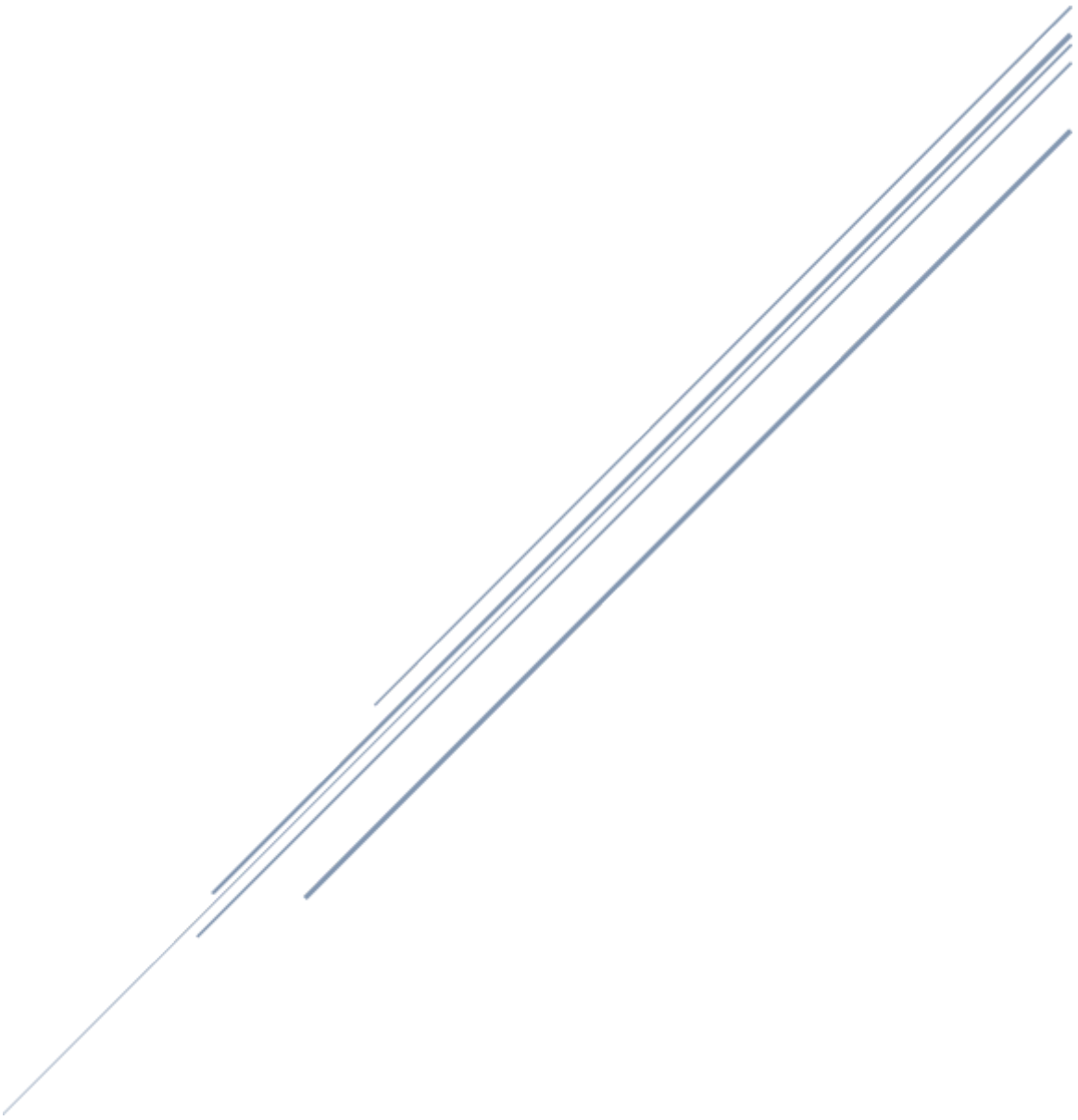
SOLVE.IT

Documento di architettura



Gruppo T053
Ingegneria del Software

Sommario

Scopo del documento

Il seguente documento presenta la modellazione dell'architettura per il progetto Solve.it mediante Class Diagram definiti nel linguaggio UML (Uniform Modeling Language) e l'ausilio di vincoli, espressi dove necessario, in linguaggio OCL (Object Constraint Language). Di seguito sono riportate le classi che modellano l'architettura di Solve.it suddivise per macro-funzionalità del sistema. In seguito alla specifica delle classi vengono riportati i vincoli sul loro comportamento opportunamente definiti in linguaggio OCL.

