

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](#_Toc118923077)

[2. Requisiti Funzionali Errore. Il segnalibro non è definito.](#_Toc118923078)

[2.1 Utente anonimo **Errore. Il segnalibro non è definito.**](#_Toc118923079)

[2.1.1 UML generale utente anonimo **Errore. Il segnalibro non è definito.**](#_Toc118923080)

[2.2 Utente studente **Errore. Il segnalibro non è definito.**](#_Toc118923081)

[2.2.1 UML generale utente studente **Errore. Il segnalibro non è definito.**](#_Toc118923082)

[2.3 Utente offerente **Errore. Il segnalibro non è definito.**](#_Toc118923083)

[2.3.1 UML generale utente offerente **Errore. Il segnalibro non è definito.**](#_Toc118923084)

[3. Requisiti Non Funzionali Errore. Il segnalibro non è definito.](#_Toc118923085)

[4. Analisi del Contesto Errore. Il segnalibro non è definito.](#_Toc118923088)

[3.1 Utenti e sistemi esterni **Errore. Il segnalibro non è definito.**](#_Toc118923089)

[3.2 Diagramma di contesto **Errore. Il segnalibro non è definito.**](#_Toc118923090)

[5. Analisi dei Componenti Errore. Il segnalibro non è definito.](#_Toc118923091)

[4.1 Definizione dei componenti **Errore. Il segnalibro non è definito.**](#_Toc118923092)

[4.2 Diagramma dei componenti **Errore. Il segnalibro non è definito.**](#_Toc118923093)

[6. Note ed eventuali Errore. Il segnalibro non è definito.](#_Toc118923094)

# 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), che permettere 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.

# 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 delle informazioni aggiuntive, al fine di rappresentare al meglio le relazioni tra di esse.

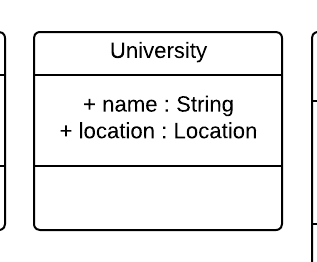
Nello specifico, di seguito vengono riportate le classi individuate dai diagrammi di contesto e delle componenti rappresentati nel documento **D1: Analisi dei requisiti**. Ognuna di queste è accompagnata da una breve descrizione di alcuni attributi e funzioni e, in alcuni casi, dal codice OCL, in modo 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 utilizzati dal sistema.

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.



1. 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, 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.



1. 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 - che verranno espressi direttamente nelle classi **Disponibility** e **Announcement**.

Immagine che contiene tavolo

Descrizione generata automaticamente

1. 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 questa funzionalità 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**.

Immagine che contiene tavolo

Descrizione generata automaticamente

1. Outcome

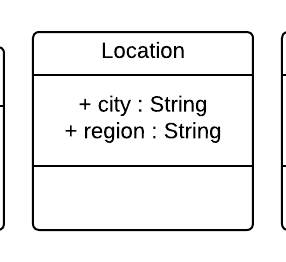
Il tipo di dato **Outcome** è stato creato per gestire lo stato delle richieste degli annunci. Questo tipo è un’enumerazione di tre valori - “Accettato”, “Respinto”, “In attesa” – attraverso cui l’utente può sapere qual è lo stato di una richiesta da lui effettuata per uno specifico servizio. Per questo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

Immagine che contiene tavolo

Descrizione generata automaticamente

1. Location

Il tipo di dato **Location** è stato creato per permettere il salvataggio di un luogo ed è costituito da due attributi, 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.



1. 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.

Immagine che contiene tavolo

Descrizione generata automaticamente

## 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.

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 (0) e italiano (1) mentre l'altro valore chiaro (0) e scuro (1).



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.

1. 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.

Immagine che contiene tavolo

Descrizione generata automaticamente

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

* sendConfirmationRegistrationEmail(u : User)permette l’invio di una notifica e-mail per confermare la registrazione dell'utente.
* 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.
* 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 e 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.
* 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.

