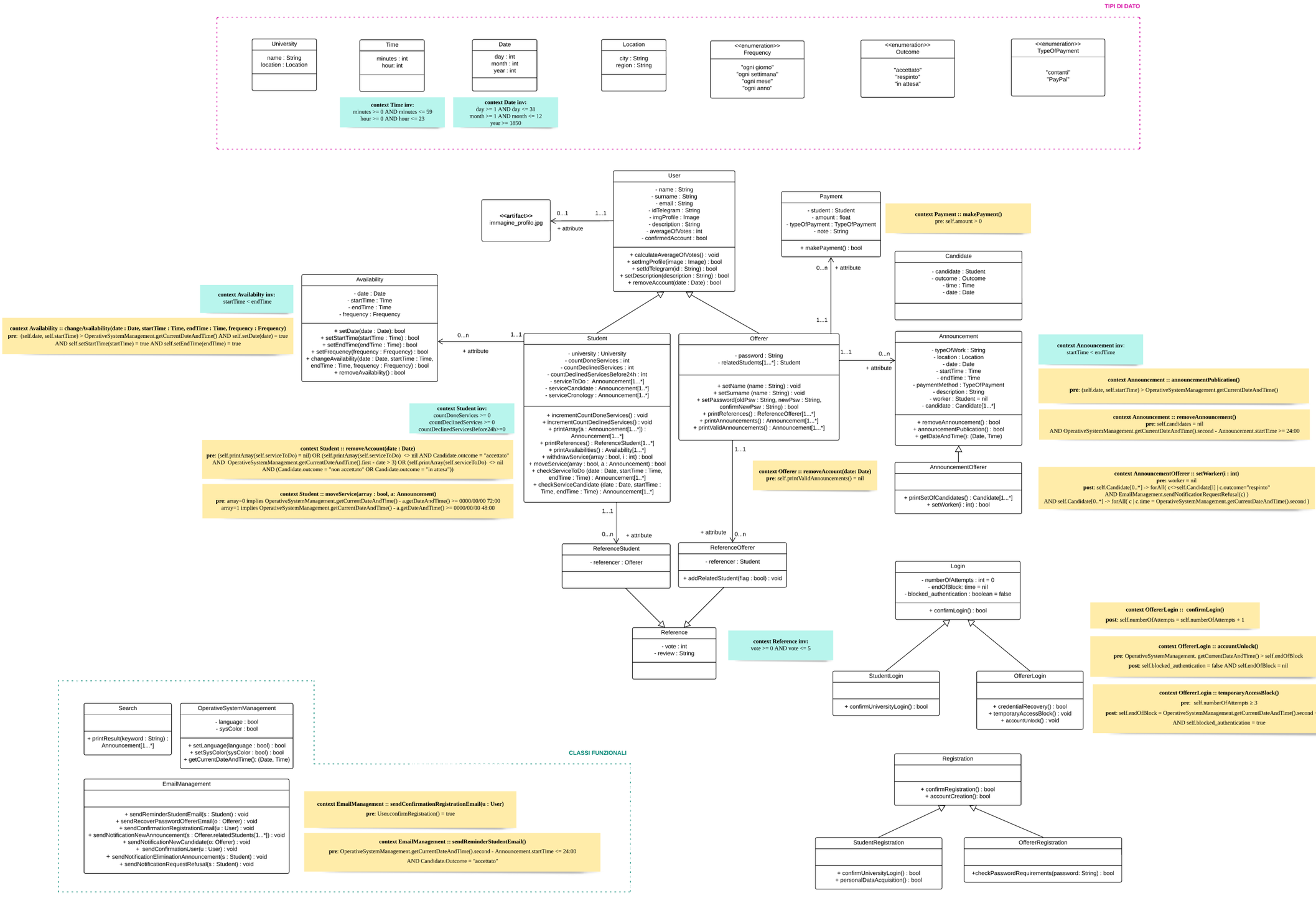
Una disponibilità temporale deve sempre essere rappresentata da un orario di inizio inferiore all'orario di fine. Questa condizione è espressa in OCL attraverso un invariante nel seguente modo:

context Availabilty inv:
startTime < endTime

Dopo aver effettuato il primo accesso, lo studente deve inserire le proprie disponibilità temporali ovvero il/i giorno/i e l'orario di disponibilità. Queste possono essere modificate in qualsiasi istante; l'unica condizione impostata dall'applicazione è che l'orario di disponibilità segnato dallo studente sia sempre maggiore di quello attuale. Nello specifico, se il giorno di disponibilità inserito è quello attuale, l'orario di inizio deve necessariamente essere maggiore dall'orario presente, mentre, se il giorno di disponibilità inserito è successivo a quello attuale, non c'è alcun vincolo riguardo all'orario di inizio. I metodi `set *` della classe - ad esempio `setDate(date : Date), setStartTime(startTime : Time)` - servono invece per controllare che il dato passato in input sia stato inserito correttamente. Ciò può essere espresso in OCL attraverso l'utilizzo della seguente pre-condizione:

context Availability :: changeAvailability(date : Date, startTime : Time, endTime : Time, frequency : Frequency)
pre: (date = ((Date) OperativeSystemManagement.getCurrentDateAndTime())
AND startTime > ((Time) OperativeSystemManagement.getCurrentDateAndTime()))
OR (date > OperativeSystemManagement.getCurrentDateAndTime()) AND self.setDate(date) = true
AND self.setStartTime(startTime) = true AND self.setEndTime(endTime) = true

3.1 Diagramma delle classi con OCL complessivo



4. Note ed eventuali

I *diagrammi delle classi* e i codici in *Object Constraint Language* (OCL) sono stati realizzati interamente dal nostro gruppo tramite il sito www.lucidchart.com.

A causa della bassa definizione delle immagini dei Diagrammi delle classi complessivo e Diagrammi delle classi con OCL complessivo presenti nel documento, rendiamo disponibili le immagini originali qui.