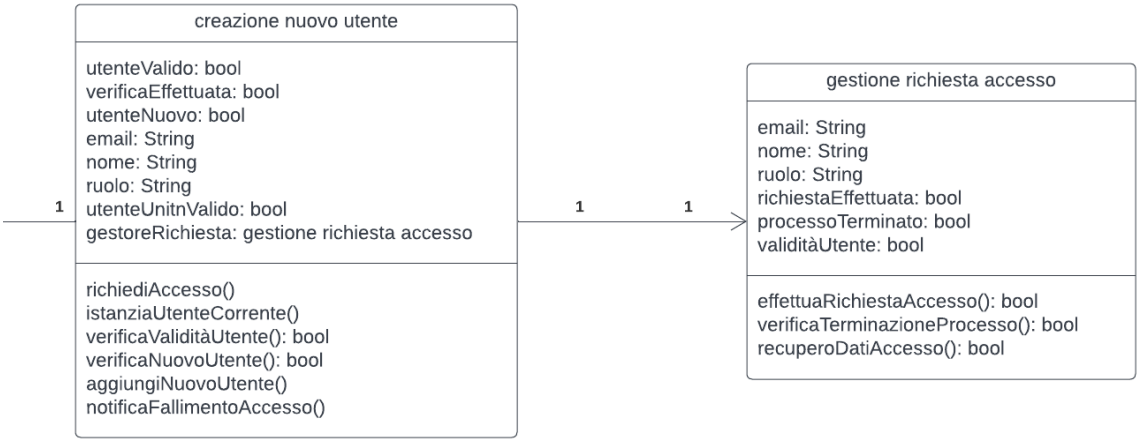
Diagramma delle classi

Autenticazione alla piattaforma

Osservando il diagramma delle componenti per il progetto Solve.it si è resa evidente la necessità di un interfacciamento alle API di Unitn per delegare l'accesso alla piattaforma. Per gestire la non-sincronia del processo di delega alle API e separare la funzionalità di delega dalla gestione degli utenti a livello locale al sistema sono state create le seguenti classi:

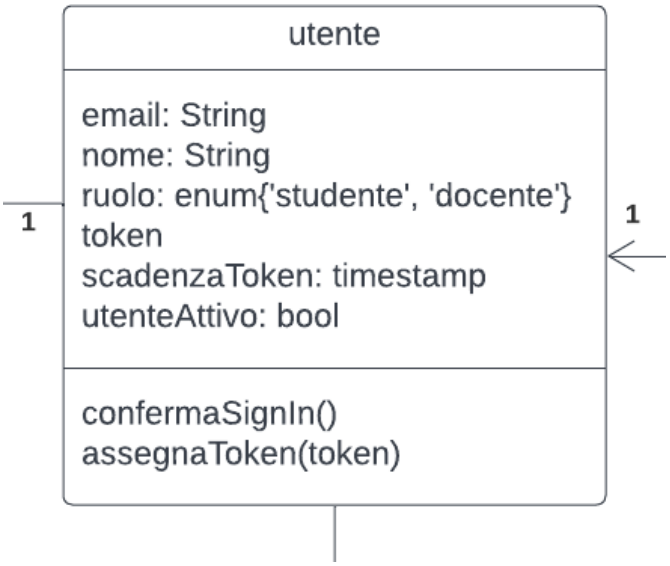
- "**gestione richiesta accesso**": questa classe si occupa di comunicare con le API di Unitn per ottenere i dati dell'utente che sta provando ad accedere alla piattaforma.

- "**creazione nuovo utente**": questa classe si occupa di certificare ed istanziare nella piattaforma gli utenti che effettuano correttamente l'accesso ad Unitn, verificando la conformità con i requisiti di ruolo. Nel caso si verifichi la presenza di un utente ancora non memorizzato nei record della piattaforma, questa classe si occupa di aggiungere un nuovo record utente.