1. 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.

Immagine che contiene tavolo

Descrizione generata automaticamente

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

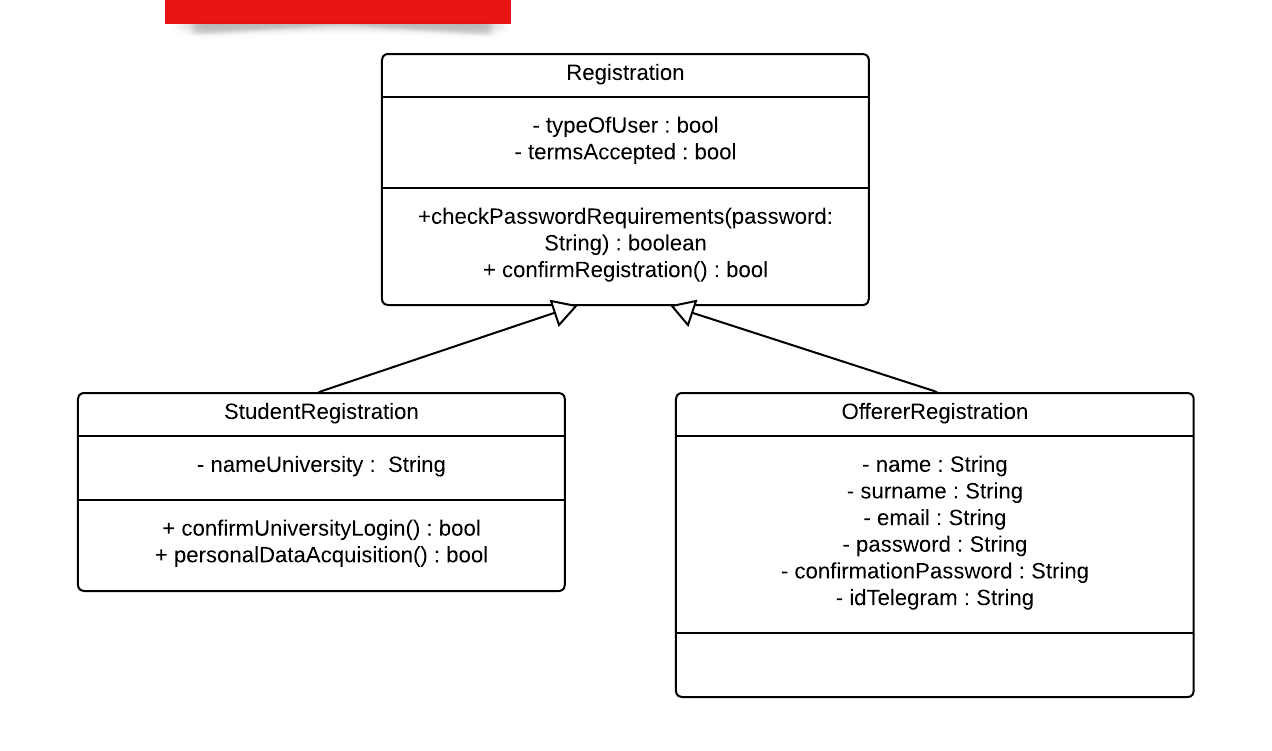
Di seguito vengono presentate le classi derivare dall’analisi del diagramma di contesto e dei componenti descritti nel documento **D2: Specifica dei Requisiti**.

1. Registration

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 **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 *StudentRegistration* e *OffererRegistration*.

La classe **Registration** presenta due attributi: *typeOfUser* è un booleano che rappresenta la tipologia di utente che intende registrarsi all'applicazione (Utente studente o Utente offerente). Per semplificare il progetto è stato deciso di indicare l'utente studente attraverso il valore true mentre l'utente offerente è rappresentato attraverso il valore false. Inoltre, *termsAccepted* è un attributo booleano che rappresenta lo stato dell'accettazione dei termini e condizioni.