Utenti del sistema

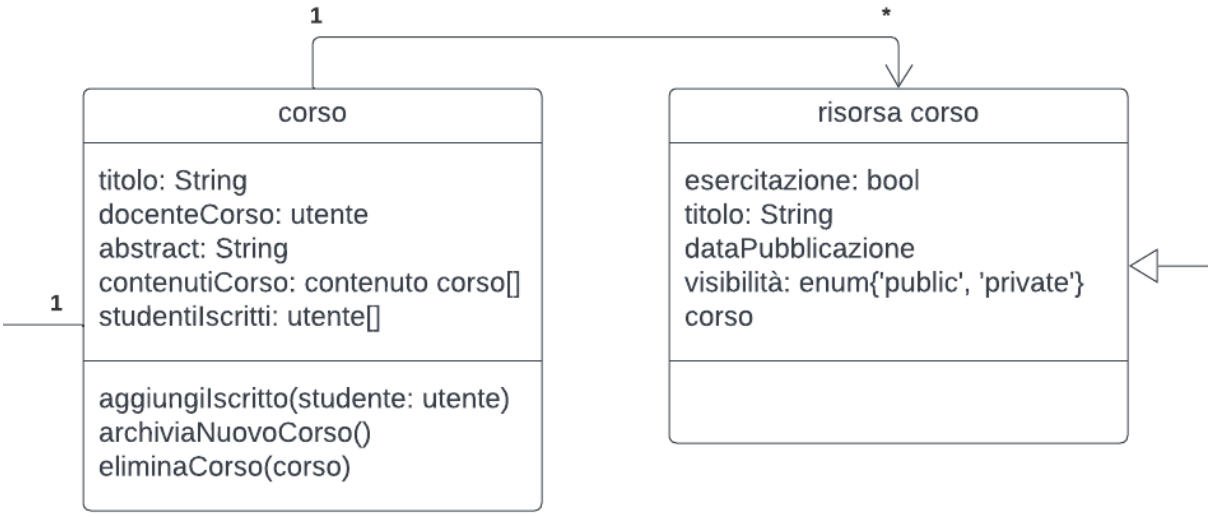
È stato deciso di creare la classe “**utente**”, per modellare un utente della piattaforma. Ciascun utente è identificato dalla propria email, nome e ruolo in Solve.it (‘studente’ o ‘docente’). Tutti gli utenti hanno anche un token necessario ad identificare le richieste che invieranno al back-end.
La classe utente contiene inoltre un metodo che conferma all’utente che ha effettuato correttamente il sign-in, ed un metodo che ne assegna o ri-assegna un token. Un’istanza della classe “**utente**” rappresenta nell’ambiente front-end l’utente attivo (a meno di non-validità del token). Se l’autenticazione al sistema non è terminata correttamente non viene creata alcuna istanza della classe “**utente**”.



Corsi

Dal diagramma delle componenti è evidente la necessità di creare interfacce per gestire i corsi e i contenuti nei un corsi, ovvero esercitazioni e risorse multimediali. Per la gestione di questa funzionalità sono state create le seguenti classi:

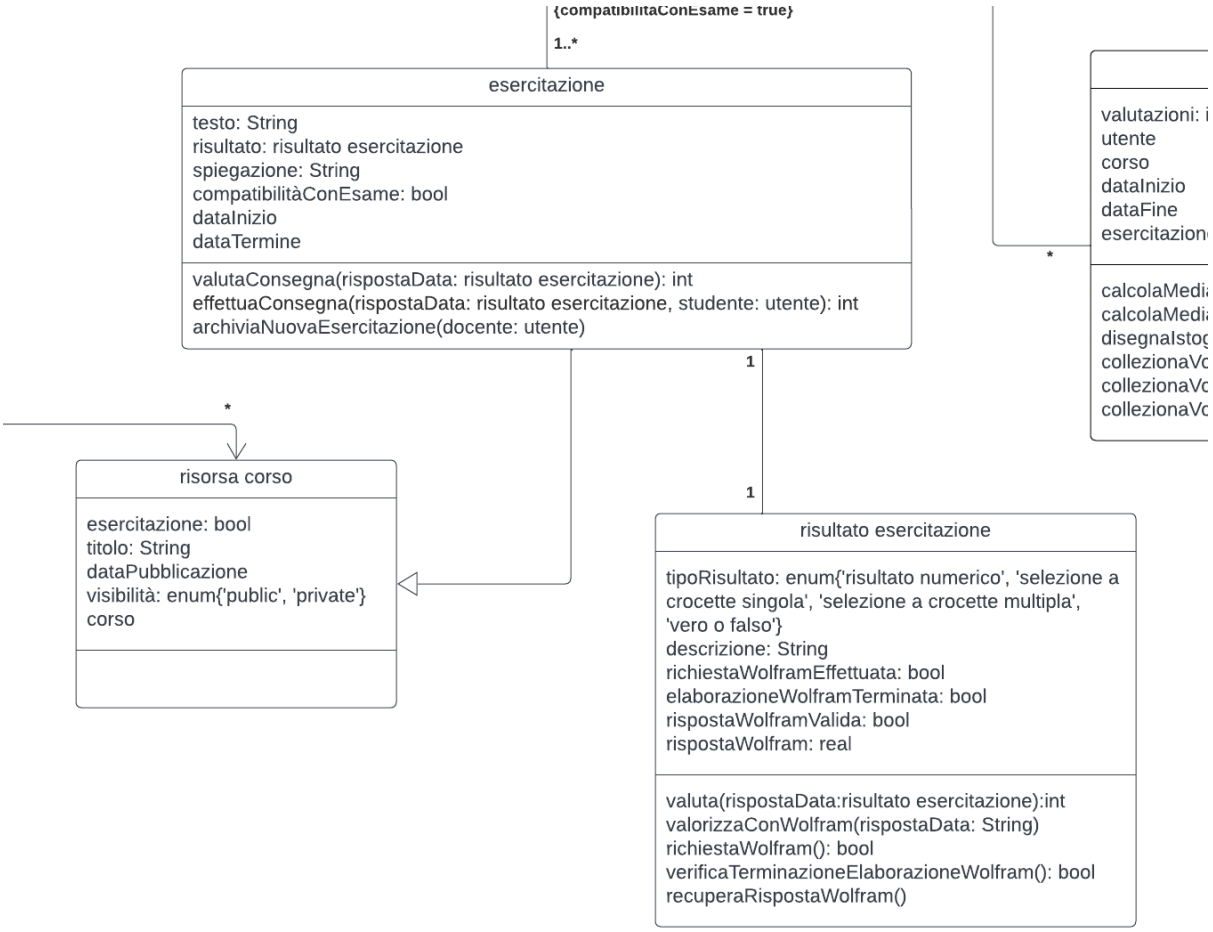
- “**corso**”: definisce il corso nel sistema. Ogni corso appartiene ad un docente che lo ha creato, ha un titolo, un abstract, una lista di contenuti, e una lista di studenti iscritti al corso. La classe corso contiene inoltre i metodi per aggiungere iscritti ad un corso, creare un corso o eliminarne uno già esistente.
- “**risorsa corso**”: rappresenta un contenuto generico di un corso, tutti i contenuti hanno un titolo, una data di pubblicazione, un attributo che ne specifica la visibilità, ed uno che specifica se quel contenuto è un'esercitazione.



Esercitazioni

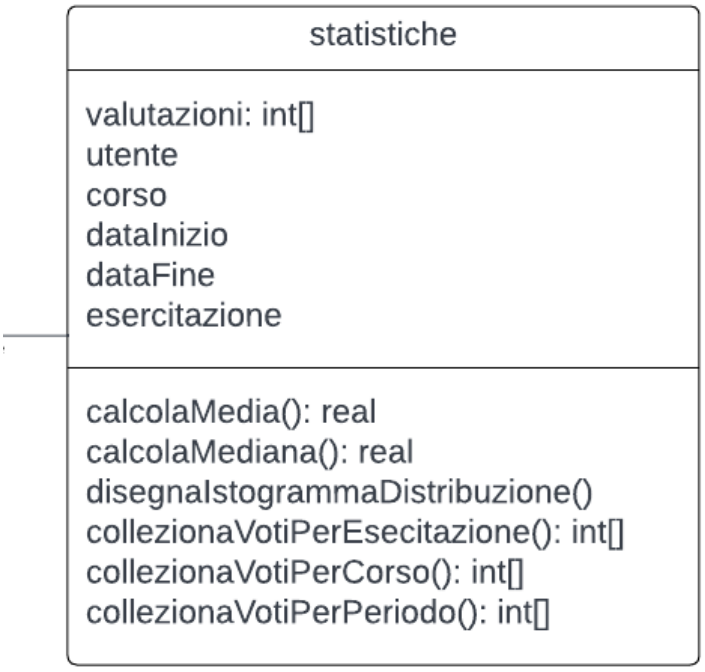
Dal diagramma delle componenti si evince la necessità di creare interfacce per la creazione, l'eliminazione e la valutazione di esercitazioni, così come per lo svolgimento di queste esercitazioni da parte degli studenti. Per la gestione di queste funzionalità sono state create le seguenti classi:

- "**esercitazione**": modella un'esercitazione all'interno del sistema. Contiene i metodi per consegnare e valutare una risposta, e consentire ad un docente di creare una nuova esercitazione in un corso.
- "**risultato esercitazione**": ogni esercitazione può accettare risultati sotto forma di risposte a crocette, vero o falso, risposta numerica, … (**RF4**). La classe "risultato esercitazione" è la classe che definisce il metodo "valuta" che permette di valutare una certa risposta in base al relativo tipo.
  La classe mette a disposizione una serie di metodi che consentono di effettuare una richiesta alle API di wolfram per calcolare il valore numerico di un'espressione espressa in formato testuale



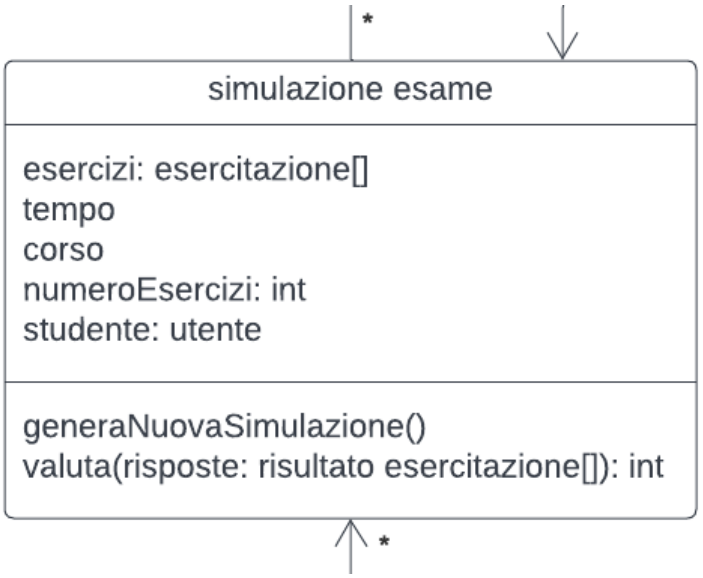
Statistiche

Per soddisfare la necessità di fornire a docenti e studenti statistiche appropriate, come definito dai requisiti funzionali RF, è stata creata la classe “**statistiche**”, che effettua dei calcoli (media, mediana e distribuzione) su un array di valutazioni. La classe inoltre valorizza tale array a seconda delle specifiche necessità (ad esempio se le statistiche vanno fornite ad uno studente può raccoglierne i voti nell’arco di un periodo).