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

* 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.
* confirmRegistration()varia in basa alla tipologia di utente su cui viene richiamata (vedi descrizione metodi classe **StudentRegistration** e classe **OffererRegistration**).

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.

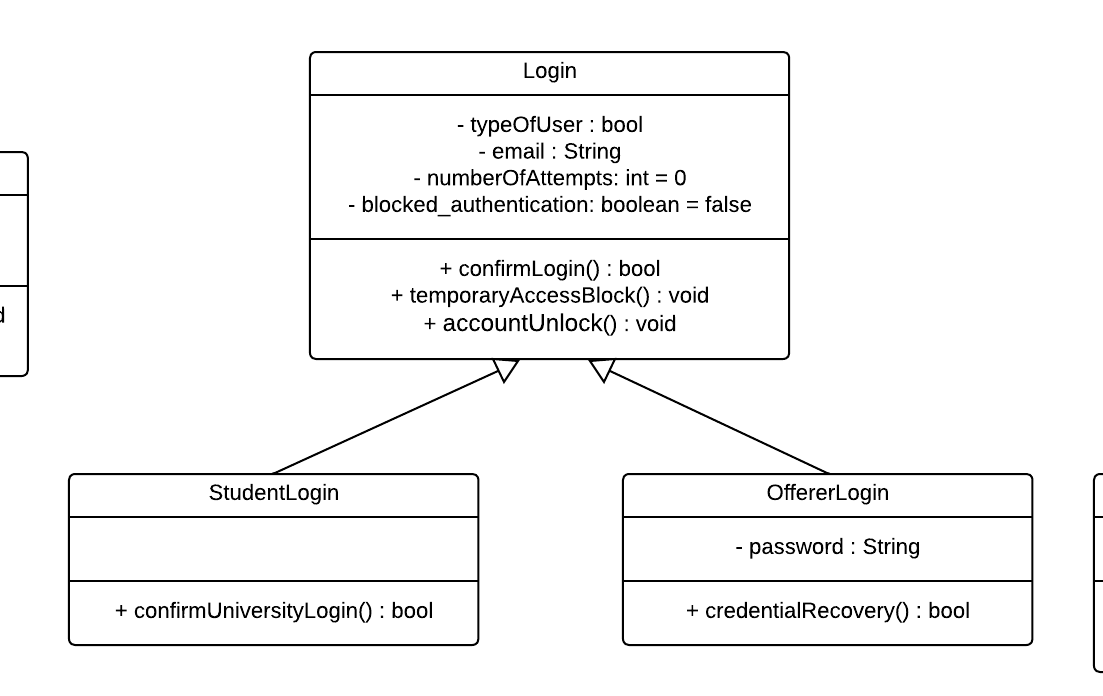
Di seguito una breve descrizione dei metodi implementati dalla classe **OffererRegistration:**

* confirmRegistration() si occupa di verificare che le credenziali inserite dall'utente offerente siano corrette.

1. 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: *typeOfUser* è la tipologia di utente che intende svolgere l'accesso all'applicazione (Utente studente o Utente offerente), *e-mail* è l'attributo che rappresenta l'indirizzo e-mail inserito al momento del login, *numberOfAttempts* sono il numero di tentavi svolti per l'inserimento corretto sella password, 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'account è sempre sbloccato. 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'account viene bloccato. Infine, *endOfBlock* è l'attributo rappresentante il tempo passato dal momento in cui è stato bloccato l'account.



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 **OffererinLog**).
* temporaryAccessBlock() gestisce il blocco dell'account nel caso in caso di inserimento errato della password per due 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.

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

* confirmRegistration() si occupa di verificare che le credenziali inserite dall'utente offerente siano corrette.

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.

1. 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 utilizzare 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**.

MANCANO ATTRIBUTI

Immagine che contiene testo, ricevuta, screenshot

Descrizione generata automaticamente

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() 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.
* printCronologyOfServices() permette di stampare la lista dei lavori per cui l’utente studente si è prenotato, e relative caratteristiche, restituendo un array di *Service*.
* 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(service : Service) permette all’utente studente di ritirarsi da un servizio a cui si era candidato. Questo metodo restituisce un *bool* per confermare l’avvenuta procedura.