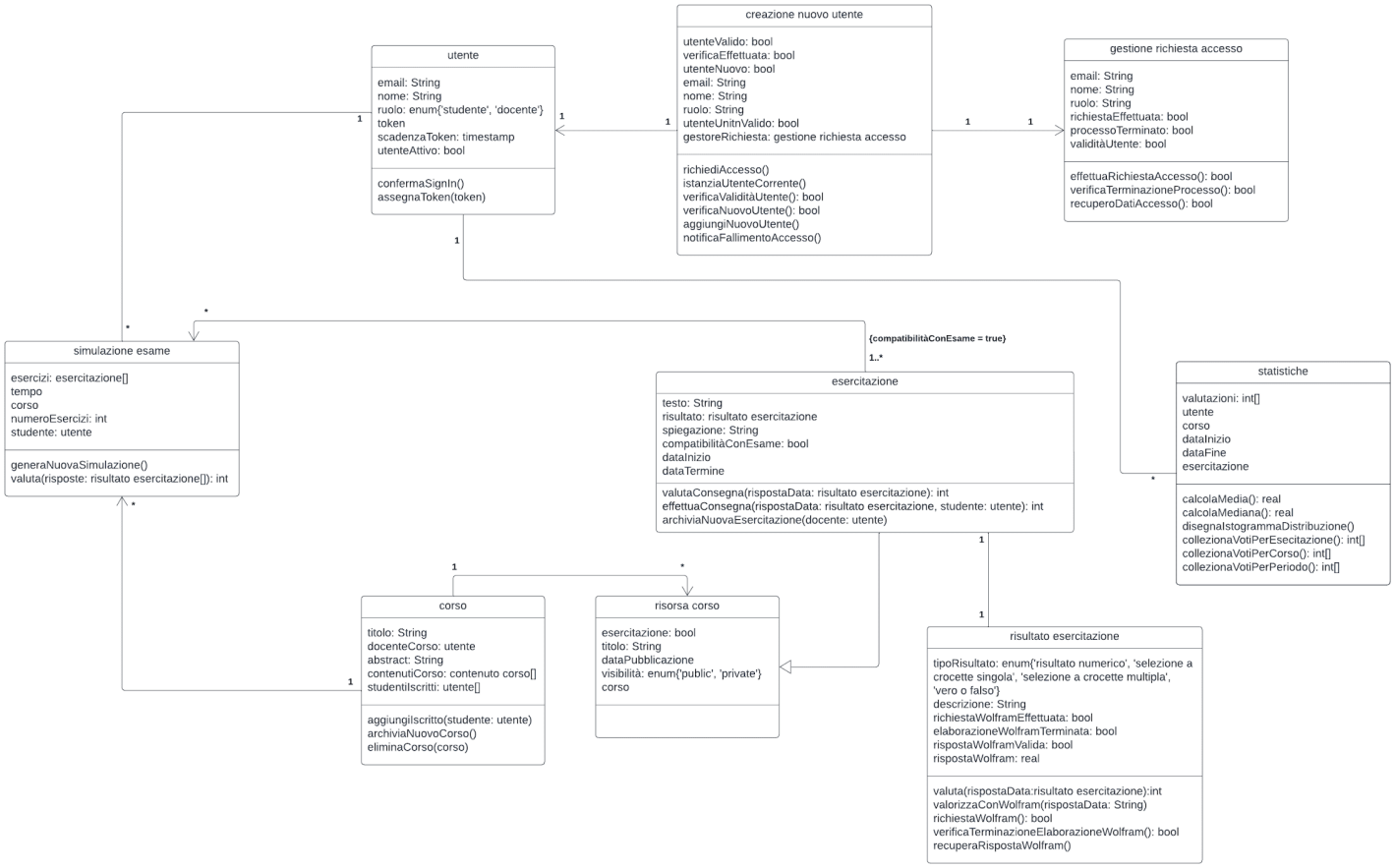


Simulazioni d'esame

Per implementare le funzionalità di creazione, svolgimento e correzione delle simulazioni d'esame (come da **RF22**) è stata creata la classe "**simulazione d'esame**", che permette di generare e rappresentare all'interno del sistema una simulazione d'esame e successivamente valutarla.



Visione complessiva del diagramma



Codice in linguaggio Object Constraint Language

In questo capitolo viene descritta in modo formale, e laddove opportuno, la logica del comportamento nell'ambito di alcuni metodi e di alcune classi sopra specificate. Tale logica viene espressa in linguaggio OCL perché il solo linguaggio UML per la descrizione delle classi non ha una semantica sufficiente a descrivere opportunamente tali condizioni sulla business logic dell'architettura. Per ogni circostanza che richieda la specifica di condizioni in linguaggio OCL di seguito è riportato un paragrafo che presenta la classe/le classi interessata/interessate e le condizioni OCL adottate per descriverne la logica.

Gestione richiesta di accesso

Quando viene effettuata una richiesta di accesso poiché non si possono eseguire contemporaneamente più richieste è necessario tenere conto di quando sta venendo effettuata una certa richiesta alle API. Per gestire questo processo è necessario che siano verificate le seguenti condizioni:

context gestione richiesta accesso::effettuaRichiestaAccesso()
pre: richiestaEffettuata = false
post: richiestaEffettuata = true and processoTerminato = false

context gestione richiesta accesso::verificaTerminazioneAccesso()
pre: processoTerminato = false and richiestaEffettuata = true

context gestione richiesta accesso::recuperoDatiAccesso()
pre: richiestaEffettuata = true and processoTerminato = true and validitàUtente = true
post: richiestaEffettuata = false

Utente attivo

Ogni utente dopo essersi loggato riceve un token che si rinnova ogni venti minuti, le condizioni relative al token sone espresse dal seguente codice OCL:

context utente
inv: (utenteAttivo = true and NOW() < scadenzaToken) or utenteAttivo = false

context utente::assegnaToken(token)
pre: utenteAttivo = false
post: self.token = token and scadenzaToken = NOW() + 20min and utenteAttivo = true

Svolgimento sessione d’esame

Per poter svolgere una sessione d’esame un utente deve essere uno studente, è necessario inoltre che alla consegna il numero di risposte sia uguale al numero di domande.
la condizione è espressa con il seguente codice OCL.

context simulazione d'esame
inv: self.studente.ruolo = 'studente'

context simulazione d'esame::valuta(risposte: risultato esercitazione[])
pre: esercizi->size() = risposte->size()

Creazione e svolgimento esercitazione

Per poter creare un esercitazione un utente deve essere un docente, mentre per poter svolgere un'esercitazione deve essere uno studente.
inoltre per ogni esercitazione la data di pubblicazione deve essere precedente alla data di termine (a seguito della quale viene pubblicata la soluzione).

Queste condizioni sono espresse in OCL dal seguente codice:

context esercitazione
inv: dataInizio < dataTermine

context esercitazione::effettuaconsegna(rispostaData, studente: utente)
pre: studente.ruolo = 'studente'

context esercitazione::archiviaNuovaEsercitazione(docente: utente)
pre: docente.ruolo = 'docente'

Gestione chiamate alle API di wolfram

Se viene effettuata una chiamata alle API di wolfram il tipo della risposta da valutare deve essere numerico.

context risultato esercitazione::valorizzaConWolfram(...)
post: tipoRisultato = 'risultato numerico'

poiché non si possono eseguire contemporaneamente più richieste è necessario tenere conto di quando sta venendo effettuata una certa richiesta alle API. Per gestire questo processo è necessario che siano verificate le seguenti condizioni:

context risultato esercitazione::richiestaWolfram()
pre: richiestaWolframEffettuata = false
post: richiestaWolframEffettuata = true and elaborazioneWolframTerminata = false

context risultato esercitazione::verificaTerminazioneElaborazioneWolfram()
pre: elaborazioneWolframTerminata = false and richiestaWolframEffettuata = true

context risultato esercitazione::recuperaRispostaWolfram()
pre: elaborazioneWolframTerminata = true
post: richiestaWolframEffettuata = false

Statistiche

La data in cui inizia il periodo di cui calcolare le statistiche deve essere precedente alla data di fine di quest'ultimo, oppure entrambe possono essere settate a null per indicare di considerare tutte le valutazioni:

context statistiche
inv: (dataInizio = null and dataFine = null) or dataInizio < dataFine

Non è possibile calcolare la media o la mediana di zero voti:

context statistiche :: calcolaMedia()
pre: self.valutazioni->size() > 0

context statistiche :: calcolaMediana()
pre: self.valutazioni->size() > 0

Non è possibile calcolare statistiche se un elemento necessario a tale fine ha valore null:

context statistiche::collezionaVotiPerEsercitazione()
pre: utente != null and esercitazione != null

context statistiche::collezionaVotiPerCorso()
pre: utente != null and corso != null

context statistiche::coellezionaVotiPerPeriodo()
pre: utente != null and dataInizio != null and dataFine != null