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

* setName(name : String) permette di inserire/modificare il nome del profilo.
* setSurname() permette di inserire/modificare il cognome del profilo.
* setPassword() 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*.

1. 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.

MANCANO ATTRIBUTI

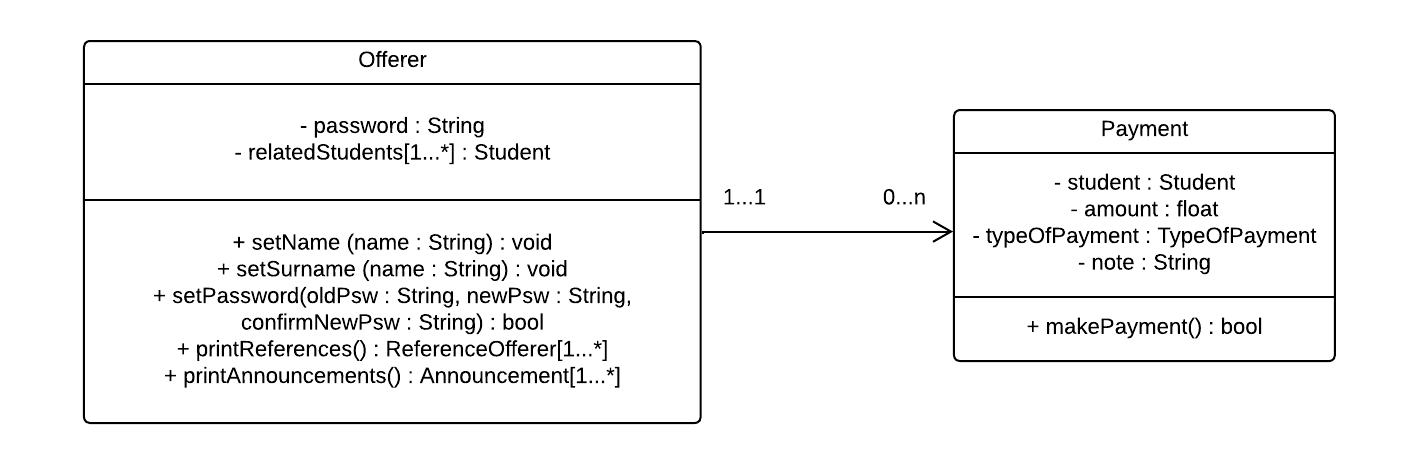
Di seguito una breve descrizione dei metodi implementati dalla classe **ReferenceOfferer**:

* addRelatedStudent(flag : bool) permette, in caso di valore 1 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.

1. 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.



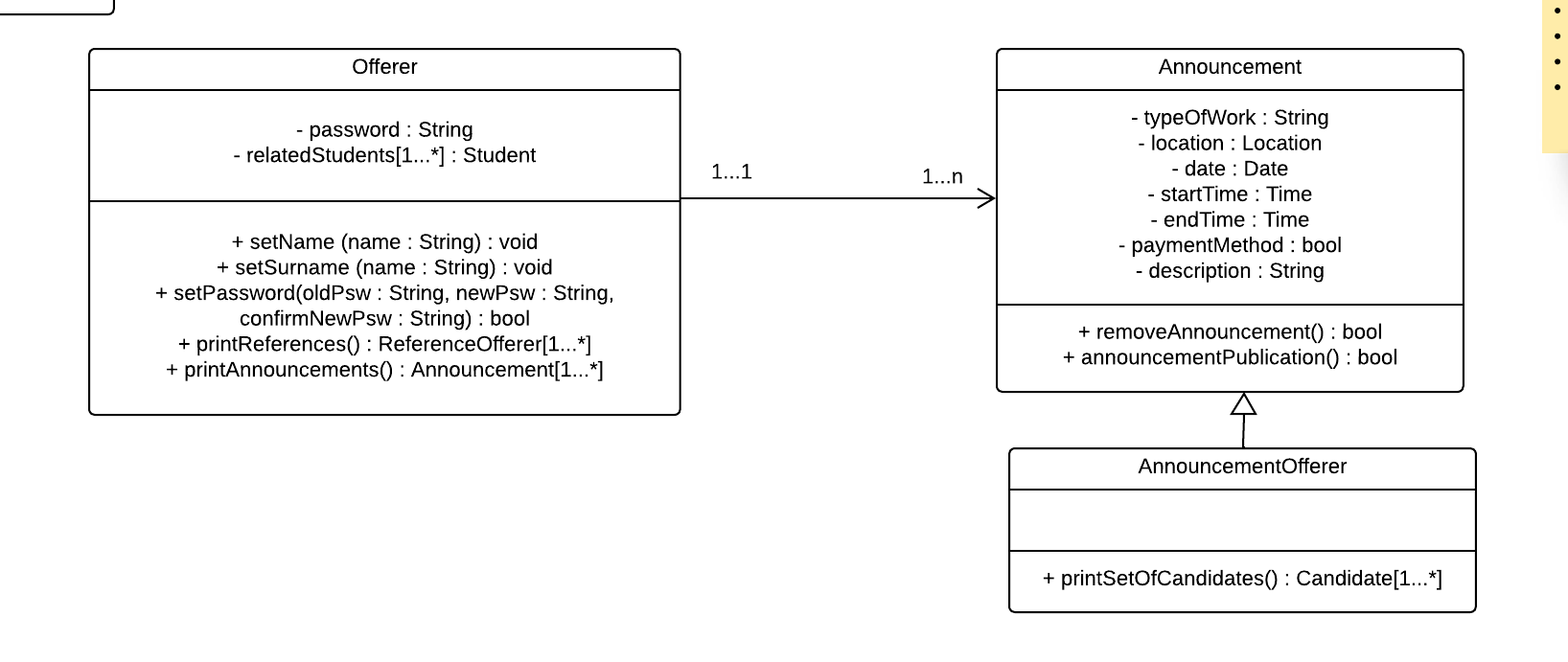
Di seguito una breve descrizione dei metodi implementati dalla classe **Payment**:

* 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).

1. 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**.

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 mentre *description* è la breve descrizione del servizio.



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

* removeAnnouncement() gestisce l'eliminazione di un annuncio attivo.
* annuncecementPublication() segnala se la pubblicazione di un nuovo annuncio è andata a buon fine

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.

!!! DA FARE OCL

1. 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* chemostra tutti gli annunci per cui lo studente si è reso disponibile. Attraverso l'attributo *outcome* è invece possibile visionare lo stato di tali annunci (accettato, non ancora accettato, rifiutato).