Corso e gestione corsi

Il docente di un corso può essere solamente un utente con il ruolo di docente:

context corso
inv: docenteCorso.ruolo = 'docente'

Tutti gli iscritti ad un corso devono essere studenti:

context corso
inv: studentiIscritti->forAll(s | s.ruolo = 'studente')

Perchè un utente possa iscriversi ad un corso esso deve essere uno studente, dopo che esso si è iscritto l'elenco di studenti iscritti al corso dovrà contenerlo:

context corso::aggiungiIscritto(studente: utente)
pre: studente.ruolo = 'studente'
post: studentiIscritti->contains(studente)

Simulazione d'esame

Solo gli studenti possono creare simulazioni d'esame:

context simulazione d'esame
inv: self.studente.ruolo = 'studente'

Una simulazione d'esame può essere valutata solo se è stata data una risposta a tutti gli esercizi da cui essa è composta:

context simulazione d'esame::valuta(risposte: risultato esercitazione[])
pre: esercizi->size() = risposte->size()

Risultato esercitazione

Perchè la classe “risultato esercitazione” si coordini con le API di Wolfram per fornire il servizio di calcolo del valore numerico di funzioni matematiche è necessario gestire il controllo di sincronia nella fase di chiamata delle API. Di seguito è reso esplicito il codice ocl per esprimere tale controllo e l’intero processo di richiesta di servizio alle API di Wolfram:

context risultato esercitazione::valorizzaConWolfram(...)
post: tipoRisultato = 'risultato numerico'

context risultato esercitazione::richiestaWolfram()
pre: richiestaWolframEffettuata = false
post: richiestaWolframEffettuata = true and elaborazioneWolframTerminata = false

context risultato esercitazione::verificaTerminazioneElaborazioneWolfram()
pre: elaborazioneWolframTerminata = false and richiestaWolframEffettuata = true

context risultato esercitazione::recuperaRispostaWolfram()
pre: elaborazioneWolframTerminata = true
post: richiestaWolframEffettuata = false

Creazione nuovo utente

Durante tutto il “lifetime” di un’istanza della classe “creazione nuovo studente” gli attributi utili alla verifica della validità dell’utente in fase di creazione devono essere sincronizzati opportunamente con i relativi attributi contenuti nell’istanza “gestoreRichiesta” della classe “gestione richiesta accesso”:

context creazione nuovo utente
inv: utenteUnitnValido = gestoreRichiesta.validitàUtente

context creazione nuovo utente
inv: utenteUnitnValido = false or (utenteUnitnValido = true and email = gestoreRichiesta.email and nome = gestoreRichiesta.nome and ruolo = gestioneRichiesta.ruolo)

Il processo di autenticazione di un utente è la successione della richiesta di accesso alla classe “gestione richiesta accesso” (mediante l’istanza “gestoreRichiesta” ed il metodo “richiediAccesso”), di verifica della validità dell’utente (tramite il metodo “verificaValiditàUtente”) e di eventuale aggiunta di

un nuovo utente locale (se non già esistente nell’archivio, e tramite il metodo “aggiungiNuovoUtente”):

context creazione nuovo utente::richiediAccesso()
pre: self.gestoreRichiesta.richiestaEffettuata = false
post: self.gestoreRichiesta.processoTerminato = true

context creazione nuovo utente::verificaValiditàUtente()
pre: verificaEffettuata = false
post: verificaEffettuata = true

context creazione nuovo utente::verificaNuovoUtente()
pre: verificaEffettuata = true

context creazione nuovo utente::aggiungiNuovoUtente()
pre: utenteNuovo = true
post: utenteNuovo = false

Di seguito è espressa la condizione affinché un utente Unitn registrato correttamente possa essere ritenuto valido localmente in Solve.it:

context creazione nuovo utente
inv: utenteValido = false or (utenteValido = true and (ruolo = 'studente' or ruolo = 'docente'))

Nel caso di fallimento del processo di sign-in dell’utente viene inviata una relativa notifica a front-end:

context creazione nuovo utente::notificaFallimentoAccesso()
pre: gestoreRichiesta.processoTerminato = true and (utenteUnitnValido = false or (verificaEffettuata = true and utenteValido = false))

Visione complessiva del diagramma con OCL