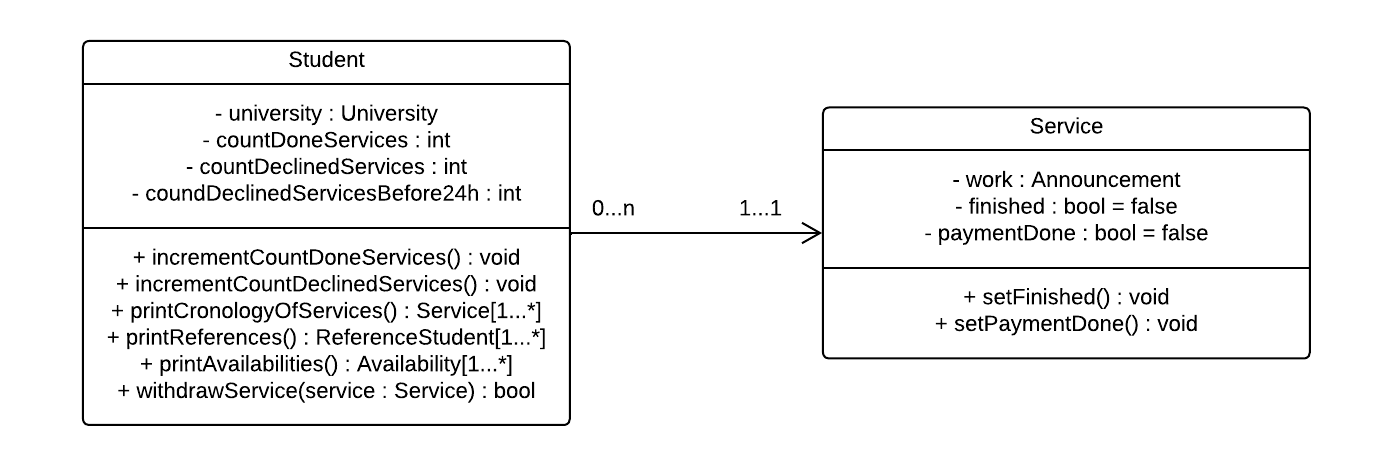
Immagine che contiene tavolo

Descrizione generata automaticamente

1. Service

Dall’analisi dei componenti **5.2.6 Visualizza annuncio** e **5.2.11 Cronologia lavori effettuati** si evince la necessità di creare una classe **Service** per la gestione degli annunci a cui uno studente si è candidato e per cui è stato selezionato dall’offerente. Tale classe è infatti in relazione con la classe **Student**.

La classe **Service** presenta diversi attributi: *work* permette di memorizzare l’annuncio di lavoro dell’offerente al quale ci si riferisce, *finished* mantiene lo stato del servizio (completato/non completato) e *paymentDone* mantiene lo stato del pagamento (effettuato/non effettuato).



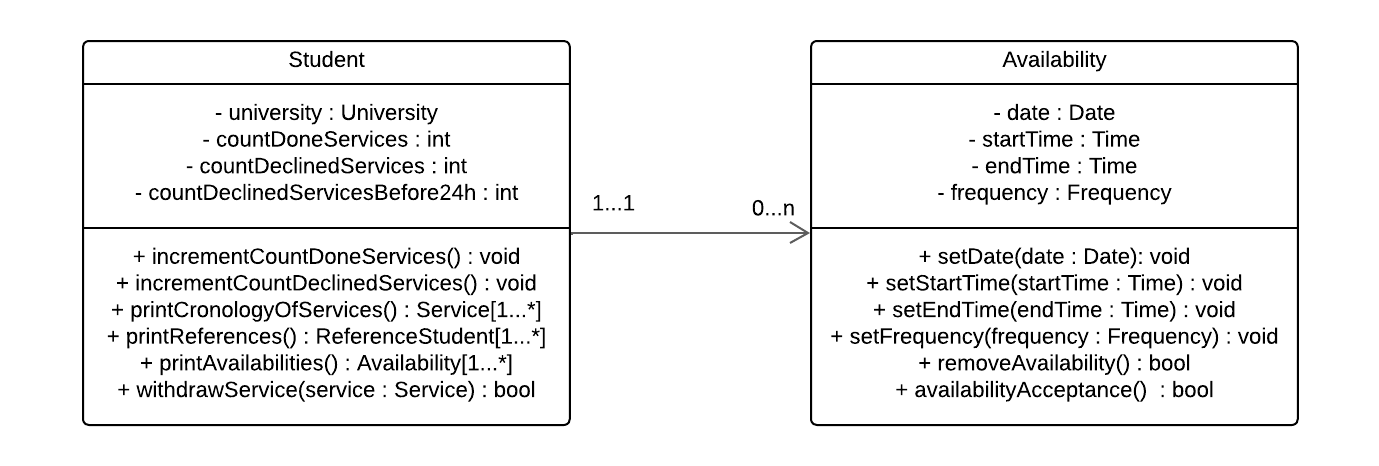
Di seguito una breve descrizione dei metodi implementati dalla classe **Service**:

* setFinished() permette di settare il campo *finished* a “true” quando il servizio è stato completato.
* setPaymentDone() permette di settare il campo *paymentDone* quando il pagamento è stato ricevuto.

1. 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*.
* removeAvailability() permette di eliminare tale disponibilità dal sistema.
* availabilityAcceptance() segnala se l'impostazione delle disponibilità è andata a buon fine.

## 2.4 Diagramma delle classi complessivo

# Codice in Object Constraint Language

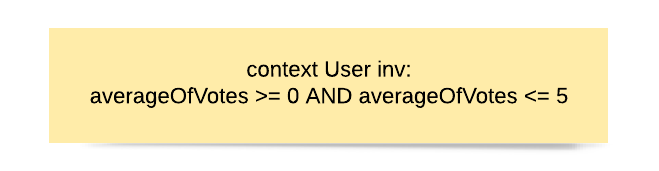
In questa sezione descriveremo la logica prevista in alcune operazioni di certe classi. Questa logica verra descritta in Object ` Constraint Language (OCL) e ricorriamo a questo strumento dal momento che non tutti i concetti sono esprimibili in modo formale in UML.

**Gestione utenti**

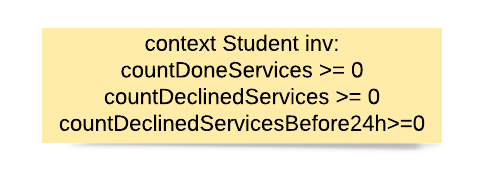
Immagine che contiene testo, ricevuta, screenshot

Descrizione generata automaticamente

Un utente deve possedere una media dei voti sempre compresa tra 0 e 5. Questa condizione è espressa in OCL attraverso un inviariante con questo codice:



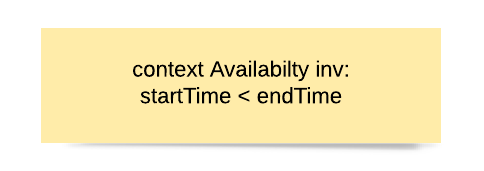
Uno studente deve avere sempre il numero di servizi effettuati e il numero di servizi respinti (oltre e prima delle 24 ore) pari o maggiore a 0. Queste condizioni sono espresse in OCL attraverso un invariante con questo codice:



**Gestione disponibilità**



Una disponibilità temporale deve avere sempre un orario di inizio inferiore all’orario di fine. Questa condizione è espressa in OCL attraverso un invariante con questo codice:



**Gestione annunci**



Un annuncio deve avere sempre un orario di inizio inferiore all’orario di fine. Questa condizione è espressa in OCL attraverso un inviariante con questo codice:

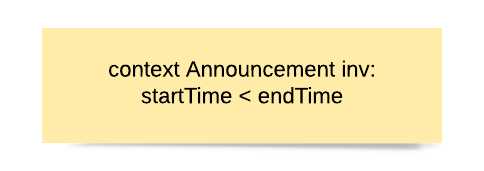


Immagine che contiene testo

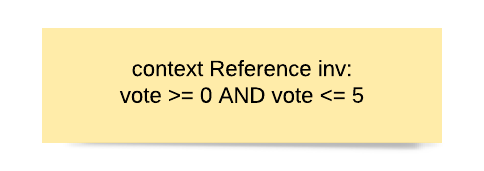
Descrizione generata automaticamente

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 precondizione nel seguente modo:



**Gestione** **referenze**

Una referenza deve avere sempre un voto compreso tra 0 e 5. Questa condizione è espressa in OCL attraverso un invariante in questo codice:



**Gestione date**

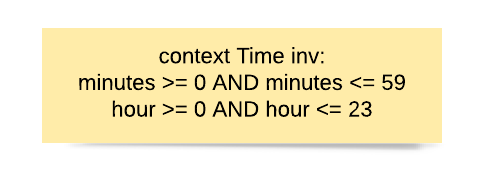
Una data deve essere caratterizzata sempre da un giorno compreso tra 1 e 31, un mese compreso tra 1 e 12 e un anno maggiore o uguale a 1850. Queste condizioni sono espresse in OCL attraverso un invariante in questo codice:

Immagine che contiene testo

Descrizione generata automaticamente

**Gestione orario**

Un orario deve essere caratterizzato sempre dai minuti compresi tra 0 e 59 e dalle ore comprese tra 0 e 23. Queste condizioni sono espresse in OCL attraverso un invariante in questo codice:



**Gestione pagamenti**

Immagine che contiene tavolo

Descrizione generata automaticamenteImmagine che contiene testo

Descrizione generata automaticamente

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 precondizione nel seguente modo:

Immagine che contiene testo

Descrizione generata automaticamente

**Gestione autenticazione**

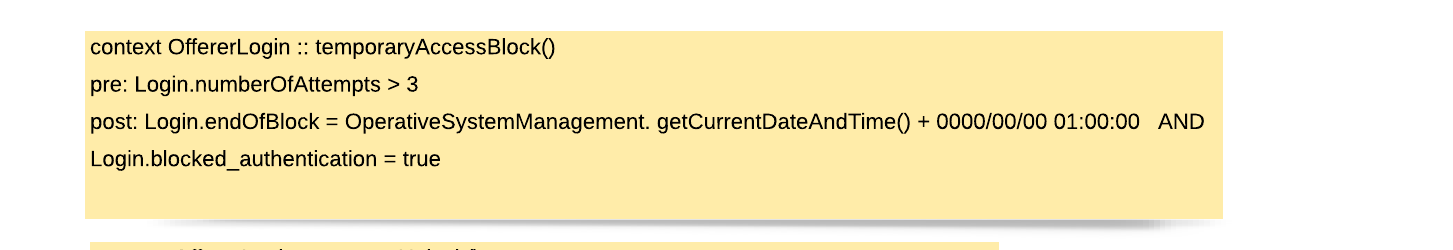
Nell'eventualità in cui l'utente offerente prova ad accedere all'applicazione inserendo la password errata, si verifica un incremento della variabile *numberOfAttempts* di un'unità. Ciò può essere espresso in OCL attraverso l'utilizzo di una postcondizione nel seguente modo:

Immagine che contiene testo

Descrizione generata automaticamente



Quando la variabile *numberOfAttempts* eguaglia il valore tre il sito si blocca per un'ora. Ciò può essere espresso in OCL attraverso l'utilizzo di una precondizione e una postcondizione nel seguente modo:



Un ulteriore decisione presa è stata quella di chiamare il 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 precondizione e una postcondizione nel seguente modo:

Immagine che contiene testo

Descrizione generata automaticamente

**Gestione notifiche e-mail**

Immagine che contiene tavolo

Descrizione generata automaticamente

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 precondizione nel seguente modo:

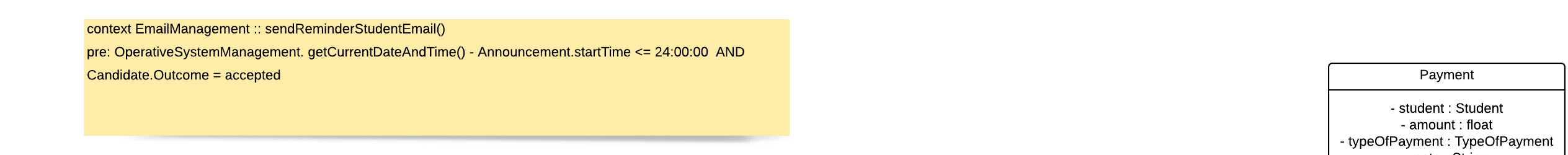


Immagine che contiene tavolo

Descrizione generata automaticamente

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. Per quanto riguarda i metodi set della classe- ad esempio setDate(date : Date), setStartTime(startTime : Time) - questi servono per controllare che il dato passato in input è stato inserito correttamente. Ciò può essere espresso in OCL attraverso l'utilizzo della seguente precondizione:

Immagine che contiene testo

Descrizione generata automaticamente